Jungle Game - Developer Guide

Table of Contents

Overview

The Jungle Game is a Python implementation of the traditional Chinese board game "Dou Shou Qi" (Fighting Animals). The project follows the Model-View-Controller (MVC) architectural pattern and emphasizes clean code, comprehensive testing, and maintainability.

Key Features

- **MVC Architecture**: Clear separation between game logic, user interface, and input handling
- **Type Hints**: Comprehensive type annotations throughout the codebase
- **Comprehensive Documentation**: Detailed docstrings for all classes and methods
- **Logging System**: Built-in logging for debugging and troubleshooting
- **File Persistence**: Save/load game states and replay game records
- **Undo Functionality**: Support for undoing up to 3 moves
- **Extensible Design**: Easy to add new pieces, rules, or features

Architecture

MVC Pattern

The application follows the Model-View-Controller pattern:

```
┌─────────────────────────────────────────────────────────┐
│                       Controller                          │
│  ┌───────────────────────────────────────────────────┐   │
│  │ GameController                                     │   │
│  │ - Main game loop                                   │   │
│  │ - Command processing                               │   │
│  │ - Coordinates Model and View                       │   │
│  └───────────────────────────────────────────────────┘   │
│  ┌───────────────────────────────────────────────────┐   │
│  │ CommandParser, FileManager, NameManager            │   │
│  └───────────────────────────────────────────────────┘   │
└─────────────────────────────────────────────────────────┘
                          │
          ┌─────────────────────────────────┐
```

```
         │                                   │
         ▼                                   ▼
┌─────────────────┐              ┌─────────────────┐
│      Model      │              │       View      │
│  ┌───────────┐  │              │  ┌───────────┐  │
│  │   Game    │  │              │  │ GameView  │  │
│  │   Board   │  │              │  │BoardRender│  │
│  │   Piece   │  │              │  │MessageDisp│  │
│  │   Player  │  │              │  └───────────┘  │
│  └───────────┘  │              └─────────────────┘
│                 │
└─────────────────┘
```

Component Responsibilities

Model (model/)

- Game logic and rules enforcement
- Board state management
- Piece movement and capture logic
- Game state persistence

View (view/)

- Board rendering (ASCII art)
- Game state display
- Message formatting
- User feedback

Controller (controller/)

- User input processing
- Command parsing and routing
- File operations
- Game flow management

Project Structure

```
jungle-game/
├── model/                    # Game logic and data models
│   ├── __init__.py
│   ├── board.py            # Board representation and terrain
│   ├── enums.py            # Enumeration types
│   ├── exceptions.py       # Custom exception classes
│   ├── game.py             # Main game logic and rules
│   ├── game_state.py       # State snapshots for undo
│   ├── move.py             # Move tracking and serialization
│   ├── piece.py            # Piece hierarchy and movement
│   ├── player.py           # Player representation
│   └── position.py         # Position on the board
│
├── view/                     # User interface components
│   ├── __init__.py
│   ├── board_renderer.py   # ASCII board rendering
│   ├── game_view.py        # Game state display
│   └── message_display.py # Message formatting
│
├── controller/               # Input handling and coordination
│   ├── __init__.py
│   ├── command_parser.py   # Command parsing and validation
│   ├── file_manager.py    # File save/load operations
│   ├── game_controller.py # Main game loop
│   └── name_manager.py    # Player name management
│
├── utils/                  # Utility modules
│   ├── __init__.py
│   └── logger.py          # Logging configuration
│
├── tests/                  # Test suite
│   ├── test_*.py          # Unit and integration tests
│   └── ...
│
├── logs/                   # Log files (created at runtime)
│
├── main.py                 # Application entry point
├── README.md               # User documentation
├── DEVELOPER_GUIDE.md      # This file
└── requirements.txt        # Python dependencies (if any)
```

Core Components

Model Components

Game (model/game.py)

The central game state manager that enforces all game rules.

Key Methods:

- `make_move(from_pos, to_pos)`: Execute a move with validation
- `undo_move()`: Revert the last move
- `is_game_over()`: Check if game has ended
- `get_winner()`: Get the winning player

Key Attributes:

- `board`: The game board
- `players`: List of players
- `move_history`: All moves made
- `game_states`: Saved states for undo

Board (model/board.py)

Represents the 7x9 game board with terrain management.

