Complete Game Design Document: Minotaur Remake

1. Game Overview

1.1. Game Concept

"Minotaur" is a first-person dungeon crawler where the player explores a multi-level maze, battles monsters, collects treasures, and manages resources. The ultimate goal is to find and defeat the Minotaur, who guards the legendary Treasure of Tarmin.

1.2. Game Modes

The game will feature two distinct modes of play:

- **Classic Mode:** This mode will be a faithful, 1:1 recreation of the original Intellivision *Treasures of Tarmin* experience. It will use the original game's mechanics, difficulty scaling, and aesthetic. The initial development of this project will focus on building the complete Classic Mode.
- **Enhanced Mode:** This mode is a long-term goal that will build upon the Classic foundation. It will introduce modernized features, including:
    - **Roguelike Progression:** Persistent progression through a skill tree and new items, monsters, and abilities to unlock over multiple runs.
    - **Crafting & Survival:** Players can butcher slain monsters for flesh, which can be cooked and consumed. Monster flesh will have procedurally generated properties, creating a risk/reward system where consumption can lead to either a temporary buff ("perk") or a negative status effect ("malady").
    - **Dynamic Combat:** Monster sprites will be component-based, allowing for dynamic dismemberment during combat. Losing limbs can alter a monster's attack patterns or abilities, and the system will provide visceral feedback upon their defeat.
    - Open world
    - Maze colors changes as you go down and are themed differently, different spawns, special effects
    - Biome based exploration once you exit the maze
    - What if combat was balatro, you find jokers and tarot etc in the maze, rarity scales with progress, to defeat monster you need to meet the ante, etc. You can still equip weapons, armor, rings books, scrolls etc and they have bonuses on top of jokers. Research ancient card games that don’t have copyright

1.3. Target Audience

This remake is for fans of classic dungeon crawlers, retro gaming enthusiasts, and new players who enjoy challenging, exploration-based games.

1.4. Core Gameplay Loop

The core gameplay loop consists of:

- **Exploring:** Navigating through mazes, discovering rooms, and finding hidden doors.
- **Fighting:** Engaging in turn-based combat with a variety of monsters.
- **Looting:** Finding weapons, armor, magical items, and treasures.
- **Managing:** Organizing inventory, monitoring health and spiritual power, and using items strategically.
- **Descending:** Progressing to deeper and more dangerous levels of the dungeon.

2. Technical Specification & Architecture

2.1. Target Java Version

The project will be developed using **Java 17 LTS (Long-Term Support)**. This version provides a modern, stable foundation with long-term support and significant performance benefits.

2.2. Core Framework

The game will be built using the **libGDX framework**. This choice is based on the following:

- **Balance:** It offers the perfect balance of high-level, game-specific tools (2D rendering, UI, audio) and low-level control, which is ideal for our project.
- **Fit for Purpose:** Its features are tailor-made for the requirements of this game, allowing for efficient development.
- **Cross-Platform:** It enables easy deployment to Windows, macOS, and Linux from a single codebase.

2.3. Package Structure

The root Java package for this project will be `com.bpm.minotaur`. All subsequent code will be organized into sub-packages within this structure.

2.4. Data Persistence

A formal database will not be used for this project. A file-based approach is simpler, faster, and better suited to a single-player game.

- **Static Game Data:** All configuration data (monster stats, item properties, etc.) will be stored in human-readable **JSON files**. This data will be loaded into memory at the start of the game.
- **Dynamic Save Data:** Player progress will be saved by **serializing the game state** (e.g., the `Player` and `Maze` objects) to a local file.
- **User Settings:** Game settings (e.g., audio volume, resolution) will be saved to a separate local file (`settings.json`) to persist user preferences between sessions.

2.5. Maze Generation Algorithm

Levels will be generated using a **Segment-based Assembly with Rotation** algorithm, based on analysis of the original game's logic.

