Tetris Game

Design & Architecture Document

Version: 1.0

Last Updated: Current State

Target Audience: Engineers & Developers

Table of Contents

1. System Overview
2. Architecture Overview
3. Core Components
4. Design Patterns
5. Data Structures
6. State Management
7. Rendering Pipeline
8. Input System
9. Scoring & Progression
10. Testing Strategy
11. Dependencies

System Overview

Project Purpose

A fully functional Tetris game implementation demonstrating:

- Object-oriented design principles
- Clean architecture with separation of concerns
- Comprehensive testing strategies
- Team collaboration practices

Technology Stack

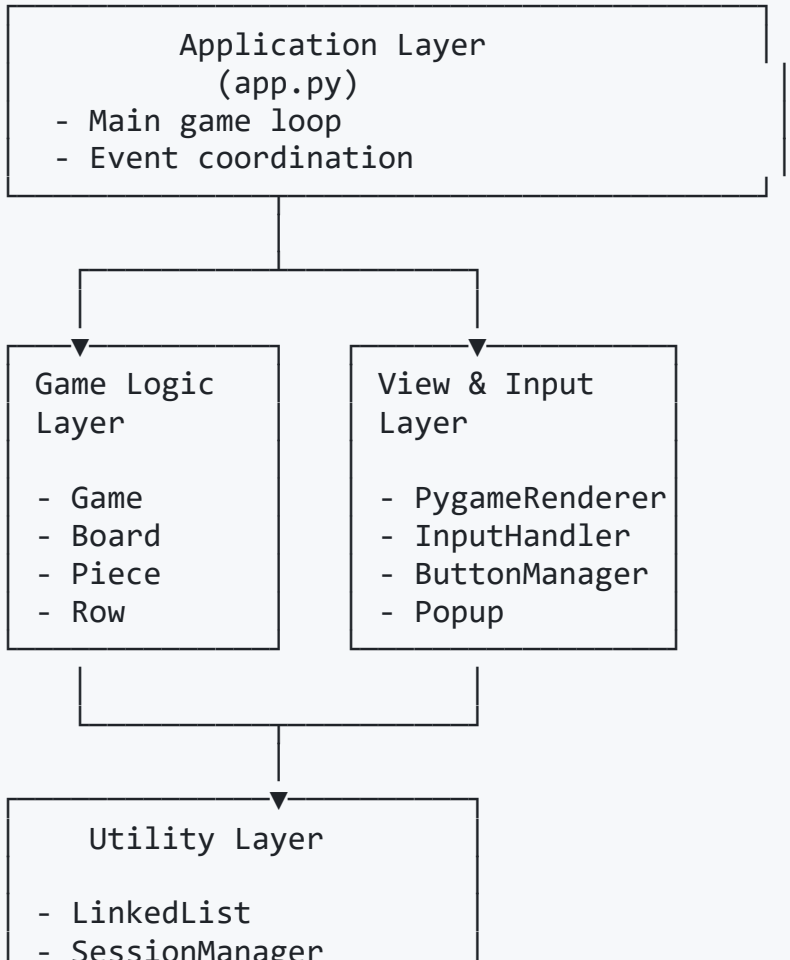
- **Language:** Python 3.x
- **Graphics Library:** Pygame
- **Testing Framework:** pytest
- **Architecture:** Layered architecture with intent-based communication

Key Metrics

- **Source Code:** ~1,200 lines
- **Test Code:** ~5,677 lines
- **Test Coverage:** 411 test cases
- **Modules:** 6 major subsystems

Architecture Overview

High-Level Architecture



Architecture Principles

Separation of Concerns

Three Distinct Layers:

1. **Game Logic Layer** (`src/game/`)
    - Pure game mechanics
    - No rendering dependencies
    - Testable in isolation
2. **View & Input Layer** (`src/view/` , `src/ui/`)
    - Rendering and presentation
    - Input event handling
    - UI component management

3. **Utility Layer** (`src/utils/`)
    - Reusable data structures
    - Session management
    - Helper functions

Intent-Based Communication

Commands flow as string intents:

