Spotlight - Digital Board Game Prototype Design Document

Design Specification

November 7, 2025

Contents

1 Program Overview

Spotlight is a turn-based digital board game prototype built using C++ and SFML (Simple and Fast Multimedia Library). The game features a hexagonal grid board where 2-6 players compete by claiming tiles, managing resources, and playing cards with various effects. The game includes a command-line console interface for game actions and a graphical rendering system for visual feedback.

1.1 Core Gameplay Concepts

- **Hexagonal Board:** Game takes place on a hex-grid with configurable radius
- **Territory Control:** Players claim tiles by building stages on them
- **Resource Management:** Players accumulate and spend various resources
- **Card System:** Cards provide special abilities and effects
- **Turn-Based Play:** Players take sequential turns with day/night cycles
- **Company Affiliation:** Each player represents a company with unique identity

1.2 Technology Stack

- **Language:** C++17
- **Graphics:** SFML 2.5+
- **Data Format:** JSON (nlohmann/json library)
- **Build System:** CMake-compatible

2 Program Organization and Module Structure

The program is organized into distinct modules, each with specific responsibilities. This modular design allows for clear separation of concerns and maintainability.

2.1 Module: StartupMenu

File: `StartupMenu.cpp`, `StartupMenu.hpp`

Responsibility: Manages the initial game configuration and menu system before gameplay begins. Handles persistent storage of player preferences and game setup.

Key Responsibilities:

- Display main menu with options (Start Game, Settings, Exit)
- Load and save game configuration from disk

- Manage settings menu for player count and player/company information
- Validate player count (2-6 players)
- Calculate appropriate board radius based on player count
- Initialize game with configured parameters

Data Structures:

- `GameConfig`: Struct containing player count, player names, company names, and company symbols

Key Functions:

- `StartMenuLoop()`: Main entry point that runs menu loop
- `saveConfig()`: Persists configuration to settings.txt
- `loadConfig()`: Loads configuration from disk or returns defaults
- `settingsMenu()`: Interactive menu for editing game setup
- `getAutoRadius()`: Determines board size based on player count

2.2 Module: Game

File: `Game.cpp`, `Game.hpp`

Responsibility: Central game controller that manages game state, coordinates between modules, and executes game logic. Acts as the main orchestrator for all gameplay systems.

Key Responsibilities:

- Maintain game state (current day, active player, board state)
- Manage player roster and company associations
- Process player commands through command console
- Coordinate turn progression and day cycles
- Handle resource transactions
- Manage card deck operations
- Execute card effects and triggers
- Provide interface between UI and game logic

Key Data Members:

- `Board board`: Hexagonal game board

- `std::vector<Player> players`: All players in game
- `std::vector<Company> companies`: All companies in game
- `std::vector<Deck> decks`: Card decks (draw and discard)
- `int currentDay`: Current day number
- `int currentActivePlayerIndex`: Index of player whose turn it is
- `CommandConsole* console`: Input/output console interface
- `Renderer* renderer`: Graphics rendering system
- `sf::RenderWindow window`: SFML window
- `std::map<std::string, std::function> commandHandlers`: Command registry

Key Functions:

- `setup()`: Initializes game state, randomizes tile ownership, loads decks
- `mainLoop()`: Main game loop handling events and rendering
- `executeCommand()`: Parses and dispatches console commands
- `startNewDay()`: Triggers day-start effects for all players
- `endTurn()`: Advances to next player's turn
- `buildStage()`: Places a stage on a tile for a player
- `drawCardForPlayer()`: Draws cards from deck to player hand
- `giveResourceToPlayer()`: Adds resources to player
- `spendResourceFromPlayer()`: Deducts resources from player
- `playCardForPlayer()`: Moves card from hand to play area
- `validateAndSetPlayerIndex()`: Validates player index or resolves -1 to current player

2.3 Module: Board

File: `Board.cpp`, `Board.hpp`

Responsibility: Manages the hexagonal tile grid using cube coordinate system. Handles tile generation, neighbor calculations, and tile property modifications.

Key Responsibilities:

- Generate hexagonal board with specified radius
- Store and retrieve tiles using cube coordinates

- Calculate neighboring tiles for any coordinate
- Modify tile ownership and color
- Validate tile coordinates

Coordinate System: The board uses cube coordinates (x, y, z) where x + y + z = 0. This system simplifies neighbor calculations and distance measurements on hexagonal grids.