- **Bitmask Definitions:** Wall data for 6x6 segments will be defined using an 8-bit mask (TTRRBBLL) to represent walls, doors, and openings on each of the four sides of a tile.
- **Segment Library:** A library of 16 base 6x6 segments will be stored in a data file.
- **Rotational Assembly:** A full 12x12 maze will be constructed by selecting four random segments, rotating them to the correct orientation (NW, NE, SW, SE), and assembling them. This ensures seamless connectivity between the segments.

2.6. Domain Model

The following UML diagram outlines the primary entities (classes) and their relationships, forming the architectural blueprint for the game. The `DebugManager` is implemented as a Singleton for global access.

```
classDiagram

    class Game {

        +startGame(SkillLevel)

        +mainLoop()

        +endGame()

    }



    class SettingsManager {

        -float masterVolume

        -String resolution
```

```
    +load()
    +save()
}

class DebugManager {
    +boolean isDebugOverlayVisible
    +boolean isGodMode
    +toggleOverlay()
}

class Player {
    -int warStrength
    -int spiritualStrength
    -int food
    -int arrows
    -int x
    -int y
    -Direction facing
    +move(Direction)
    +attack(Monster)
    +useItem(Item)
    +rest()
}
```

```
class Inventory {
    -Item rightHand
    -Item leftHand
    -Item[] backpack
    +swapHands()
    +swapPack()
    +rotatePack()
    +pickup(Item)
    +drop()
}

class Monster {
    -String name
    -int warStrength
    -int spiritualStrength
    -MonsterType type
    -Color color
    +attack(Player)
}

class Item {
    -String name
    -ItemType type
    -Color color
```

```
}

class Maze {
    -int level
    -Tile[][] grid
    +getTile(x, y)
}

class Tile {
    -TileType type
    -Monster monster
    -Item item
}

class MazeGenerator {
    +generate(level, skillLevel) Maze
}

class Quadrant {
    -TileType[][] grid
    -int difficulty
}

enum Direction { NORTH, SOUTH, EAST, WEST }
```

```
    enum SkillLevel { EASIEST, EASY, MEDIUM, HARD }

    enum MonsterType { BAD, NASTY, HORRIBLE }

    enum ItemType { WAR_WEAPON, SPIRITUAL_WEAPON, ARMOR, RING, CONTAINER,
TREASURE, USEFUL }

    enum TileType { WALL, FLOOR, DOOR, HIDDEN_DOOR, GATE, LADDER }

    enum Color { TAN, ORANGE, BLUE, GRAY, YELLOW, WHITE, PLATINUM, PINK, RED,
PURPLE }



    Game "1" -- "1" Player

    Game "1" -- "1" SettingsManager

    Game "1" -- "1" DebugManager

    Player "1" -- "1" Inventory

    Inventory "1" -- "0..8" Item

    Maze "1" -- "144" Tile

    Player "1" -- "1" Maze : located in

    Player "1" -- "0..1" Monster : in combat with

    MazeGenerator "1" ..> "4" Quadrant : uses

    MazeGenerator "1" ..> "1" Maze : creates

    Tile "1" -- "0..1" Monster

    Tile "1" -- "0..1" Item
```

2.7. Sequence Diagram: Typical Game Session

This diagram illustrates the sequence of interactions between the core classes during a typical game session, from startup to a combat encounter resulting in the player's death.

```
sequenceDiagram

    participant Game
```

```
participant MazeGenerator

participant Player

participant Maze

participant Monster


%% -- Game Start --

Game->>Game: startGame(SkillLevel.EASY)

Game->>Player: new Player()

Game->>MazeGenerator: generate(level=1, skillLevel)

MazeGenerator-->>Game: return new Maze

Game->>Game: mainLoop()


%% -- Gameplay Loop --

loop Player Interaction

    Game->>Player: processInput()

    alt Player moves

        Player->>Maze: getTile(newX, newY)

        Maze-->>Player: return Tile

        opt Tile contains Monster

            Player->>Game: notifyMonsterEncounter(Monster)

            Game->>Monster: activate()


            loop Combat

                Player->>Monster: attack(playerWeapon)
```

```
            Monster-->>Player: takeDamage()

            opt Monster is alive
                Monster->>Player: attack(monsterWeapon)
                Player-->>Player: takeDamage()

                opt Player is dead
                    Game->>Game: endGame()
                    Note over Game: Player Defeated
                end
            end

            opt Monster is dead
                Game->>Maze: remove(Monster)
                Note over Game: End Combat
            end
        end
    end
  else Player uses item
    Player->>Player: useItem(someItem)
  end
end

%% -- Game End --
```

```
Game->>Game: showGameOverScreen()
```

3. Development Roadmap

This section outlines the high-level tasks for the iterative development of the game.

