1 Overview & Goals

1.1 What is the Minigames Plugin?

The Minigames Plugin is a modular, extensible Minecraft server plugin that allows server owners to run multiple, simultaneous instances of different minigames such as TNT Run, Bed Wars, Sky Wars, and more—all within a single shared world or distributed across multiple worlds. The plugin is designed to be flexible, configurable, and integrated with popular Minecraft plugins, making it an essential component for minigame hub servers, PvP communities, or general-purpose gaming networks.

1.2 Primary Goals

Flexibility & Modularity: Goal: Allow the easy addition of new minigames without modifying core code.

How: Utilize a modular architecture with a `GameInstance` base class and plugin managers (`GameManager`, `ArenaManager`, `StatsManager`).

Outcome: Developers can build new game types by extending a consistent, documented API.

Scalability: Goal: Support multiple instances of the same minigame type running in parallel.

How: Manage arenas, players, and game states independently using unique IDs and instance tracking.

Outcome: Multiple groups can play the same minigame without conflicts.

Configurability: Goal: Let server admins control settings via YAML files and in-game GUIs.

How: Dynamic configuration system that merges file-based config with in-game edits, with live reload support.

Outcome: Seamless control for admins and rapid iteration.

Integration with Popular Plugins: Goal: Enhance functionality using Citizens2 (NPCs), PlaceholderAPI (dynamic stats), WorldEdit (arena setup), and Vault (economy).

How: Use plugin hooks and conditional dependencies to integrate features.

Outcome: Smooth experience with the broader Minecraft plugin ecosystem.

Player Engagement & Community Features: Goal: Encourage player retention and community building.

How: Add features like party systems, leaderboards, and achievements.

Outcome: Players enjoy a rich social experience, increasing server engagement.

1.3 Secondary Goals

Developer-Friendly Design: Goal: Make it easy for both humans and AI to understand and contribute.

How: Document each module thoroughly, use consistent naming conventions, and adopt standard design patterns.

Outcome: Fast onboarding and consistent code quality.

Extensibility: Goal: Prepare the plugin for future expansions like quests, seasonal events, and cross-server support.

How: Use event-driven architecture, plugin hooks, and a robust configuration system.

Outcome: Smooth addition of new features over time.

1.4 Target Audience

- **Server Owners:** Those running PvP hubs, minigame servers, or general-purpose Minecraft networks.
- **Plugin Developers:** Both experienced and new developers looking to expand the plugin with new games or features.
- **AI Assistants:** Tools like ChatGPT that can generate code or documentation for new features.

1.5 Summary

The Minigames Plugin is a powerful, modular foundation for Minecraft servers, combining game management, configuration, stats tracking, NPCs, and integrations into a single system. This document will guide you through each module and integration, ensuring both human and AI developers can confidently contribute to this project.

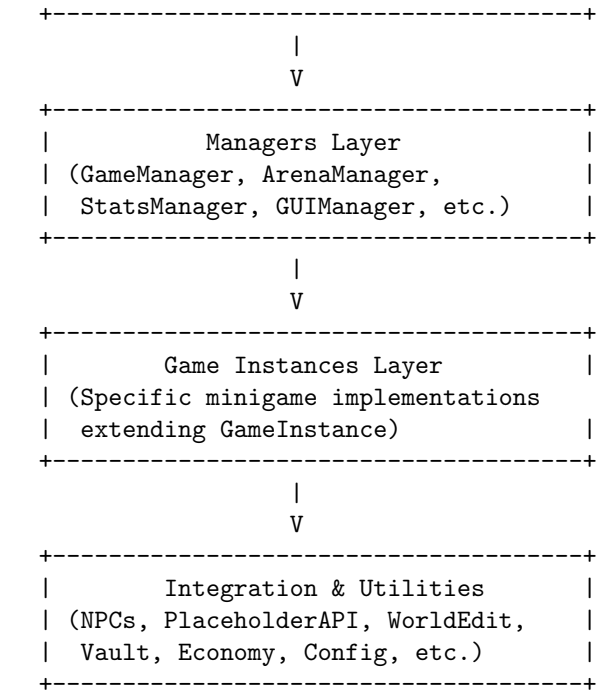
2 Architecture Overview

2.1 High-Level Plugin Structure

At its core, the Minigames Plugin is structured using a modular, layered architecture that separates concerns clearly between different systems, ensuring maintainability, extensibility, and testability.