- `"LEFT"`, `"RIGHT"`, `"ROTATE"`, `"DROP"`
- `"PAUSE"`, `"RESUME"`, `"START"`, `"EXIT"`
- Decouples input sources from game logic
- Enables easy testing and extension

Core Components - Game Logic

Game Class (`src/game/game.py`)

Responsibility: Game state orchestration and rule enforcement

Key Methods:

- `apply(intents)` - Process player commands
- `update()` - Gravity and game progression
- `start_new_game()` - Initialize new game session
- `_freeze_piece()` - Lock piece and handle line clearing

State Management:

- `_state` : START_SCREEN | PLAYING | GAME_OVER
- `paused` : Boolean pause flag
- `level` , `lines_cleared` , `gravity_delay` : Progression tracking

Dependencies:

- `Board` (injected)
- `spawn_piece` function (injected)
- `SessionManager` (injected)

Core Components - Board

Board Class (`src/game/board.py`)

Responsibility: Grid management and piece placement

Key Features:

- Bitmask-based row storage via `LinkedList`
- Collision detection (`will_piece_collide`)
- Line clearing (`clear_full_lines`)
- Piece movement (`go_down`, `go_side`, `rotate`)
- Ghost piece calculation (`get_landing_y`, `get_ghost_cells`)

Board Class - Data Structure

Data Structure:

```
self._rows: LinkedList[Row]  # Linked list of Row objects
self.__height: int           # Board height (20 rows)
self.__width: int            # Board width (10 columns)
```

Factory Pattern:

- Accepts `row_factory` function for dependency injection
- Enables testing with mock rows

Core Components - Piece & Row

Piece Class (`src/game/piece.py`)

Responsibility: Falling piece representation

Attributes:

- `x`, `y`: Grid coordinates
- `type`: Shape index (0-6)
- `rotation`: Rotation state (0-3)
- `color`: Color index
- `cells`: List of occupied board cells

Initialization:

- Random shape and color selection

Row Class (`src/game/row.py`)

Responsibility: Single row representation using bitmasks

Key Features:

- Bitmask (`__bits`) for occupancy tracking
- Color dictionary (`__colors`) for cell colors
- O(1) full-row detection (`is_full()`)

Bitmask Operations:

```
def set_bit(self, col, color):
    self.__bits |= (1 << col)      # Set bit
    self.__colors[col] = color      # Store color

def get_bit(self, col) -> bool:
    return bool(self.__bits & (1 << col))  # Check bit
```


Memory Efficiency:

Core Components - View Layer

PygameRenderer (`src/view/pygame_renderer.py`)

Responsibility: All visual rendering

Rendering Methods:

- `draw_board()` - Grid and filled cells
- `draw_piece()` - Active falling piece
- `draw_next_piece_preview()` - Next piece display
- `draw_ghost_piece()` - Landing position indicator
- `draw_score()` - Score and high score
- `draw_level_info()` - Level, lines, gravity

PygameRenderer (Continued)

Rendering Methods (Continued):

- `draw_start_screen()` - Start menu
- `draw_game_over_screen()` - Game over menu
- `draw_pause_popup()` - Pause menu

Button Management:

- `button_manager`: Popup buttons
- `hud_button_manager`: In-game HUD buttons

Core Components - Input System

InputHandler (`src/view/input.py`)

Responsibility: Map pygame events to game intents

Key Mapping:

```
pygame.K_UP → "ROTATE"
pygame.K_LEFT → "LEFT"
pygame.K_RIGHT → "RIGHT"
pygame.K_DOWN → "DOWN"
pygame.K_SPACE → "DROP"
pygame.K_p → "PAUSE"
pygame.K_ESCAPE → "QUIT" + "PAUSE"
pygame.K_RETURN → "START"
pygame.K_r → "RESTART"
```

Design:

Design Patterns

1. Factory Pattern

Usage: Row creation and piece spawning

```
# Board accepts row factory function
board = Board(lambda: Row(WIDTH))

# Piece spawning via function injection
def spawn_piece():
    return Piece(START_X, START_Y)
game = Game(board, spawn_piece, session)
```