Key Data Members:

- `std::map<CubeCoord, Tile> tiles`: Map of all board tiles
- `int radius`: Board radius (determines size)

Key Functions:

- `generateBoard()`: Creates all tiles within radius
- `getTile()`: Retrieves tile pointer at coordinate (or nullptr)
- `getNeighbors()`: Returns vector of valid neighboring coordinates
- `setTileOwner()`: Assigns company ownership to tile
- `setTileColor()`: Sets tile color with validation

2.4 Module: Tile

File: `include/Tile.hpp`, `src/Tile.cpp`

Responsibility: Represents a single board tile that can be owned by a company and display a color. Each tile may have an owner (a pointer to a Company) and an associated color. Tiles start unowned and "neutral" in color by default.

Key Responsibilities:

- Store tile ownership (Company pointer or nullptr for neutral/unowned)
- Store tile color (string identifier)
- Provide getters/setters for tile properties
- Maintain default state (nullptr owner, "neutral" color)

Key Data Members:

- `Company* owner`: Owning company or nullptr for unowned tiles
- `std::string color`: Color identifier (defaults to "neutral")

Key Functions:

- `Tile()`: Default constructor - initializes with no owner and neutral color

- `getOwner()`: Returns pointer to owning company or nullptr if unowned
- `setOwner(Company* newOwner)`: Sets tile ownership (can be nullptr for unowned)
- `getColor()`: Returns reference to current color string
- `setColor(const std::string& newColor)`: Sets tile color by string identifier
- `printInfo()`: Console output for debugging and visualization

2.5 Module: Player

File: `Player.cpp`, `Player.hpp`

Responsibility: Represents a player in the game. Manages player's resources, cards, and company affiliation.

Key Responsibilities:

- Store player identity (name, company)
- Manage resource inventory (dynamic map of resource types to quantities)
- Manage card collections (held cards and played cards)
- Provide operations for card movement between zones
- Add and remove resources

Key Data Members:

- `std::string name`: Player name
- `Company* company`: Associated company
- `std::map<std::string, int> resources`: Resource inventory
- `std::vector<Card> heldCards`: Cards in hand
- `std::vector<Card> playedCards`: Cards in play

Key Functions:

- `addResource()`: Increases resource amount
- `addHeldCard()`: Adds card to hand
- `playCard()`: Moves card from hand to play area
- `removePlayedCard()`: Removes card from play area
- `removeHeldCard()`: Removes card from hand

2.6 Module: Company

File: `Company.cpp`, `Company.hpp`

Responsibility: Represents a company entity that players control. Stores company identity information.

Key Responsibilities:

- Store company name
- Store company symbol (abbreviation)
- Provide identity information for UI display

Key Data Members:

- `std::string name`: Full company name
- `std::string symbol`: Company abbreviation/symbol

2.7 Module: Card

File: `Card.cpp`, `Card.hpp`

Responsibility: Represents individual game cards with effects. Handles card data loading from JSON and trigger execution.

Key Responsibilities:

- Store card identity (name, description)
- Store trigger-action mappings
- Parse card data from JSON format
- Execute effects when triggers activate
- Modify player state based on card effects

Trigger System: Cards can have multiple triggers (e.g., "onPlay", "onStartOfDay") that execute actions when activated. Actions include resource modifications, card draws, and other game effects.

Key Data Members:

- `std::string name`: Card name
- `std::string description`: Card text
- `std::map<std::string, std::vector<json>> triggers`: Trigger-to-actions mapping

Key Functions:

- `Card(const json& data)`: Constructor that parses JSON
- `executeTrigger()`: Executes all actions for a given trigger

2.8 Module: Deck

File: `Deck.cpp`, `Deck.hpp`

Responsibility: Manages collections of cards. Handles deck loading from JSON files, card drawing, and shuffling operations.

Key Responsibilities:

- Load card definitions from JSON files
- Support card duplication via "copies" field
- Provide draw operations (removes from top)
- Support shuffling
- Move cards between decks
- Track deck size

Key Data Members:

- `std::string name`: Deck identifier
- `std::vector<Card> cards`: Card collection

Key Functions:

- `loadFromJsonFile()`: Loads cards from JSON with error handling
- `drawCard()`: Removes and returns top card
- `addCard()`: Adds card to deck
- `moveCardTo()`: Copies card to another deck
- `shuffle()`: Randomizes card order
- `size()`: Returns card count
- `empty()`: Checks if deck is empty

2.9 Module: Colors

File: `Colors.cpp`, `Colors.hpp`

Responsibility: Defines valid color palette for tiles and provides color validation.