Top-Level Architecture

```
+-------------------------------------+
|            MinigamesPlugin          |
| (Main plugin entry point and loader) |
```



2.2 Key Plugin Components

2.2.1 Main Plugin Class (`MinigamesPlugin`)

- Entry point for the plugin lifecycle.
- Loads configuration files and managers on enable.
- Registers commands, event listeners, and PlaceholderAPI expansions.
- Shuts down all active games cleanly on disable.

2.2.2 Managers Layer

Manages core subsystems:

- **GameManager:** Handles game lifecycle, registers minigames, and coordinates active game instances.
- **ArenaManager:** Manages arena definitions, allocations, and resets.
- **StatsManager:** Collects and persists player statistics.
- **PartyManager:** Manages player parties, invites, and queues.
- **GUIManager:** Builds all interactive GUIs for game selection, settings, stats, and parties.

- **NPCManager:** Integrates with Citizens2 for interactive NPCs.
- **PlaceholderAPIHook:** Exposes dynamic stats to PlaceholderAPI.

Each manager is responsible for a single concern, promoting clean code and ease of testing.

2.3 Flow of Interactions

1. **Player Interacts with GUI/NPC:** GUIManager or NPCManager triggers a request to join a minigame.
2. **GameManager Starts a Game:**
   - Requests an available arena from ArenaManager.
   - Instantiates a new `GameInstance` subclass.
3. **GameInstance Manages Gameplay:**
   - Handles game logic (start, tick, end).
   - Updates stats via StatsManager.
   - Sends updates to GUIManager and PlaceholderAPI.
4. **Game End:**
   - `GameInstance` triggers `GameEndEvent`.
   - ArenaManager resets the arena.
   - StatsManager finalizes player stats.

2.4 Event-Driven Design

The plugin uses an event-driven architecture:

- `GameStartEvent`, `GameEndEvent`, `PlayerJoinGameEvent`, etc.
- Managers and other modules listen for events to coordinate tasks.
- Makes adding new features easier (e.g. hooking quests or achievements).

2.5 Extensibility & Modularity

- **New Minigames:** Extend `GameInstance` and register with GameManager.
- **New Integrations:** Implement hooks (e.g. economy, PlaceholderAPI).
- **New GUIs:** Use GUIManager with YAML-driven templates.
- **New Stats or Leaderboards:** Extend StatsManager or add new modules.

2.6 Summary

This architecture prioritizes separation of concerns, testability, and flexibility. Each manager or module focuses on a single responsibility and communicates through events and interfaces, making the plugin easy to expand and maintain.

3 Modular System Design

3.1 What Does "Modular" Mean in This Context?

In the Minigames Plugin, modular means that each piece of functionality (e.g. game types, stats, parties, NPCs) is isolated into its own package, class, and/or interface, so that developers (and AI) can:

- Add new features without changing existing code.
- Test each module independently.
- Swap out or extend modules (e.g. using different database backends).
- Scale the plugin as more minigames or features are added.

3.2 Key Design Principles

Separation of Concerns: Each module has a single responsibility:

- **GameManager** manages games.
- **ArenaManager** manages arenas.
- **StatsManager** handles player stats.
- ...

Dependency Injection (Optional but recommended): Managers can be passed into minigames rather than hardcoded. Makes testing easier and allows swapping implementations.

Event-Driven Communication: Modules emit and listen to custom events (e.g. `GameStartEvent`, `PartyJoinEvent`) instead of calling each other directly. Promotes loose coupling and easier debugging.

Plugin API for Extensions: Developers (or AI) can register new minigames, GUI panels, or integrations by extending defined interfaces or abstract classes.

3.3 Module Structure & Responsibilities

3.3.1 Managers Layer

- **GameManager:** Registers and starts/stops games. Keeps track of active game instances.
- **ArenaManager:** Loads, allocates, and resets arenas. Integrates with WorldEdit.
- **StatsManager:** Tracks player stats and saves them (YAML or database).
- **PartyManager:** Manages party creation, invites, queues, and rewards.
- **GUIManager:** Handles menu building, dynamic updates, and click handlers.
- **NPCManager:** Integrates Citizens2 to spawn and manage NPCs.
- **PlaceholderAPIHook:** Exposes dynamic placeholders for stats, game states, etc.

3.4 Extending the Plugin

3.4.1 Adding a New Minigame

1. Create a new class that extends `GameInstance`.
2. Implement core methods: `startGame()`, `handleTick()`, `checkWinCondition()`, `endGame()`.
3. Register your game type with GameManager.
4. Define arena requirements in `arenas.yml`.

3.4.2 Adding a New Integration

1. Add a new package under `integrations/`.
2. Implement the required interface (e.g. `EconomyIntegration`).
3. Register the integration in `MinigamesPlugin.java`.