- **Phase 1: Core Engine Foundation**
  - [X] **Task 1.1:** Implement the main `MinotaurGame` class.
  - [X] **Task 1.2:** Create a basic screen management system (`BaseScreen`).
  - [X] **Task 1.3:** Implement a `MainMenuScreen` with a "Start Game" button.
  - [X] **Task 1.4:** Implement a `GameScreen` to serve as the main play area.
  - [X] **Task 1.5:** Implement an input handling system for basic player actions.
  - [ ] **Task 1.6:** Implement a `SettingsManager` to handle game preferences. (Moved to Phase 6)
  - [X] **Task 1.7:** Implement a `DebugManager` and a key to toggle a debug overlay.
- **Phase 2: Game World & Player (REVISED)**
  - [X] **Task 2.1:** Create the data classes for `Player`, `Maze`, and `Tile`.
  - [ ] **Task 2.2 (REVISED):** Re-implement `MazeGenerator` using Segment-based Assembly with Rotation.
  - [X] **Task 2.3:** Implement a 2D top-down debug renderer.
  - [X] **Task 2.4:** Implement player movement within the maze.
  - [X] **Task 2.5:** Add the 2D maze view to the debug overlay.
  - [X] **Task 2.6:** Create unit tests for `Player` and `MazeGenerator` classes.
  - [X] **Task 2.7 (REVISED):** Re-implement `FirstPersonRenderer` using a painter's algorithm with true perspective.
- **Phase 3: Items & Inventory**
  - [X] **Task 3.1:** Create the data classes for `Item` and its subclasses.
  - [X] **Task 3.2:** Implement the `Inventory` management system.
  - [X] **Task 3.3:** Implement picking up and dropping items.
  - [ ] **Task 3.4:** Render items in the world and in the UI. (Moved to Phase 6)
  - [X] **Task 3.5:** Create unit tests for the `Inventory` class.
- **Phase 4: Monsters & Combat**
  - [X] **Task 4.1:** Create the `Monster` data class.
  - [X] **Task 4.2:** Implement the turn-based combat logic.
  - [X] **Task 4.3:** Render monsters in the world.
  - [X] **Task 4.4:** Implement the player and monster attack sequences.
  - [X] **Task 4.5:** Add combat information to the debug overlay.
  - [X] **Task 4.6:** Create unit tests for combat logic.
- **Phase 5: UI & HUD**
  - [X] **Task 5.1:** Implement the main game HUD (stats, scores, etc.).

