**Table of Contents**

**Introduction 1**

**System Overview 4**

**UML Class Diagram 4**

**MVC Architecture Explanation 4**

**Specification and Assertions 4**

**Testing Summary 4**

**Project Summary and Reflection 4**

**Appendix - Key Source Code 4**

1. **Introduction**

This project implements a Battleships game, including both command line interface and graphical interface implementations (CLI and GUI). The project adopts MVC architecture to manage the game logic, interface presentation and user interaction separately to ensure the maintainability and extensibility of the system.

1. **System Overview**

Development Language: Java 8

Graphical Interface: Swing

Organization: Model / View / Controller / CLI

Functionality Overview:

* 10x10 board
* Randomly placed fleets
* Attacks by coordinates, hits and misses
* Records when hit, ends game when fleet is completely out of water

1. **UML Class Diagram**

This class diagram shows the structure of the main classes of the system, including the MVC hierarchy, inheritance, dependencies and connections between classes.

**图示

AI 生成的内容可能不正确。**

1. **MVC Architecture Explanation**
   1. **Explanation of MVC architecture in Battleships games**

This Battleships game project strictly follows the MVC (Model-View-Controller) architecture, which realizes a clear hierarchical separation and ensures that the system is maintainable, extensible and testable.

Model

- Model class is responsible for encapsulating the core logic of the game, including Board, Ship, and tries.

- Model is inherited from Observable and actively notifies the observer (i.e. View) when the game state changes.

- It provides public interfaces such as attack(coordinate), allShipsSunk(), getBoard() for other layers to call.

- The Model does not contain any interface-related code at all and focuses on handling game data and business logic.

View

- View is responsible for presenting the current game state to the user in a graphical interface.

- The View class implements the Observer interface, which can listen to the changes of the Model and refresh the interface automatically.

- In the GUI version, View uses Java Swing to draw a 10×10 grid of buttons representing the game board.

- When receiving update notifications from the Model, the View updates the button color according to the state of the grid (e.g. red for hits, blue for misses) and disables the attacked grid.

- The View only handles the interface presentation logic, leaving the user action events to the Controller.

Controller

- Controller acts as a bridge between View and Model.

- It listens to user actions (e.g. button clicks) and converts them into method calls (e.g. attack) for the Model.

- Controller is responsible for verifying the legitimacy of user inputs and controlling the state changes of the Model.

- At the same time, the Controller also updates the View based on feedback from the game logic, such as displaying “hit”, “miss”, or a hint that the game has been won.

* 1. **CLI Integration with GUI Versions Description**

CLI version: directly manipulate Model class, no need for Controller and View, input and output via command line.

GUI version: follow MVC architecture completely, connect View and Model through Controller, realize high cohesion and low coupling design.

1. **Specification and Assertions**
   1. **Model Layer Class Invariants**

**The board is always a 10x10 2D grid.**

**The length of each Ship object in the ships list is a positive integer and the hits count does not exceed the length.**

**The number of tries is always a non-negative integer.**

**The Model always maintains a compliant, synchronized game state: the board state and fleet hit state are consistent with the interface presentation.**

* 1. **Public Methods Specification**

**Model()**

Description: Initialize the game model, including an empty board and randomly arranged fleets.

Pre-condition: None.

Postconditions:

- The Board is correctly initialized to 10x10 EMPTY.

- The list of ships is randomized to a standard length with no overlap.

**attack(coordinate: String): AttackResult**

Description: Execute an attack based on the coordinates passed in and return the attack result.

Pre-conditions:

- The input coordinate string coordinate is in the correct format, e.g. "A5", "C10".

- The row and column numbers are within the range of the board (A-J, 1-10).

Postconditions:

- Perform an attack on the target grid, update CellStatus.

- If a partial hit on a ship occurs, update the hit count of the corresponding Ship.

- If it results in a full hit on a ship, the ship is ruled sunk.

- Updates the tries count.

- Notify all Viewers to refresh the interface.

**loadConfiguration(file: String): boolean**

Description: load ship configuration from an external file (reserved feature, currently not implemented).

Pre-condition: the file exists and is in the correct format.

Postconditions:

-If the load succeeds, replace the current fleet layout.

-If it fails, returning false does not affect the current game state.

**allShipsSunk(): boolean**

Description: detect if all ships have sunk.

Precondition: none.

Postconditions: if isSunk() is true for all ships, return true; otherwise, return false.

* 1. **Internal Assertions**

Use assert length > 0 in the Ship constructor to ensure that the length of the ship must be a positive integer.