3.4.3 Adding a New GUI

1. Define a new YAML layout in `guis.yml`.
2. Build it in GUIManager using the Inventory API.
3. Link it to a command or NPC click.

3.5 Example: How a New Minigame Might Look

```
// Java code for a new minigame (SpleefGame)
public class SpleefGame extends GameInstance {
    @Override
    public void startGame() {
        // Setup arena, send players messages
    }
    @Override
    public void handleTick() {
        // Check for win conditions
    }
    @Override
    public void endGame() {
        // Update stats, reset arena
    }
}

Then register in GameManager:

// Java code to register the new game type
gameManager.registerGameType("SPLEEF", SpleefGame.class);
```

3.6 Benefits of the Modular System

- **Easier maintenance:** Each module can be updated independently.
- **Simpler debugging:** Narrow down issues to a single module.
- **AI assistant-friendly:** AI can generate or modify modules in isolation.
- **Future proof:** Supports adding integrations (e.g. quests, achievements) without rewriting the plugin.

3.7 Summary

The modular system design empowers developers and AI assistants to extend, test, and maintain the plugin with confidence. Each module has a clearly defined role, and communication is event-driven for maximum flexibility.

4 GameManager

4.1 Purpose

The `GameManager` is responsible for orchestrating the lifecycle of all active minigame instances. It ensures that games start, run, and end properly, coordinating with other managers (like `ArenaManager` and `StatsManager`) as needed.

It also provides a consistent interface for interacting with minigames across the plugin.

4.2 Responsibilities

- **Registering Minigame Types:**
  - Maintains a registry of all available minigame types (e.g. TNT Run, Bed Wars).
  - Allows dynamic addition of new minigame types at runtime.
- **Starting New Game Instances:**
  - Allocates an arena (via `ArenaManager`).
  - Instantiates a new `GameInstance` subclass.
  - Configures the instance with arena settings, player list, etc.
- **Tracking Active Games:**
  - Maintains a collection of all currently running `GameInstance` objects.
  - Supports multiple simultaneous games of the same type.
- **Stopping and Cleaning Up Games:**
  - Listens for `GameEndEvent`s or calls to forcibly stop games.
  - Ensures arenas are reset and stats are updated.
- **Player Management:**
  - Assigns players to games, removes them on quit/disconnect.
  - Interfaces with `PartyManager` to keep parties together.

4.3 Class Design

4.3.1 Class Signature

```
public class GameManager {
    private Map<String, Class<? extends GameInstance>> gameTypeRegistry;
    private List<GameInstance> activeGames;
    private final ArenaManager arenaManager;
    private final StatsManager statsManager;
    private final PartyManager partyManager;

    public void registerGameType(String gameType, Class<? extends GameInstance> gameClass);
    public GameInstance startGame(String gameType, List<Player> players);
    public void stopGame(GameInstance instance);
    public List<GameInstance> getActiveGames();
}
```

4.4 Key Methods

`registerGameType(String gameType, Class<? extends GameInstance> gameClass)` Registers a minigame type with the plugin. Makes it available to the GUI, NPCs, and commands.

`startGame(String gameType, List<Player> players)`

- Validates arena availability.
- Allocates an arena from `ArenaManager`.
- Instantiates the game class using reflection or a factory method.
- Initializes the game with settings and players.
- Adds it to the `activeGames` list.

`stopGame(GameInstance instance)`

- Ends the game (calls `endGame()` on the instance).
- Triggers `GameEndEvent`.
- Removes the game from the `activeGames` list.
- Tells `ArenaManager` to reset the arena.

`getActiveGames()` Returns a list of all currently running games.

4.5 Integration Points

- **ArenaManager:** Allocates and resets arenas.
- **StatsManager:** Updates player stats after each game.
- **PartyManager:** Keeps party members together in the same game instance.
- **GUIManager:** Provides menus to select and join games.
- **PlaceholderAPIHook:** Exposes active game counts and player status.

4.6 Events and Communication

The `GameManager` emits events like:

- `GameStartEvent`
- `GameEndEvent`
- `PlayerJoinGameEvent`
- `PlayerLeaveGameEvent`

These events allow other modules (e.g. quests, leaderboards) to react dynamically.

4.7 Usage Example

```
// Registering a new game type
gameManager.registerGameType("SPLEEF", SpleefGame.class);
// Starting a game with a party
GameInstance spleefGame = gameManager.startGame("SPLEEF", party.getMembers());
```

4.8 Testing and Debugging

- Use unit tests with mock `ArenaManager` and `StatsManager`.
- Validate that start/stop logic correctly registers/unregisters games.
- Check event firing with a test listener.