    - [X] **Task 5.2:** Implement the inventory UI.
    - [X] **Task 5.3:** Implement the `CastleMap` and `MazeMap` screens.
    - [X] **Task 5.4:** Implement a `SettingsScreen` to modify game options.
- **Phase 6: Polishing & Finalization (Classic Mode v1.0)**
    - [ ] **Task 6.1:** Add sound effects and audio cues.
    - [ ] **Task 6.2:** Implement the save/load game functionality.
    - [ ] **Task 6.3:** Implement skill levels and game balancing.
    - [ ] **Task 6.4:** Bug fixing and performance tuning.
    - [ ] **Task 6.5 (from 2.7):** Evolve renderer to use trapezoids and vanishing points for a true perspective effect.
    - [ ] **Task 6.6 (from 3.4):** Render items in the UI (inventory, hands).
    - [ ] **Task 6.7 (from 1.6):** Implement `SettingsManager` and connect it to the `SettingsScreen`.
- **Phase 7: Enhanced Mode - Core Systems (Post-v1.0)**
    - [ ] **Task 7.1:** Design the roguelike progression systems (skill tree, unlocks).
    - [ ] **Task 7.2:** Implement the skill tree and persistent player profile.
    - [ ] **Task 7.3:** Design the crafting, cooking, and survival mechanics.
    - [ ] **Task 7.4:** Implement the buff/debuff (perk/malady) system.
- **Phase 8: Enhanced Mode - Content & Features (Post-v1.0)**
    - [ ] **Task 8.1:** Implement a component-based sprite system for monster dismemberment.
    - [ ] **Task 8.2:** Create new items, abilities, and monsters for Enhanced Mode.
    - [ ] **Task 8.3:** Update combat logic to handle dismemberment effects.
    - [ ] **Task 8.4:** Update UI to support new Enhanced Mode features.
- **Phase 9: Process Automation (Research Spike)**
    - [ ] **Task 9.1:** Research and select an AI agent framework (e.g., CrewAI, LangChain).
    - [ ] **Task 9.2:** Define agent roles (e.g., `Architect`, `Coder`, `QA_Tester`, `Project_Manager`) to mirror our development process.
    - [ ] **Task 9.3:** Implement a proof-of-concept workflow using the selected framework and a personal Gemini API key to automate a small task from this roadmap.
    - [ ] **Task 9.4:** Evaluate the efficiency and effectiveness of the agent-based workflow compared to our current interactive process.

4. Core Mechanics

4.1. Player Movement

- The player navigates the world from a first-person perspective.
- Basic movements include moving forward, turning left, and turning right.
- The player can also "glance" left or right without changing their orientation and can retreat one step.

4.2. Combat System

- Combat is turn-based.
- The player can attack monsters with weapons or magical items held in their right hand.
- A shield can be held in the left hand for defense.
- Both the player and the monsters have "War Strength" and "Spiritual Strength" scores, which determine their combat effectiveness.
- Some weapons and items are consumed upon use.

4.3. Inventory Management

- The player has a backpack that can hold up to six items.
- The player can hold one item in each hand.
- Items can be swapped between hands, between the right hand and the pack, and rotated within the pack.
- The player can pick up and drop items.

4.4. Resource Management

- **Food:** Consumed to restore War and Spiritual Strength by resting.
- **Arrows:** Ammunition for bows and crossbows.

5. Player Character

5.1. Attributes

- **War Strength:** Represents the player's physical power.
- **Spiritual Strength:** Represents the player's magical power.
- **Armor/Ring Defenses:** Reduces damage from physical and spiritual attacks, respectively.
- **Weapon Score:** The power of the currently equipped weapon.

5.2. Actions

The player can perform the following actions:

- Move (Forward, Turn Left/Right, Glance Left/Right, Retreat)
- Interact with objects (Pick Up/Drop, Swap Hands, Rotate Pack, Swap Pack)
- Open doors and containers
- Use items
- Attack
- Rest
- Descend ladders to the next level

6. Items

6.1. War Weapons

Type	Special Secret
Bows	Use 1 arrow at a time; may break
Crossbows	Use 1 arrow at a time; may break
Knives	Vanish when used in an attack
Axes	Vanish when used in an attack
Darts	Vanish when used in an attack
Spears	Vanish when used in an attack

Color	Material	Power
Tan	Wood/Leather	Regular Power
Orange	Rusty Iron	Greater Power

Blue	Steel	Fair Power
Gray	Silver	Medium Power
Yellow	Gold	High Power
White	Platinum	Super Power

6.2. Spiritual Weapons

Type	Special Secret
Scrolls	Reusable; may break
Books	Reusable; may break
Small Fireballs	Vanish when used in attack
Small Lightning Bolts	Vanish when used in attack
Large Fireballs	Vanish when used in attack
Large Lightning Bolts	Vanish when used in attack

Color	Power
Blue	Regular Power
Gray	Fair Power
White	Medium Power
Pink	High Power
Red	Super Power
Purple	Greater Power

6.3. Armor & Shields

Provide defense against war weapons.