Assert hits <= length in Ship.hit() method to prevent over-accumulation of hits.

Assert in Model.attack() method:

- After parsing the number of rows and columns, make sure the coordinates are legal (0 <= row < 10, 0 <= col < 10).

- Call setChanged() and notifyObservers() after updating the state to make sure the interface is refreshed synchronously.

In randomlyPlaceShips(), loop to check if it is safe to place the ships, don't allow the ships to overlap or go beyond the board.

1. **Testing Summary**

To verify the correctness of the core functionality of the Battleships project, I used JUnit 4 to unit test the Model class, focusing on the verification of the main parts of the game logic, including the attack hit detection, the number of attempts to count the number of attempts and the victory conditions.

**Specific test content**

Test 1: Attack Return Value Verification (testAttackMissOrHit)

Purpose: Verify that the AttackResult object returned by the Model.attack(String coordinate) method is not null and the hit status is reasonable.

Result: The test passes, the attack() method returns valid results in different cases, as expected.

Test 2: Tries Increment (testTriesIncrement)

Purpose: Verify that the number of tries attempted in the game is automatically incremented after each attack.

Method: After attacking a legal coordinate, compare the number of tries before and after the attack.

Result: The test passes, each attack correctly increments the number of tries.

Test 3: Victory Condition Eventually (testVictoryConditionEventually)

Purpose: Verify that the allShipsSunk() method correctly recognizes a game victory by attacking all ship positions on the board.

Result: The test passes, and the model correctly returns the game over flag after all ships have been sunk.

**Test Methods and Tools**

Test Framework: JUnit 4.13.2

Test Environment: IntelliJ IDEA

Organization: Each test method is independent, use @Before annotation to initialize the new Model instance before testing to ensure that the tests do not interfere with each other.

Assertion methods: assertNotNull, assertTrue, assertEquals.

**Test Results Summary**

All unit tests passed successfully.

The core functions (attack processing, victory judgment, counting the number of attempts) are all verified without errors.

The program performs stably under normal input without abnormal crash.

The Model class has good stability and functional correctness, and meets the basic testing standards of software engineering.  
图形用户界面, 文本, 应用程序, 电子邮件

AI 生成的内容可能不正确。

1. **Project Summary and Reflection**

During the development of Battleships, I practiced the principles of object-oriented design and the MVC architecture pattern.

First, in the design phase, according to the requirements of the project, I reasonably divided the three main parts into Model, View, and Controller, with Model handling the game logic independently, View being responsible for the interface display, and Controller serving as a bridge connecting the user operation and the model, thus realizing a clear separation of responsibilities.

The main challenges I encountered during the development process include:

- How to connect Model and View with Observer/Observable pattern under MVC layering principle;

- How to ensure that CLI and GUI share the same model, with concise interface and no need to rewrite the logic.

-At first, I didn't consider that each ship should indicate that it has been sunk when it sinks, which led to the modification of the class diagram and code.

Through this project, I was exposed to Java Swing GUI development, gained an in-depth understanding of the practical use of unit testing (JUnit), and improved my experience in object-oriented programming thinking and design. Overall, this project is a practical exercise with great gains and comprehensive progress.

1. **Appendix - Key Source Code**

**Model.java**

package model;  
  
import java.util.\*;  
  
*/\*\*  
 \* Main game model handling the game state.  
 \*/*public class Model extends Observable {  
 private Board board;  
 private List<Ship> ships;  
 private int tries;  
  
 */\*\*  
 \* Constructor for Model.  
 \*/* public Model() {  
 board = new Board();  
 ships = new ArrayList<>();  
 tries = 0;  
 randomlyPlaceShips();  
 }  
  
 private void randomlyPlaceShips() {  
 int[] shipLengths = {5, 4, 3, 2, 2};  
 Random random = new Random();  
  
 for (int length : shipLengths) {  
 boolean placed = false;  
 while (!placed) {  
 int row = random.nextInt(board.getSize());  
 int col = random.nextInt(board.getSize());  
 boolean horizontal = random.nextBoolean();  
  
 Ship ship = new Ship();  
 if (board.placeShip(ship, row, col, length, horizontal)) {  
 ships.add(ship);  
 placed = true;  
 }  
 }  
 }  
 }  
  