Key Responsibilities:

- Maintain list of valid colors
- Validate color strings
- Provide color mappings to RGB values for rendering

Key Data Members:

- `std::vector<std::string> all`: List of valid color names
- `std::map<std::string, sf::Color> colorMap`: Color name to RGB mapping

2.10 Module: CommandConsole

File: `CommandConsole.cpp`, `CommandConsole.hpp`

Responsibility: Manages text-based command input and output display. Provides command queuing and pagination for long outputs.

Key Responsibilities:

- Accept text input from user
- Queue commands for processing
- Display output messages with pagination
- Manage console UI rendering
- Handle text editing (cursor, backspace)

Key Functions:

- `handleInput()`: Processes keyboard events
- `print()`: Displays single message
- `printPaged()`: Displays messages with pagination support
- `hasCommand()`: Checks if commands are queued
- `nextCommand()`: Retrieves next command from queue
- `clear()`: Clears output display

2.11 Module: Renderer

File: `Renderer.cpp`, `Renderer.hpp`

Responsibility: Handles graphical rendering of the game board, tiles, and UI elements using SFML.

Key Responsibilities:

- Convert cube coordinates to pixel positions
- Render hexagonal tiles with colors and ownership
- Handle camera panning and zooming
- Display game information (day, player turn)
- Process window events

2.12 Module: PathUtils

File: `PathUtils.cpp`, `PathUtils.hpp`

Responsibility: Provides platform-independent path resolution for assets and configuration files.

Key Responsibilities:

- Resolve asset file paths (fonts, JSON files)
- Resolve configuration directory paths
- Handle cross-platform file system differences

3 Program Inputs and Outputs

3.1 Startup Phase Inputs

3.1.1 Main Menu Input

Input: Integer selection (1-3)

Valid Values:

- 1: Start Game
- 2: Settings
- 3: Exit

Expected Output:

- Option 1: Proceeds to game initialization with current configuration
- Option 2: Opens settings menu
- Option 3: Exits program with return code 0

Error Handling:

- Non-numeric input: Display "Invalid choice", clear input stream, re-prompt
- Out-of-range number: Display "Invalid choice", re-prompt
- Stream failure: Clear error flags, discard invalid input, re-prompt

3.1.2 Settings Menu - Player Count

Input: Integer representing number of players
Valid Range: 2-6
Expected Output:

- Valid input: Store player count, proceed to player name entry
- Input ¡ 2: Automatically adjust to 2, proceed
- Input ¿ 6: Automatically adjust to 6, proceed

Error Handling:

- Non-numeric input: Display "Invalid input. Please enter a number.", return to main menu
- Stream failure: Clear error flags, discard input, return to main menu

3.1.3 Settings Menu - Player Information

Input: String values for each player (repeated N times for N players)

- Player name (any string including spaces)
- Company name (any string including spaces)
- Company symbol (any string including spaces)

Expected Output:

- Store all values in GameConfig structure
- Save configuration to disk (settings.txt)
- Display "Settings saved!"
- Return to main menu

Error Handling:

- Unable to create config directory: Log error to stderr, continue (config not saved)
- Unable to write file: Log error to stderr, continue (config not saved)

3.2 Gameplay Phase Inputs

All gameplay inputs are text commands entered through the console interface.

3.2.1 Command: set_color

Syntax: `set_color <x> <y> <z> <color>`

Input Parameters:

- `x`: Integer cube coordinate X
- `y`: Integer cube coordinate Y
- `z`: Integer cube coordinate Z (must satisfy x+y+z=0)
- `color`: String color name from valid palette

Expected Output:

- Success: "Set tile (x,y,z) to [color]", tile visually updates
- Invalid coordinates: "Tile (x,y,z) does not exist."
- Invalid color: "Color [color]is not a valid color." followed by list of valid colors

Error Handling:

- Missing arguments: Display "Usage: set_color ¡x¿ ¡y¿ ¡z¿ ¡color¿", no state change
- Parse failure: Display usage message, no state change
- Null tile pointer: Log error, return early, no crash
- Invalid color: Log warning but still set color (known limitation)

3.2.2 Command: set_owner

Syntax: `set_owner <x> <y> <z> <company_index>`

Input Parameters:

- `x`, `y`, `z`: Integer cube coordinates
- `company_index`: Integer index (0 to N-1 where N is company count)

Expected Output:

- Success: "Set tile (x,y,z) to [CompanyName]: [Symbol]", tile ownership updates

Error Handling:

- Missing arguments: Display usage message
- Invalid tile coordinates: No output (handled in Board module)
- Company index out of range: "Error: Company index [N] is out of range. Max valid index: [M]"
- Negative index: Same as out of range error

3.2.3 Command: build

Syntax: `build <x> <y> <z> <color> [player_index]`

Input Parameters:

- `x`, `y`, `z`: Integer cube coordinates
- `color`: String color name
- `player_index`: Optional integer (defaults to -1 = current player)

Expected Output:

- Success: "[PlayerName] ([CompanyName]) built a [color] stage at [xyz]"
- Tile ownership set to player's company
- Tile color set to specified color

Error Handling:

- Missing required arguments: Display usage message
- Invalid player index: "Error: Player index [N] is out of range. Max valid index: [M]"
- Invalid color: Display list of valid colors
- Invalid coordinates: Error message from setTileOwner and setTileColor

3.2.4 Command: list_players

Syntax: `list_players`

Input Parameters: None

Expected Output:

- If players exist: "Players:" followed by list of " - [PlayerName] ([CompanyName]: [Symbol])"
- If no players: "No players available."

Error Handling: None required (read-only operation)

3.2.5 Command: show_resources

Syntax: `show_resources [player_index]`

Input Parameters:

- `player_index`: Optional integer (defaults to -1 = current player)

Expected Output:

- "Resources for player [PlayerName]:"

- List of " [resource]: [amount]" for each resource
- Empty list if player has no resources

Error Handling:

- Invalid player index: "Error: Player index [N] is out of range..."

3.2.6 Command: give_resource

Syntax: `give_resource <resource> <amount> [player_index]`

Input Parameters:

- `resource`: String resource type (any string, dynamically created)
- `amount`: Integer quantity to add
- `player_index`: Optional integer (defaults to -1 = current player)

Expected Output:

- Success: "Gave [amount] [resource] to [PlayerName]."
- Player's resource map updated (creates key if new)

Error Handling:

- Missing arguments: Display usage message
- Parse failure: Display usage message
- Invalid player index: Error message, no state change

3.2.7 Command: spend_resource

Syntax: `spend_resource <resource> <amount> [player_index]`

Input Parameters:

- `resource`: String resource type
- `amount`: Integer quantity to deduct
- `player_index`: Optional integer (defaults to -1 = current player)

Expected Output:

- Success: "[PlayerName] spent [amount] [resource]."
- Resource quantity decreased

Error Handling:

- Missing arguments: Display usage message

- Invalid player index: Error message, no state change
- Insufficient resources: "Error: Not enough [resource] for [PlayerName].", no state change, returns false
- Resource doesn't exist: Same as insufficient (map returns 0 for missing keys)

3.2.8 Command: show_cards

Syntax: `show_cards [player_index]`

Input Parameters:

- `player_index`: Optional integer (defaults to -1 = current player)

Expected Output:

- "Cards held by [PlayerName]:"
- List of " - [CardName]" for each card
- "(no cards)" if hand is empty
- Output is paginated if list is long

Error Handling:

- Invalid player index: Error message

3.2.9 Command: get_card_count

Syntax: `get_card_count <deck_name>`

Input Parameters:

- `deck_name`: String deck identifier (e.g., "drawDeck", "discardDeck")

Expected Output:

- Success: "[deckName] has [N] card(s)"

Error Handling:

- Missing argument: Display usage message
- Deck not found: "Error: Deck '[name]' not found."