Benefits:

- Dependency injection
- Testability with mocks
- Flexibility for different implementations

Design Patterns (Continued)

2. Singleton Pattern

Usage: SessionManager for persistent session data

```
class SessionManager:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            cls._instance._high_score = 0
        return cls._instance
```

Purpose:

- Maintain high score across game restarts
- Single source of truth for session data

3. Strategy Pattern

Usage: Intent-based command processing

```
# Different input sources produce same intents
intents = input_handler.get_intents(events)
intents.extend(button_manager.handle_click(pos))
game.apply(intents)  # Game handles strategies uniformly
```

Benefits:

- Decouples input from game logic
- Easy to add new input methods
- Consistent command interface

4. State Machine Pattern

Usage: Game state transitions

```
START_SCREEN
    │ (START intent)
    ▼
PLAYING
    │ (game over condition)
    ▼
GAME_OVER
    │ (RESTART intent)
    └──► START_SCREEN
```

Implementation:

- `Game._state` tracks current state
- State-specific behavior in `apply()` method
- Clear state transitions

Data Structures

Custom LinkedList (`src/utils/linked_list.py`)

Purpose: Optimized for board row operations

Key Operations:

- `insert_top(value)` - O(1) - Add rows at top
- `delete_node(index)` - O(n) - Remove full lines
- `get_node_at(index)` - O(n) - Access by index
- `append(value)` - O(n) - Add to end

Why Custom:

- Efficient top insertions (new empty rows)
- Efficient mid-list deletions (line clearing)

Data Structures (Continued)

Bitmask Row Representation

Structure:

```
class Row:
    __bits: int           # Bitmask (e.g., 0b1111111111 for full row)
    __colors: dict[int, int]  # Column index → color mapping
    _mask: int            # Full row mask (1 << width) - 1
```

Operations:

- `is_full()`: `self.__bits == self._mask` (O(1))
- `set_bit(col)`: `self.__bits |= (1 << col)` (O(1))
- `get_bit(col)`: `bool(self.__bits & (1 << col))` (O(1))

Memory Efficiency:

- 10 columns = 10 bits = minimal memory
- Color storage only for occupied cells

State Management

Game States

Three Primary States:

1. **START_SCREEN**
    - Initial state on launch
    - Shows controls and start/exit buttons
    - Transitions to PLAYING on START intent
2. **PLAYING**
    - Active gameplay
    - Handles piece movement, rotation, gravity
    - Can pause (sub-state)

3. **GAME_OVER**
    - Game ended
    - Shows final score
    - Options: RESTART or QUIT

State Management (Continued)

Pause Sub-State

Behavior:

- Toggle via PAUSE intent (P key or ESC)
- Paused: No gravity, no piece movement
- Resume: RESUME intent or CLICK on popup
- Overlay: Semi-transparent pause popup

Implementation:

```
if self.paused:
    continue  # Skip gameplay intents
# Gravity only updates when not paused
if not self.paused:
    self.gravity_timer += 1
```

Rendering Pipeline

Render Order

```
1. Clear screen (white background)
2. Draw board grid and filled cells
3. Draw ghost piece (if playing, not paused)
4. Draw active piece (if playing)
5. Draw next piece preview (if playing)
6. Draw score and level info (if playing)
7. Draw HUD buttons (pause button if playing)
8. Draw overlays (start screen, game over, pause popup)
9. Flip display buffer
```

Coordinate System:

- Board origin: `(70, 60)` pixels
- Cell size: `20x20` pixels
- Screen size: `600x500` pixels

Rendering Pipeline (Continued)

Ghost Piece Rendering

Purpose: Show landing position of current piece

Implementation:

1. Calculate landing Y: `board.get_landing_y(piece)`
2. Render semi-transparent fill (alpha=50)
3. Draw outline in piece color
4. Only shown when playing and not paused

Visual Style:

- Faded color fill
- Distinct outline
- Helps players plan placement

Scoring & Progression

Scoring System

Two Scoring Mechanisms:

1. **Base Scoring** (`points_for_clear()`)
    - Pure function in `src/utils/score.py`
    - Mapping: 1→100, 2→300, 3→500, 4→800
2. **Level-Based Scoring** (`_add_score()`)
    - Multiplier: `1.0 + (level - 1) * 0.1`
    - Base points: 1→40, 2→100, 3→300, 4→1200
    - Applied after base scoring

Score Updates:

- On line clear events
- High score tracked via SessionManager
- Updated after all scoring calculations

Scoring & Progression (Continued)

Level Progression

Level Calculation:

- Level = `(lines_cleared // 10) + 1`
- Levels increase every 10 lines cleared

Gravity System:

- Base delay: 30 frames
- Speed increase: -3 frames per level
- Minimum delay: 10 frames (cap)
- Formula: `max(10, base_gravity_delay - (level - 1) * 3)`

Progression Flow:

```
Lines Cleared → Level Update → Gravity Recalculation
```

UI Components

Popup System (`src/ui/pop_up.py`)

Flexible Popup Container:

- Title (optional)
- Body lines (text)
- Images (optional)
- Buttons (action, label, color)

Layout:

- Auto-calculated height
- Centered on screen
- Semi-transparent overlay
- Button registration via ButtonManager

Usage:

- Start screen
- Game over screen
- Pause popup

UI Components (Continued)

ButtonManager (`src/ui/button_manager.py`)

Responsibility: Button lifecycle management

Features:

- Add/remove buttons dynamically
- Click detection and action dispatch
- Cursor state management (hand on hover)
- Separate managers for popups and HUD

Button Structure:

```
Button(rect, label, action, color, text_color)
```

Actions:

- "START", "EXIT", "RESTART", "PAUSE", "RESUME"

Testing Strategy

Test Organization

Test Structure:

```
tests/
├── unit/           # 17 test files, core logic
├── integration/    # 11 test files, feature interaction
├── acceptance/    # 5 test files, user scenarios
└── regression/    # 5 test files, cross-sprint validation
```

Total: 411 test cases

Testing Strategy (Continued)

Test Categories

1. **Unit Tests**
    - Individual class methods
    - Pure functions
    - Edge cases and error handling
    - Examples: `test_board.py`, `test_piece.py`, `test_score.py`

2. **Integration Tests**
    - Component interaction
    - Feature workflows
    - Examples: `test_game_board_integration.py`

Test Categories (Continued)

3. **Acceptance Tests**
    - End-to-end user scenarios
    - Feature completeness
    - Examples: `test_complete_game_flow.py`
4. **Regression Tests**
    - Cross-sprint validation
    - Previously fixed bugs
    - Examples: `test_regression_sprint1.py`

Dependencies

External Libraries

Pygame:

- Graphics rendering
- Event handling
- Window management
- Image loading

pytest:

- Test framework
- Fixtures and parametrization
- Test discovery

Internal Dependencies

Module Structure:

```
app.py
├── src/game/ (Game, Board, Piece, Row)
├── src/view/ (PygameRenderer, InputHandler)
├── src/ui/ (ButtonManager, Popup)
├── src/utils/ (LinkedList, SessionManager, score)
└── src/constants/ (dimensions, colors, states)
```

No Circular Dependencies:

- Clear dependency hierarchy
- Game logic independent of rendering

Constants & Configuration

Constants Organization

Main Constants (`src/constants.py`):

- Screen dimensions: `SCREEN_SIZE = (600, 500)`
- Board dimensions: `HEIGHT = 20` , `WIDTH = 10`
- Cell size: `CELL_SIZE = 20`
- Starting position: `START_X = 3` , `START_Y = 0`
- Colors: RGB tuples
- FPS: `60`

Constants Organization (Continued)

Modular Constants (`src/constants/`):

- `game_states.py` : State strings
- `colors.py` : Color definitions
- `game_dimensions.py` : Board dimensions

Shapes (`src/figures.py`):