Type	Special Secret
Small Shields	Hold in left hand during fight
Large Shields	Hold in left hand during fight

Gauntlets	Use to put on; best color is kept
Hauberks	Use to put on; best color is kept
Helmets	Use to put on; best color is kept
Breastplates	Use to put on; best color is kept

Armor Colors & Power are the same as War Weapons.

6.4. Rings

Provide defense against spiritual weapons.

Type	Special Secret
Small Ring	Use to put on; best color is kept
Large Ring	Use to put on; best color is kept

Ring Colors & Power are the same as Spiritual Weapons.

6.5. Containers

Type	Contents Quality	Special Secret

Money Belts	Regular	Open to grab contents
Small Bags	Better	Open to grab contents
Large Bags	Great	Open to grab contents
Boxes	Fair	Locked; use key to open
Packs	Medium	Locked; use key to open
Chests	Super	Locked; use key to open

Containers may contain bombs. Better containers have nastier bombs.

Color	Quality
Tan	Mild Quality
Orange	Good Quality
Blue	Best Quality

6.6. Treasures

Type	Silver Value	Gold Value	Platinum Value
Coins	10	30	170
Necklaces	20	70	200
Ingots	50	350	450
Lamps	100	150	220
Chalices	120	250	400
Crowns	300	500	600

6.7. Useful Items

Type	Color	Special Secret
Keys	Tan	Unlock tan containers
	Orange	Unlock tan or orange containers
	Blue	Unlock any container

War Books	Blue	Increase war exp; max strength 99
(Vanish on use)	Pink	Increase war exp; max strength 149
	Purple	Increase war exp; max strength 199
Spiritual Books	Blue	Increase spiritual exp; max strength 49
(Vanish on use)	Pink	Increase spiritual exp; max strength 74
	Purple	Increase spiritual exp; max strength 99
Small Potions	Blue	Refresh war & spiritual strength
(Vanish on use)	Pink	Find better items in containers
	Purple	Become invisible
Large Potions	Blue	Raise war strength by 10
(Vanish on use)	Pink	Raise spiritual power by 10
	Purple	Switch war & spiritual strength traits

Special Books	Blue	Teleport Books: Move forward through walls
(Never vanish)	Pink	Vision Books: See through walls for a time
	Purple	Midas Books: Turn items at feet to platinum

7. Enemies

7.1. Monster Categories

- **Bad Monsters:** Use spiritual weapons only.
- **Nasty Monsters:** Use war weapons only.
- **Horrible Monsters:** Use either spiritual or war weapons.

7.2. Monster Roster

- **Bad Monsters**
  - Giant Ants (Blue, Pink, Purple)
  - Dwarfs (Yellow, Tan, Orange)
  - Dwarfs with Shields (Blue, Pink, Purple)
  - Giant Scorpions (Yellow, Tan, Orange)
  - Giant Snakes (Blue, Pink, Purple)
- **Nasty Monsters**
  - Skeletons (White, Gray, Orange) - With & Without Shields
  - Cloaked Skeletons (White, Gray, Orange) - With & Without Shields
  - Giants (Yellow, Tan, Orange)
  - Wraiths (White, Gray, Orange) - With & Without Shields
- **Horrible Monsters**
  - Alligators (Blue, Pink, Purple)
  - Ghouls (Blue, Pink, Purple)
  - Dragons (Blue, Pink, Purple)
- **The Minotaur**
  - Purple (Guards the Tarmin treasure)

8. World and Level Design

8.1. The Castle and Mazes

- The game takes place in a castle with a vast system of mazes and dungeons.
- There are 256 levels in total.
- Each level is a 12x12 grid.

8.2. Level Features

- **Rooms and Corridors:** The basic building blocks of the maze.
- **Doors:** Can be opened to access new areas.
- **Hidden Doors:** Can be found by searching suspicious walls.
- **Gates:** Transport the player to an adjacent maze and may alter their stats.
- **Ladders:** Allow the player to descend to the next level.
- **Eyeball Murals:** Indicate the type of maze ahead (war, spiritual, or mixed).

8.3. Maps

- **Maze Map:** A top-down view of the current level, showing the player's immediate surroundings.
- **Castle Map:** An overview of the entire dungeon, showing the player's location across all levels.