3.2.10 Command: draw_card

Syntax: `draw_card <deck_name> <amount> [player_index]`

Input Parameters:

- `deck_name`: String deck identifier
- `amount`: Integer number of cards to draw
- `player_index`: Optional integer (defaults to -1 = current player)

Expected Output:

- For each drawn card: "[PlayerName] drew a card: [CardName]"
- Cards added to player's hand
- Cards removed from deck

Error Handling:

- Missing arguments: Display usage message
- Invalid player index: Error message, no cards drawn
- Deck not found: Error message, no cards drawn
- Empty deck: "The deck is empty! No more cards to draw.", stops drawing
- Deck becomes empty during draw: Stops at last available card, displays message
- Attempt to draw from empty deck: Error logged to stderr, returns default card

3.2.11 Command: play_card

Syntax: `play_card <card_name> [player_index]`

Input Parameters:

- `card_name`: String card name (supports multi-word names)
- `player_index`: Optional integer (defaults to -1 = current player)

Expected Output:

- Success: "[PlayerName] played card: [CardName]"
- Card moved from hand to play area
- "onPlay" trigger executes (e.g., resource changes logged)

Error Handling:

- Missing card name: Display usage message
- Invalid player index: Error message, no state change
- Card not in hand: "Error: [PlayerName] does not have card '[CardName]' in hand."
- Ambiguous card name ending with number: May interpret last token as player index

3.2.12 Command: remove_played_card

Syntax: `remove_played_card <card_name> [player_index]`

Input Parameters:

- `card_name`: String card name
- `player_index`: Optional integer (defaults to -1 = current player)

Expected Output:

- Success: "Removed played card '[CardName]' from [PlayerName]"
- Card removed from play area

Error Handling:

- Missing card name: Display usage message
- Invalid player index: Error message
- Card not in play: "Error: [PlayerName] does not have card '[CardName]' in play."

3.2.13 Command: remove_held_card

Syntax: `remove_held_card <card_name> [player_index]`

Input Parameters:

- `card_name`: String card name
- `player_index`: Optional integer (defaults to -1 = current player)

Expected Output:

- Success: "Removed card '[CardName]' from [PlayerName]'s hand."
- Card removed from hand

Error Handling:

- Missing card name: Display usage message
- Invalid player index: Error message
- Card not in hand: "Error: [PlayerName] does not have card '[CardName]' in hand."

3.2.14 Command: end_turn

Syntax: `end_turn`
Input Parameters: None
Expected Output:

- "[CurrentPlayerName] ended their turn."
- "[NextPlayerName] starts their turn."
- If last player: "Last player finished turn. Starting day: [N]"
- Day counter increments if applicable
- "onStartOfDay" triggers execute for all played cards

Error Handling: None required (always succeeds)

3.2.15 Command: next

Syntax: `next`
Input Parameters: None
Expected Output:

- If pagination active: Display next page of output
- If no more pages: "No more pages to show."

Error Handling: None required

3.2.16 Command: clear

Syntax: `clear`
Input Parameters: None
Expected Output:

- Console output cleared
- Screen refreshes with empty output area

Error Handling: None required

3.2.17 Command: help

Syntax: `help`
Input Parameters: None
Expected Output:

- Display paginated list of all available commands with syntax
- Each command shown with usage example

Error Handling: None required

3.2.18 Unknown Commands

Input: Any string not matching registered command names
Expected Output:

- "Unknown command: [command]"
- No state change
- Ready for next command

Error Handling: Safe default behavior, no crash

3.3 File System Inputs

3.3.1 Configuration File

File Location: Platform-dependent configuration directory + "settings.txt"
Format: Plain text with line-based structure

```
<player_count>
<player1_name>
<company1_name>
<company1_symbol>
<player2_name>
<company2_name>
<company2_symbol>
...
```

Expected Behavior:

- If file exists and valid: Load all values into GameConfig
- If file missing: Use defaults (playerCount=2, empty strings)
- Configuration loaded at startup

Error Handling:

- File not found: Log to stderr, return default configuration
- File cannot be opened: Log to stderr, return default configuration
- Parse errors: Partial data loaded, remaining fields use defaults

3.3.2 Card Definition File (JSON)

File Location: Assets directory + "cards.json"
Format: JSON array of card objects
Example Card Object:

```json
{
  "name": "Harvest",
  "description": "Gain resources at start of day",
  "copies": 3,
  "triggers": {
    "onStartOfDay": [
      {
        "action": "addResource",
        "type": "Gold",
        "amount": 2
      }
    ]
  }
}
```

Field Descriptions:

- `name`: Required string, card identifier
- `description`: Optional string, card text
- `copies`: Optional integer (default 1), number of duplicates to create
- `triggers`: Optional object mapping trigger names to action arrays

Expected Behavior:

- Parse all card objects
- Create N copies of each card based on "copies" field
- Add all cards to deck
- Display "[N] cards loaded from [path]"

Error Handling:

- File not found: Log "Error: Could not find [path]", deck remains empty
- File cannot be opened: Log error, deck remains empty
- Malformed JSON: Catch exception, log "Error parsing JSON from [path]: [details]", deck remains empty
- Missing "name" field: Use "Unnamed Card" as default
- Missing "description": Use empty string
- Missing "copies": Default to 1
- Invalid "copies" type: Ignore field, use default 1
- Missing "triggers": Card created with empty trigger map

3.3.3 Font File

File Location: Assets directory + "consolas.ttf"
Format: TrueType Font file
Expected Behavior:

- Font loaded into SFML font object
- Used for all text rendering

Error Handling:

- File not found or load failure: Log "Error: Could not load font at [path]"
- Application continues with SFML default font
- No crash occurs

3.4 Graphical Outputs

3.4.1 Game Window

Window Properties:

- Size: 800x600 pixels
- Title: "Hex Board"
- Resizable: Yes (handled by SFML)

Visual Elements:

- Hexagonal tiles rendered with colors
- Company symbols displayed on owned tiles
- Console interface at bottom of screen
- Day counter and current player indicator
- Input text field with cursor

3.4.2 Tile Rendering

Visual Properties:

- Shape: Regular hexagon
- Fill color: Determined by tile color property
- Border: Visible outline
- Owner symbol: Displayed at tile center if owned

- Neutral tiles: Gray color, no symbol

Coordinate Conversion: Cube coordinates (x, y, z) converted to pixel positions using:

- Hexagon size and orientation parameters
- Camera offset for panning
- Zoom level for scaling

4 Data Flow and Program Execution

4.1 Startup Sequence

1. `main()` calls `StartupMenu::StartMenuLoop()`
2. Load configuration from disk via `loadConfig()`
3. Display main menu and wait for user choice
4. If Settings selected:
    - Call `settingsMenu()`
    - Accept player count with validation
    - Accept player/company information
    - Save configuration via `saveConfig()`
    - Return to main menu
5. If Start Game selected:
    - Calculate board radius via `getAutoRadius()`
    - Create Company objects from configuration
    - Construct Game object with radius and companies
    - Add players to game via `addPlayer()`
    - Call `game.setup()`
    - Call `game.mainLoop()`

4.2 Game Setup Sequence

1. Generate board tiles (Board constructor calls `generateBoard()`)
2. Randomize initial tile state:
    - Collect all tile coordinates
    - Shuffle coordinate list

- Assign random colors and owners to first 50% of tiles
- Set remaining tiles to neutral

3. Initialize card system:
   - Create draw deck
   - Load cards from "cards.json" via `loadFromJsonFile()`
   - Shuffle draw deck
   - Create empty discard deck
   - Add both decks to game deck list
4. Deal starting card to first player
5. Initialize command handlers (register all commands in map)

4.3 Main Game Loop

1. While window is open:
   - `renderer->handleEvents()` - Process SFML events
     - Window close event: Close window
     - Keyboard input: Route to console
     - Mouse events: Handle camera pan/zoom
   - While console has queued commands:
     - Retrieve next command via `console->nextCommand()`
     - Call `executeCommand(cmd)`
     - Parse command string into action and arguments
     - Lookup action in `commandHandlers` map
     - If found: Execute handler lambda with argument stream
     - If not found: Display "Unknown command" error
   - `renderer->render()` - Draw all visual elements
     - Clear window
     - Draw all board tiles
     - Draw console interface
     - Draw UI overlays
     - Display window

4.4 Command Execution Flow

1. User types command and presses Enter
2. Console adds command string to queue
3. Main loop retrieves command from queue
4. `executeCommand()` parses action word
5. Handler lambda retrieved from `commandHandlers` map
6. Lambda executes with remaining argument stream
7. Handler performs validation:
    - Parse required arguments from stream
    - Check for parse failures (missing/invalid args)
    - Validate indices against bounds
    - Validate string values against allowed sets
8. If validation fails:
    - Display error or usage message
    - Return without modifying state
9. If validation succeeds:
    - Call appropriate Game method
    - Modify game state
    - Display success message