Key Methods:

- `get_piece(pos)`: Get piece at position
- `set_piece(pos, piece)`: Place or remove piece
- `is_water(pos)`, `is_den(pos)`, `is_trap(pos)`: Terrain queries

Board Layout:

```
  0 1 2 3 4 5 6
0 E W P # P W E  (Blue)
1 . C * . * C .
```

```
2 T . . . . . L
3 . ~ ~ . ~ ~ .
4 . ~ ~ . ~ ~ .
5 . ~ ~ . ~ ~ .
6 L . . . . . T
7 . D * . * D .
8 R P W # W P E  (Red)

Legend:
# = Den
* = Trap
~ = Water
. = Land
```

Piece Hierarchy (model/piece.py)

Abstract base class `Piece` with concrete implementations:

- **Rat (Rank 1)**: Can move in water, captures elephants
- **Cat (Rank 2)**: Standard land piece
- **Dog (Rank 3)**: Standard land piece
- **Wolf (Rank 4)**: Standard land piece
- **Leopard (Rank 5)**: Standard land piece
- **Tiger (Rank 6)**: Can jump over rivers
- **Lion (Rank 7)**: Can jump over rivers
- **Elephant (Rank 8)**: Highest rank, cannot capture rats

Key Methods:

- `can_move_to(board, target)`: Check if move is valid
- `can_capture(target_piece, board)`: Check if capture is allowed
- `get_valid_moves(board)`: Get all valid moves

Controller Components

GameController (controller/game_controller.py)

Main game loop and command processor.

Key Methods:

- `run_game_loop()`: Main game loop
- `process_command(command)`: Route commands to handlers
- `_handle_move_command(command)`: Process move commands
- `_handle_save_command(parts)`: Save game state
- `_handle_load_command(parts)`: Load game state

CommandParser (controller/command_parser.py)

Parses user input into game commands.

Supported Formats:

- Chess notation: `a0 b1`, `A0 B1`
- Coordinate notation: `0,0 1,0`, `(0,0) (1,0)`

- Verbose: `move from a0 to b1`

FileManager (controller/file_manager.py)

Handles file operations for game persistence.

File Formats:

- `.jungle`: JSON format for complete game state
- `.record`: Text format for move history

View Components

GameView (view/game_view.py)

Coordinates display of complete game state.

Key Methods:

- `display_game_state(game)`: Show current game state
- `display_game_over(game)`: Show game over message
- `display_welcome_message()`: Show welcome screen

BoardRenderer (view/board_renderer.py)

Renders the board as ASCII art.

Piece Symbols:

- Uppercase = Blue player
- Lowercase = Red player
- R/r=Rat, C/c=Cat, D/d=Dog, W/w=Wolf, P/p=Leopard, T/t=Tiger, L/l=Lion, E/e=Elephant

Development Setup

Prerequisites

- Python 3.8 or higher
- No external dependencies required (uses only Python standard library)
- Windows recommended
- IDE:
- Recommended: Visual Studio Code with the Python extension (by Microsoft)
- Alternative: PyCharm Community Edition

Installation

1. Download Zip File and Extract it:
2. Run the game in root folder:

```
python main.py
```

3. Run tests:

```
python -m pytest tests/
# or
python -m unittest discover tests/
```

Development Mode

Enable debug logging for development:

```
python main.py --debug
```

This will:

- Enable DEBUG level logging
- Create detailed log files in `logs/` directory
- Include file and line number information in logs

Testing

Test Structure

Tests are organized by component:

```
tests/
├── test_board.py           # Board functionality
├── test_piece.py           # Piece movement and capture
├── test_game.py            # Game logic and rules
├── test_game_state.py      # State management
├── test_move.py            # Move tracking
├── test_player.py          # Player management
├── test_command_parser.py # Input parsing
├── test_file_manager.py    # File operations
├── test_game_controller.py# Controller logic
├── test_integration.py     # End-to-end tests
└── ...
```

Running Tests

Run all tests:

```
python -m pytest tests/ -v
```

Run specific test file:

```
python -m pytest tests/test_game.py -v
```

Run tests with coverage:

```
python -m pytest tests/ --cov=model --cov=controller --cov=view
```

Writing Tests

Example test structure:

```
import unittest
from model.game import Game
from model.position import Position
```

```
class TestGameMoves(unittest.TestCase):
    def setUp(self):
        """Set up test fixtures."""
        self.game = Game("Player 1", "Player 2")

    def test_valid_move(self):
        """Test that a valid move is executed successfully."""
        from_pos = Position(8, 0)  # Red rat
        to_pos = Position(7, 0)

        result = self.game.make_move(from_pos, to_pos)

        self.assertTrue(result.success)
        self.assertIsNone(self.game.board.get_piece(from_pos))
        self.assertIsNotNone(self.game.board.get_piece(to_pos))
```

Logging

Logging System

The game uses a centralized logging system (`utils/logger.py`) that provides:

- **File Logging**: Logs saved to `logs/jungle_game_YYYYMMDD_HHMMSS.log`
- **Log Levels**: DEBUG, INFO, WARNING, ERROR, CRITICAL
- **Automatic Cleanup**: Old logs deleted after 7 days
- **Detailed Format**: Includes timestamp, module, level, and message

Using Logging in Code

```
from utils.logger import get_logger

# Get logger for current module
logger = get_logger(__name__)

# Log at different levels
logger.debug("Detailed debugging information")
logger.info("General information")
logger.warning("Warning message")
logger.error("Error occurred", exc_info=True)  # Include stack trace
logger.critical("Critical error")
```

Log Levels

- **DEBUG**: Detailed information for diagnosing problems
- **INFO**: General informational messages
- **WARNING**: Warning messages for potentially harmful situations
- **ERROR**: Error messages for serious problems
- **CRITICAL**: Critical errors that may cause the program to abort