 */\*\*  
 \* Handles an attack based on input coordinate.  
 \*/* public AttackResult attack(String coordinate) {  
 int row = coordinate.charAt(0) - 'A';  
 int col = Integer.*parseInt*(coordinate.substring(1)) - 1;  
  
 if (!isValidCoordinate(row, col)) {  
 throw new IllegalArgumentException("Invalid coordinate: " + coordinate);  
 }  
  
 tries++;  
 Ship shipHit = board.attack(row, col);  
  
 boolean hit = (shipHit != null);  
 boolean sunk = false;  
  
 if (hit) {  
 shipHit.registerHit(new Position(row, col));  
 if (shipHit.isSunk()) {  
 sunk = true;  
 }  
 }  
  
 setChanged();  
 notifyObservers();  
 return new AttackResult(hit, sunk);  
 }  
  
 private boolean isValidCoordinate(int row, int col) {  
 return row >= 0 && row < board.getSize() && col >= 0 && col < board.getSize();  
 }  
  
 public boolean allShipsSunk() {  
 for (Ship ship : ships) {  
 if (!ship.isSunk()) {  
 return false;  
 }  
 }  
 return true;  
 }  
  
 public int getTries() {  
 return tries;  
 }  
  
 public Board getBoard() {  
 return board;  
 }  
  
 public boolean loadConfiguration(String filename) {  
 return false; *// TODO* }  
}

**Controller.java**

package controller;  
  
import model.\*;  
import view.View;  
  
*/\*\*  
 \* Controller for the Battleships GUI version.  
 \* Handles user actions and updates the Model and View accordingly.  
 \*/*public class Controller {  
 private Model model;  
 private View view;  
  
 */\*\*  
 \* Constructor for Controller.  
 \* @param model the game model  
 \* @param view the game view  
 \*/* public Controller(Model model, View view) {  
 this.model = model;  
 this.view = view;  
 }  
  
 */\*\*  
 \* Processes a user's guess based on clicked row and column.  
 \*/* public void processGuess(int row, int col) {  
 String coordinate = convertToCoordinate(row, col);  
 try {  
 AttackResult result = model.attack(coordinate);  
 if (result.isHit()) {  
 view.showMessage("Hit!");  
 if (result.isSunk()) {  
 view.showMessage("You sunk a ship!");  
 }  
 } else {  
 view.showMessage("Miss!");  
 }  
  
 if (model.allShipsSunk()) {  
 view.showMessage("Congratulations! You sank all ships in " + model.getTries() + " tries!");  
 view.disableBoard();  
 }  
 } catch (IllegalArgumentException e) {  
 view.showMessage("Invalid coordinate!");  
 }  
 }  
  
 */\*\*  
 \* Converts row and column into Battleships coordinate format (e.g., A5).  
 \*/* private String convertToCoordinate(int row, int col) {  
 char rowChar = (char) ('A' + row);  
 return rowChar + String.*valueOf*(col + 1);  
 }  
  
 public void loadFile(String filename) {  
 boolean success = model.loadConfiguration(filename);  
 if (!success) {  
 view.showMessage("Failed to load configuration file.");  
 } else {  
 view.showMessage("Configuration loaded successfully.");  
 }  
 }  
  
 public void updateView() {  
 view.refresh();  
 }  
}

**View.java**

package view;  
  
import model.\*;  
import controller.Controller;  
  
import javax.swing.\*;  
import java.awt.\*;  
import java.awt.event.\*;  
import java.util.Observable;  
import java.util.Observer;  
  
*/\*\*  
 \* Graphical user interface (GUI) view for Battleships.  
 \*/*public class View extends JFrame implements Observer {  
 private Model model;  
 private Controller controller;  
 private JButton[][] gridButtons;  
 private JLabel attemptsLabel;  
  
 */\*\*  
 \* Constructor for View.  
 \* @param model the game model  
 \* @param controller the game controller (can be set later)  
 \*/* public View(Model model, Controller controller) {  
 this.model = model;  
 this.controller = controller;  
  
 model.addObserver(this);  
  
 initializeGUI();  
 }  
  
 */\*\*  
 \* Initializes the GUI components.  
 \*/* private void initializeGUI() {  
 setTitle("Battleships Game");  
 setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);  
 setLayout(new BorderLayout());  
  