4.5 Turn Progression Flow

1. Current player takes actions (resource spending, card playing, building)
2. Player executes `end_turn` command
3. `Game::endTurn()` called:
    - Display "[Player] ended their turn"
    - Increment `currentActivePlayerIndex`
    - If index exceeds player count:
        - Reset index to 0
        - Increment `currentDay`
        - Call `startNewDay()`

- Display "[NextPlayer] starts their turn"

4. If new day started:
   - For each player:
     - For each card in `playedCards`:
       * Call `card.executeTrigger("onStartOfDay", player)`
       * Execute all actions in trigger (e.g., add resources)

4.6 Card Play Flow

1. User executes `play_card <name>` command
2. `Game::playCardForPlayer()` called:
   - Validate player index
   - Call `player.playCard(cardName)`
     - Search `heldCards` for matching name
     - If not found: Return false
     - If found: Move card to `playedCards`, return true
   - If play succeeded:
     - Find card in `playedCards` vector
     - Call `card.executeTrigger("onPlay", player)`
     - For each action in "onPlay" trigger:
       * Parse action type
       * Execute effect (e.g., `addResource`)
       * Log effect to console
     - Display "[Player] played card: [Name]"
   - If play failed:
     - Display error message

4.7 Resource Transaction Flow

4.7.1 Giving Resources

1. `Game::giveResourceToPlayer()` called
2. Validate and resolve player index
3. If validation fails: Return without change
4. Access player's `resources` map
5. Add amount to resource key (auto-creates if new)
6. Log transaction to console

4.7.2 Spending Resources

1. `Game::spendResourceFromPlayer()` called
2. Validate and resolve player index
3. If validation fails: Return false
4. Search for resource in player's map
5. If resource not found OR amount insufficient:
   - Log error to console
   - Return false (transaction failed)
6. If sufficient resources:
   - Deduct amount from resource value
   - Log transaction to console
   - Return true (transaction succeeded)

5 Error Handling Strategies

5.1 Input Validation Strategy

The program employs multi-layered validation:

1. **Stream State Checking:** After parsing arguments, check `istringstream.fail()` to detect missing or type-mismatched arguments
2. **Range Validation:** Compare numeric inputs against valid ranges before use
3. **Existence Checking:** Verify objects exist before dereferencing (null pointer checks)
4. **String Validation:** Compare string inputs against allowed sets

5.2 Error Recovery Mechanisms

5.2.1 Non-Fatal Errors

For most errors, the program:

- Logs descriptive error message to console or stderr
- Returns early without modifying state
- Continues accepting new commands
- Never crashes or enters undefined state

Examples: Invalid player index, missing card, insufficient resources

5.2.2 Degraded Functionality

For missing assets:

- Log error with full file path
- Continue execution with defaults or reduced functionality
- Example: Missing font uses SFML default font

5.2.3 Safe Defaults

For missing configuration:

- Use sensible default values
- Example: Missing settings.txt results in 2-player game with empty names
- Program remains fully functional

5.3 Defensive Programming Practices

5.3.1 Null Pointer Protection

Before dereferencing pointers:

```
Tile* tile = getTile(coord);
if (!tile) {
    // Log error and return
    return;
}
// Safe to use tile
```

5.3.2 Boundary Checking

Before array/vector access:

```
if (playerIndex < 0 || playerIndex >= players.size()) {
    // Log error with valid range
    return false;
}
// Safe to access players[playerIndex]
```

5.3.3 Stream State Management

After input operations:

```
std::cin >> value;
if (std::cin.fail()) {
    std::cin.clear();  // Clear error flags
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
    // Display error and handle gracefully
}
```

5.3.4 Exception Handling

For operations that can throw:

```
try {
    file >> jsonData;
} catch (const std::exception& e) {
    std::cerr << "Error parsing JSON: " << e.what() << "\n";
    return;  // Safe failure
}
```

5.3.5 Transaction Semantics

For resource operations:

- Check before modifying (all-or-nothing)
- Return boolean success indicator
- Caller can respond to failure appropriately

5.4 Error Message Guidelines

All error messages follow consistent format:

- Prefix with "Error: " for user-facing errors
- Include specific values that caused error
- Provide valid ranges or options when applicable
- Use descriptive context (e.g., "Player index 5 is out of range. Max valid index: 2")