Viewing Logs

Logs are stored in the `logs/` directory:

```
# View latest log
tail -f logs/jungle_game_*.log
```

```
# Search logs
grep "ERROR" logs/jungle_game_*.log
```

Extending the Game

Adding a New Piece Type

1. Create a new class in `model/piece.py`:

```
class NewPiece(Piece):
    """
    NewPiece - Rank X.
    Description of special abilities.
    """

    def __init__(self, owner: 'Player', position: Position):
        super().__init__(rank=X, owner=owner, position=position)

    def can_move_to(self, board: 'Board', target: Position) -> bool:
        # Implement movement logic
        pass

    def can_capture(self, target_piece: 'Piece', board: 'Board') -> bool:
        # Implement capture logic
        pass

    def get_valid_moves(self, board: 'Board') -> List[Position]:
        # Implement valid moves logic
        pass
```

2. Add symbol to `BoardRenderer.PIECE_SYMBOLS` in `view/board_renderer.py`

3. Update `Game._initialize_pieces()` to place the new piece

4. Write tests in `tests/test_piece.py`

Adding a New Command

1. Add command handler in `GameController`:

```
def _handle_new_command(self, parts: list) -> bool:
    """
    Handle a new command.

    Args:
        parts: Command parts

    Returns:
        True if successful, False otherwise
    """
    logger.info(f"Executing new command: {parts}")
    # Implement command logic
    return True
```

2. Add routing in `process_command()`:

```
elif cmd in ['new', 'n']:
    return self._handle_new_command(parts)
```

3. Update help message in `_handle_help_command()`

4. Write tests in `tests/test_game_controller.py`

Adding New Terrain Types

1. Add enum value in `model/enums.py`:

```
class TerrainType(Enum):
    LAND = "land"
    WATER = "water"
    DEN = "den"
    TRAP = "trap"
    NEW_TERRAIN = "new_terrain"  # Add this
```

2. Update `Board._initialize_terrain()` to place new terrain

3. Add rendering in `BoardRenderer._render_cell()`

4. Update piece movement logic to handle new terrain

Code Style Guidelines

Python Style

- Follow PEP 8 style guide
- Use type hints for all function parameters and return values
- Maximum line length: 100 characters
- Use docstrings for all classes and methods

Docstring Format

```
def method_name(param1: Type1, param2: Type2) -> ReturnType:
    """
    Brief description of what the method does.

    More detailed description if needed. Explain the purpose,
    behavior, and any important details.

    Args:
        param1: Description of param1
        param2: Description of param2

    Returns:
        Description of return value

    Raises:
        ExceptionType: When this exception is raised
    """
    pass
```

Naming Conventions

- **Classes**: PascalCase (e.g., `GameController`, `BoardRenderer`)
- **Functions/Methods**: snake_case (e.g., `make_move`, `get_valid_moves`)
- **Constants**: UPPER_SNAKE_CASE (e.g., `MAX_UNDO_MOVES`, `BOARD_WIDTH`)

- **Private members**: Prefix with underscore (e.g., `_grid`, `_validate_move`)

Import Organization

```
# Standard library imports
import sys
import json
from typing import Optional, List

# Local imports
from model.game import Game
from model.position import Position
from utils.logger import get_logger
```

Common Development Tasks

Running the Game

```
# Start new game
python main.py

# Load saved game
python main.py --load mygame.jungle

# Replay game record
python main.py --replay mygame.record

# Enable debug logging
python main.py --debug

# Disable file logging
python main.py --no-log-file
```

Debugging

1. Enable debug logging:

```
python main.py --debug
```

2. Check log files in `logs/` directory

3. Use Python debugger:

```
import pdb; pdb.set_trace()
```

Adding Tests

1. Create test file in `tests/` directory
2. Import necessary modules
3. Create test class inheriting from `unittest.TestCase`
4. Write test methods (must start with `test_`)
5. Run tests to verify

Profiling Performance

```
import cProfile
import pstats

# Profile the game
cProfile.run('main()', 'profile_stats')

# View stats
stats = pstats.Stats('profile_stats')
stats.sort_stats('cumulative')
stats.print_stats(20)
```

Generating Documentation

```
# Generate HTML documentation from docstrings
python -m pydoc -w model controller view utils

# Or use Sphinx for more comprehensive docs
sphinx-quickstart
sphinx-apidoc -o docs/source .
make html
```

Troubleshooting

Common Issues

Issue: Game crashes on startup

- **Solution**: Check Python version (3.8+), verify all files are present

Issue: Moves not working as expected

- **Solution**: Enable debug logging, check piece movement logic in logs

Issue: File save/load errors

- **Solution**: Check file permissions, verify file format, check logs

Issue: Tests failing

- **Solution**: Run individual tests to isolate issue, check test fixtures

Getting Help

1. Check log files in `logs/` directory
2. Enable debug logging with `--debug` flag
3. Review relevant test files for examples
4. Check docstrings for API documentation