 JPanel boardPanel = new JPanel(new GridLayout(model.getBoard().getSize(), model.getBoard().getSize()));  
 gridButtons = new JButton[model.getBoard().getSize()][model.getBoard().getSize()];  
  
 for (int row = 0; row < model.getBoard().getSize(); row++) {  
 for (int col = 0; col < model.getBoard().getSize(); col++) {  
 JButton button = new JButton();  
 final int r = row;  
 final int c = col;  
 button.addActionListener(new ActionListener() {  
 public void actionPerformed(ActionEvent e) {  
 if (controller != null) {  
 controller.processGuess(r, c);  
 }  
 }  
 });  
 gridButtons[row][col] = button;  
 boardPanel.add(button);  
 }  
 }  
  
 attemptsLabel = new JLabel("Attempts: 0");  
 add(attemptsLabel, BorderLayout.NORTH);  
 add(boardPanel, BorderLayout.CENTER);  
  
 setSize(600, 600);  
 setLocationRelativeTo(null); *// Center the window* setVisible(true);  
 }  
  
 */\*\*  
 \* Update method called when the Model changes.  
 \*/* @Override  
 public void update(Observable o, Object arg) {  
 refresh();  
 }  
  
 */\*\*  
 \* Refreshes the board display according to the current model state.  
 \*/* public void refresh() {  
 Board board = model.getBoard();  
  
 for (int row = 0; row < board.getSize(); row++) {  
 for (int col = 0; col < board.getSize(); col++) {  
 CellStatus status = board.getCellStatus(row, col);  
 JButton button = gridButtons[row][col];  
  
 switch (status) {  
 case HIT:  
 button.setBackground(Color.RED);  
 button.setEnabled(false);  
 break;  
 case MISS:  
 button.setBackground(Color.BLUE);  
 button.setEnabled(false);  
 break;  
 default:  
 *// Do nothing if still empty* break;  
 }  
 }  
 }  
 attemptsLabel.setText("Attempts: " + model.getTries());  
 }  
  
 */\*\*  
 \* Displays a popup message to the user.  
 \*/* public void showMessage(String message) {  
 JOptionPane.showMessageDialog(this, message);  
 }  
  
 */\*\*  
 \* Disables all buttons on the board.  
 \*/* public void disableBoard() {  
 for (int row = 0; row < gridButtons.length; row++) {  
 for (int col = 0; col < gridButtons[row].length; col++) {  
 gridButtons[row][col].setEnabled(false);  
 }  
 }  
 }  
  
 */\*\*  
 \* Allows setting the controller after View initialization.  
 \*/* public void setController(Controller controller) {  
 this.controller = controller;  
 }  
}

**CLI.java**

package cli;  
  
import model.\*;  
import java.util.Scanner;  
  
*/\*\*  
 \* Command-line interface for the Battleships game.  
 \*/*public class CLI {  
 private Model model;  
 private Scanner scanner;  
  
 */\*\*  
 \* Constructor for CLI.  
 \* @param model the game model  
 \*/* public CLI(Model model) {  
 this.model = model;  
 this.scanner = new Scanner(System.*in*);  
 }  
  
 */\*\*  
 \* Runs the CLI version of the game.  
 \*/* public void run() {  
 System.*out*.println("=== Welcome to Battleships CLI ===");  
 displayGrid();  
  
 while (!model.allShipsSunk()) {  
 System.*out*.print("Enter coordinate (e.g., A5): ");  
 String input = getUserInput();  
  
 try {  
 AttackResult result = model.attack(input);  
 if (result.isHit()) {  
 System.*out*.println("Hit!");  
 if (result.isSunk()) {  
 System.*out*.println("You sunk a ship!");  
 }  
 } else {  
 System.*out*.println("Miss!");  
 }  
 } catch (IllegalArgumentException e) {  
 System.*out*.println("Invalid input. Try again.");  
 }  
  
 displayGrid();  
 }  
  
 System.*out*.println("Congratulations! You sank all the ships!");  
 System.*out*.println("Total tries: " + model.getTries());  
 }  
  
 */\*\*  
 \* Displays the current grid.  
 \*/* public void displayGrid() {  
 Board board = model.getBoard();  
 System.*out*.println(" 1 2 3 4 5 6 7 8 9 10");  
  
 for (int row = 0; row < board.getSize(); row++) {  
 char rowLabel = (char) ('A' + row);  
 System.*out*.print(rowLabel + " ");  
 for (int col = 0; col < board.getSize(); col++) {  
 CellStatus status = board.getCellStatus(row, col);  
 switch (status) {  
 case *HIT*:  
 System.*out*.print("H ");  
 break;  
 case *MISS*:  
 System.*out*.print("M ");  
 break;  
 default:  
 System.*out*.print(". ");  
 break;  
 }  
 }  
 System.*out*.println();  
 }  
 }  
  
 */\*\*  
 \* Gets user input.  
 \*/* public String getUserInput() {  
 return scanner.nextLine().toUpperCase().trim();  
 }  
}