6 Special Considerations

6.1 Player Index Convention

The program uses -1 as a special value meaning "current active player":

- Most commands accept optional player index parameter
- If omitted, defaults to -1
- `validateAndSetPlayerIndex()` converts -1 to `currentActivePlayerIndex`
- Simplifies commands for active player

6.2 Cube Coordinate System

Hexagonal grid uses cube coordinates (x, y, z) with constraint x + y + z = 0:

- Simplifies neighbor calculations (add direction vectors)
- Simplifies distance calculations
- All coordinate inputs must satisfy constraint (not enforced, assumed valid)

6.3 Dynamic Resource System

Resources are not predefined:

- Any string can be a resource type
- Resources auto-created on first `giveResource`
- Map structure allows unlimited resource types
- Cards and commands can reference any resource name

6.4 Card Trigger System

Cards support event-driven effects:

- Triggers are string identifiers (e.g., "onPlay", "onStartOfDay")
- Each trigger maps to array of action objects
- Actions executed sequentially when trigger fires
- System is extensible (new action types can be added)
- Current supported actions: "addResource"

6.5 Command Parsing Ambiguity

Known limitation in multi-word card name parsing:

- Parser attempts to interpret last token as player index
- Card names ending with numbers may be misparsed
- Example: "Level 2" may be parsed as card="Level" and playerIndex=2
- Workaround: Avoid card names ending with digits
- Alternative: Always specify explicit player index

6.6 Color Validation Weakness

Current implementation has partial validation:

- Invalid colors trigger warning message
- But tile color is still set to invalid value
- Does not crash or cause visual errors
- Should be strengthened in future to reject invalid colors

7 Building an Equivalent Program

To recreate Spotlight functionality, a programmer would need to:

7.1 Core Requirements

1. Implement hexagonal grid using cube coordinate system
2. Create tile ownership and property system
3. Build player/company data structures with resource management
4. Implement card system with JSON loading and trigger execution
5. Create deck management (draw, shuffle, move operations)
6. Build command parsing and dispatch system
7. Implement turn progression with day cycling
8. Add console interface with pagination
9. Integrate SFML for window and rendering
10. Handle all error cases as documented

7.2 Key Algorithms

1. **Board Generation:** Triple nested loop over x, y with z=-x-y, filtering by radius
2. **Neighbor Finding:** Add six direction vectors to coordinate, check existence
3. **Tile Randomization:** Shuffle coordinate list, assign first half to players
4. **Command Dispatch:** Map of string to lambda functions
5. **Turn Wraparound:** Modulo arithmetic with day increment check
6. **Card Name Parsing:** Tokenize, attempt numeric conversion of last token

7.3 Critical Implementation Details

1. Use `std::map` for tile storage (fast coordinate lookup)
2. Use `std::map` for resource storage (dynamic types)
3. Use `std::vector` for player and card lists (ordered iteration)
4. Use `std::function` for command handlers (flexible dispatch)
5. Always validate indices before vector access
6. Always check pointers before dereferencing
7. Always check stream state after parsing
8. Use early returns for error handling (fail-fast)
9. Log all errors with context
10. Never crash on invalid input

8 Future Enhancement Opportunities

While not currently implemented, these areas could be improved:

1. Strengthen color validation to prevent invalid assignments
2. Add coordinate validation in high-level functions to reduce redundant error messages
3. Implement font fallback mechanism for missing asset files
4. Improve card name parser to handle ambiguous cases
5. Add automated test harness for regression testing
6. Implement save/load game state functionality
7. Add network multiplayer support
8. Create AI opponents
9. Expand card action types beyond resource manipulation
10. Add sound effects and music
11. Implement animations for game actions
12. Create visual card display instead of text-only
13. Add undo/redo functionality
14. Implement game rules validation (e.g., adjacency requirements)

9 Conclusion

Spotlight is a well-structured digital board game prototype with robust error handling and modular design. The program gracefully handles invalid inputs, missing files, and edge cases without crashing. The command system provides flexible text-based control, while SFML provides graphical feedback. The cube coordinate system and event-driven card system demonstrate thoughtful technical design choices.

A programmer reading this design document should be able to implement an equivalent program by following the module descriptions, input/output specifications, error handling strategies, and data flow diagrams provided herein.