9. Aesthetic & Style Guide

9.1. Overall "Vibe"

The primary feeling of the game is one of **tense, isolated exploration**. The player is alone in a hostile, confusing labyrinth. The atmosphere is not one of epic fantasy, but of claustrophobic, methodical survival. The simple graphics and sounds force the player to use their imagination, making the unseen threats all the more menacing.

- **Keywords:** Claustrophobic, Tense, Mysterious, Methodical, Isolating, Unforgiving, Retro, Minimalist.

9.2. User Interface (UI)

The UI is purely functional, inspired by early computer RPGs. It should be clean, easy to read, and present information directly without unnecessary artistic flair.

- **Layout:** A large first-person viewport with a dedicated, static HUD at the bottom of the screen. The Main Menu will provide access to a Settings screen. A toggleable debug overlay will be available for development and testing.
- **HUD Elements:**
    - **Text:** Use a blocky, pixelated, sans-serif font for all text elements (scores, level number, etc.). Text should be high-contrast (e.g., white or bright colors on a black background).

    - **Icons:** Inventory items, held items, and objects at the player's feet should be represented by simple, iconic, pixel-art sprites.
    - **Compass:** A simple, functional compass should always be visible.
- **No Frills:** Avoid decorative borders, complex animations, or transparency in the UI. It should feel like a straightforward instrument panel.

9.3. Sound Design

Sound is a critical gameplay element, used to build tension and provide vital information to the player. The absence of a continuous musical score is a key feature.

- **Atmosphere:** The dominant sound is the player's footsteps. It is a singular, low-frequency **"thud"** that repeats at a steady rhythm as the player moves. This creates a baseline of quiet tension.
- **Action Sounds:** All player actions should have simple, distinct, 8-bit style sound effects:
    - **Player Attack:** A sharp, high-pitched **"zap"** or **"pew"** sound, characteristic of early electronic games.
    - **Monster Attack:** A lower-pitched, slightly distorted **"buzz"** or **"hiss"** sound.
    - **Item Pickup:** A quick, positive-sounding **"bloop"** or **"chime"**.
    - **Door Open:** A short, mechanical **"sch-lick"** sound.
    - **Descend Ladder:** A descending series of quick tones, like a synthesized **"whoosh"** downwards.
- **No Music:** There should be no background music during exploration. A simple, short musical sting could be used for significant events like descending a level or dying.

9.4. Visuals & Graphics

9.4.1. Maze Design

The maze itself should feel oppressive and geometrically rigid.

- **Geometry:** Strictly grid-based. Corridors are long and narrow, and rooms are simple rectangles. The perspective should be clean but basic, with clear vanishing points.
- **Rendering Style:** The first-person view will be a "pseudo-3D" renderer. It will use 2D drawing primitives (lines and polygons) to create the illusion of a 3D space, calculating vanishing points to draw trapezoids for floors, ceilings, and walls. This will maintain the simple, clean, color-blocked aesthetic of the original game while providing a true sense of perspective.
- **Core Color Palette:** The color palette is central to the game's identity. It is defined by high-contrast, solid colors.
    - **UI Background (`#000000`):** A pure, solid black serves as the backdrop for the entire user interface, making all text and icons pop.
    - **Maze Floor/Ceiling (`#5B602F`):** A muted, military-style olive green for all floor and ceiling surfaces. This grounds the scene and provides a dark, earthy base.

    - **Dark Wall Green (`#005945`):** A deep, dark teal-green used for the main wall color.
    - **Light Wall Green (`#00A95A`):** A vibrant, brighter green used for alternating wall panels and creating a sense of depth and perspective.
    - **Minotaur Magenta (`#D73B93`):** A shocking, vibrant magenta/pink used for the Minotaur and other high-level threats. This makes key enemies stand out dramatically from the green environment.
    - **UI Text & Icons:**
        - **Primary (`#FFFFFF`):** White for key info like inventory icons.
        - **War Score (`#45BBDC`):** A bright cyan for the War Strength score.
        - **Spiritual Score (`#F1E03C`):** A vibrant yellow for the Spiritual Strength score.