- 7 piece types (I, Z, S, L, J, T, O)
- Rotation states as grid position tuples

Error Handling

Validation Patterns

Input Validation:

```
def _check_row_index(self, row: int):
    if not (0 <= row < self.height):
        raise IndexError(f"Row index {row} out of bounds")
```

Type Checking:

```
if not isinstance(lines_cleared, int):
    raise TypeError("lines_cleared must be an integer")
```

Validation Patterns (Continued)

State Validation:

```
def validate_integrity(self):
    if self.rows.length() != self.height:
        raise RuntimeError("Row count mismatch")
```

Performance Considerations

Optimizations

1. **Bitmask Operations**
    - O(1) cell checks
    - O(1) full-row detection
    - Minimal memory footprint
2. **LinkedList for Rows**
    - Efficient top insertions
    - Efficient line deletions
    - No array shifting overhead

Optimizations (Continued)

3. **Intent-Based Input**
    - Single pass event processing
    - No redundant state checks
4. **Rendering**
    - Only redraw changed regions (future optimization)
    - Efficient pygame surface operations

Development Workflow

Project Structure

```
ase-420-team-project/
├── app.py                    # Entry point
├── src/                      # Source code
│   ├── game/                # Game logic
│   ├── view/                # Rendering & input
│   ├── ui/                  # UI components
│   ├── utils/               # Utilities
│   └── constants/           # Configuration
├── tests/                   # Test suite
├── docs/                    # Documentation
└── scripts/                 # Utility scripts
```

Build & Run

Execution:

```
python app.py
```

Testing:

```
pytest tests/
python run_tests.py
```

Key Design Decisions

1. Intent-Based Communication

Decision: Use string intents instead of direct method calls

Rationale:

- Decouples input from game logic
- Enables multiple input sources
- Simplifies testing
- Easy to extend with new commands

Key Design Decisions (Continued)

2. Bitmask Row Representation

Decision: Use bitmasks for row occupancy

Rationale:

- Memory efficient (10 bits vs 10 booleans)
- Fast full-row detection (single comparison)
- Color stored separately only when needed
- Scales well for larger boards

3. Custom LinkedList

Decision: Implement custom LinkedList instead of Python list

Rationale:

- Optimized for board operations (top insert, mid delete)
- No array shifting overhead
- Clear ownership and control
- Educational value

Key Design Decisions (Continued)

4. Factory Pattern for Rows

Decision: Inject row factory function into Board

Rationale:

- Enables testing with mock rows
- Allows different row implementations
- Follows dependency inversion principle
- Maintains flexibility

5. Singleton SessionManager

Decision: Use singleton for session data

Rationale:

- High score persists across game restarts
- Single source of truth
- Simple implementation
- No global variables

Summary

Architecture Highlights

✅ Clean Separation of Concerns

- Game logic independent of rendering
- Clear layer boundaries
- Testable components

✅ SOLID Principles

- Single responsibility per class
- Dependency injection
- Open/closed for extension

✅ Comprehensive Testing

- 411 test cases
- Multiple test categories
- High coverage

✅ Maintainable Codebase

- Clear structure
- Good documentation
- Consistent patterns

Contact & Resources

Repository

GitHub: https://github.com/jeffreyperdue/ase-420-team-project

Team

- Anna Dinius
- Cody King
- Owen Newberry

Documentation

- Final Presentation: `docs/final_presentation.marp.md`
- Testing Strategy: `tests/TESTING_STRATEGY.md`

Appendix: Class Diagrams

Core Game Classes

```
Game
├── Board
│   ├── LinkedList[Row]
│   └── Row (bitmask)
├── Piece
└── SessionManager (singleton)

PygameRenderer
├── ButtonManager (popups)
├── ButtonManager (HUD)
└── Popup

InputHandler → intents → Game.apply()
```

Appendix: Data Flow

Game Loop Flow

```
1. Events (pygame)
   ↓
2. InputHandler.get_intents()
   ↓
3. Game.apply(intents)
   ↓
4. Game.update() (gravity)
   ↓
5. PygameRenderer.draw_*()
   ↓
6. pygame.display.flip()
```