9.4.2. Monster & Item Design

The visual design for entities should be iconic, simple, and heavily reliant on color to convey information.

- **Sprites:** All monsters and items should be 2D sprites. They should be designed with a clear, recognizable silhouette.
- **Animation:** Keep animation minimal. Monsters can have a simple idle animation (e.g., a slight bob or shimmer) but should largely be static sprites that slide toward the player during combat. Item-use effects can be simple flashes or particle effects.
- **Color-Coding:** This is a crucial part of the original's design language. The same sprite should be used for monsters or items of the same type, with the color being the primary indicator of its power or variant (e.g., a blue ant is weaker than a purple ant). This system should be clear and consistent.

10. Skill Levels

The game will feature four skill levels, which affect:

- The number of maze levels.
- The player's starting vulnerability.
- The player's starting War/Spiritual Strength.
- The player's starting food and arrows.

11. Coding Standards

To ensure a high-quality, maintainable, and consistent codebase, this project will adhere to the following standards.

11.1. General Principles

- **Readability:** Code should be written to be easily understood by other developers. Prioritize clarity over cleverness.
- **Maintainability:** Follow the **SOLID** principles of object-oriented design to create a modular and flexible architecture.
- **Separation of Concerns:** Strictly separate game logic (the "model") from rendering and UI (the "view") and input handling (the "controller"). This will make the code easier to test and debug.

11.2. Naming and Formatting

- **Naming Conventions:** Adhere to standard Java naming conventions:
  - `PascalCase` for classes and interfaces.
  - `camelCase` for methods and variables.
  - `UPPER_SNAKE_CASE` for constants.
- **Formatting:** A consistent code style will be enforced. This includes standard indentation (4 spaces), line length (max 120 characters), and brace style. An automated formatter will be used.

11.3. Commit Messages

- **Conventional Commits:** All commit messages will adhere to the **Conventional Commits** specification. This provides a clear and standardized commit history. Common types will include:
  - `feat:` for new features.
  - `fix:` for bug fixes.
  - `docs:` for documentation changes.
  - `test:` for adding or refactoring tests.
  - `refactor:` for code changes that don't add features or fix bugs.

11.4. Architectural Patterns

- **Model-View-Controller (MVC) Variant:** The libGDX framework encourages a pattern where game state and logic are managed independently of how they are drawn to the screen. Our classes will reflect this:
  - **Model:** Classes like `Player`, `Monster`, `Maze`, and `Item` will contain only data and logic, with no rendering code.
  - **View:** Separate classes will be responsible for rendering the model objects to the screen (e.g., `MazeRenderer`, `PlayerSprite`).
  - **Controller:** Input handling will be managed in dedicated classes that translate user actions into changes in the model.
- **Data-Driven Design:** Game balance parameters (monster stats, item properties) will be loaded from external JSON files, not hardcoded in the source.

- **Singleton Pattern:** For global managers that should only have one instance (e.g., `DebugManager`, `SettingsManager`), a simple public static final instance will be used for easy, global access.

11.5. Quality Gates & Static Analysis

The project will aim to meet the following quality metrics, which can be measured by tools like SonarQube:

- **Unit Test Coverage:** All core game logic (combat calculations, item effects, player stats) must have a unit test coverage of **> 80%**.
- **Cyclomatic Complexity:** Methods should be kept simple and focused. The cyclomatic complexity of any single method should not exceed **10**.
- **Code Duplication:** Duplicated code blocks should be minimized, aiming for a duplication percentage of **< 5%**.
- **Security:** As a single-player, offline game, security risks are minimal. However, best practices such as avoiding code injection vulnerabilities will be followed. No sensitive information will be stored in the codebase.

11.6. Documentation

- **Javadoc:** All public classes, methods, and non-obvious private methods must have clear Javadoc comments explaining their purpose, parameters, and return values.
- **Inline Comments:** Use comments within methods to explain complex algorithms or business logic that is not immediately obvious from the code itself.