4.9 Summary

The `GameManager` is the core orchestrator of the plugin, tying together all other systems. It ensures that games are started, tracked, and ended cleanly, while interacting seamlessly with arenas, stats, parties, and GUIs.

5 GameInstance & Game Types

5.1 Purpose

The `GameInstance` class represents a single running instance of a minigame, managing players, game state, and specific logic for that game. Each minigame (TNT Run, Bed Wars, etc.) is implemented as a subclass of `GameInstance`, allowing developers to easily create new minigames by focusing only on the game-specific logic.

5.2 Key Responsibilities

- **Lifecycle Management:** Start, tick/update, and end the game.
- **Player Management:** Handle player joins, leaves, deaths, and respawns. Manage teams (if applicable).
- **Arena Integration:** Request assigned arena from `ArenaManager`. Setup spawn points, resources, and environmental features.
- **Stats Tracking:** Call `StatsManager` to update player stats (wins, kills, etc.). Fire events for PlaceholderAPI and quest integrations.
- **Win Condition Evaluation:** Regularly check for win conditions and trigger game end.

5.3 Class Design

5.3.1 Abstract Base Class

```
public abstract class GameInstance {
    protected final Arena arena;
    protected final List<Player> players;
    protected GameState state;
    protected final GameManager gameManager;
    protected final StatsManager statsManager;

    public GameInstance(Arena arena, List<Player> players) {
        this.arena = arena;
        this.players = players;
        this.state = GameState.WAITING;
    }

    public abstract void startGame();
    public abstract void handleTick();
    public abstract boolean checkWinCondition();
    public abstract void endGame();

    public void removePlayer(Player player);
    public GameState getState();
}
```

5.4 Typical Lifecycle

1. **Initialization:** Arena is assigned and players are teleported. Game state set to `WAITING`.
2. **Start Game:** Setup resources (e.g. shops, loot, objectives). Announce start to players (GUIManager/NPCManager). Change state to `RUNNING`.
3. **Game Tick:** Called by `GameManager` every X ticks. Handles game-specific logic (e.g. TNT decay, border shrink).
4. **Check Win Condition:** Called every tick or after key events (deaths, bed breaks). Calls `endGame()` if win condition met.
5. **End Game:** Fires `GameEndEvent`. Updates `StatsManager`. `ArenaManager` resets the arena.

5.5 Example Game Type: TNT Run

```
public class TntRunGame extends GameInstance {
    @Override
    public void startGame() {
```

```
        state = GameState.RUNNING;
        // Remove support blocks beneath players after delay
    }
    @Override
    public void handleTick() {
        // Check player positions, remove blocks
    }
    @Override
    public boolean checkWinCondition() {
        return players.size() <= 1;
    }
    @Override
    public void endGame() {
        state = GameState.ENDED;
        // Reward winner(s), update stats
        gameManager.stopGame(this);
    }
}
```

5.6 Adding a New Minigame

1. Extend `GameInstance`.
2. Implement all abstract methods (`startGame`, `handleTick`, etc.).
3. Register game type in `GameManager` (e.g. `gameManager.registerGameType("SPLEEF", SpleefGame.class)`).
4. Define arena requirements in `arenas.yml`.
5. Integrate stats tracking and PlaceholderAPI placeholders as needed.

5.7 Key Design Guidelines

- Keep game logic self-contained; don't directly manipulate external modules (use events and managers).
- Use event-driven design for player deaths, win conditions, etc.
- Document game-specific features in code and update design doc if needed.

5.8 Summary

The `GameInstance` system provides a consistent, flexible foundation for implementing new minigames. Each game type focuses on its unique logic while leveraging shared systems (`arena`, `stats`, `parties`) for consistency and modularity.

6 Game Modes

Purpose
Introduce different game styles or modes to enhance replayability and customization.

Overview
Each minigame can define multiple game modes (e.g. Classic, Hardcore, Teams) to tweak gameplay parameters without rewriting core logic.

Technical Design

1. GameMode Enum

[language=Java] public enum GameMode CLASSIC, HARDCORE, TEAMS, TIMED, CUSTOM

2. GameInstance Enhancements

[language=Java] public abstract class GameInstance protected GameMode gameMode;
public GameInstance(GameMode mode) this.gameMode = mode;
public abstract void start();

3. GameManager Enhancements

[language=Java] public GameInstance createGame(String gameType, GameMode mode) switch (gameType) case "TNT_{RUN}" : $return new TNTRunGame(mode); case "BED_{WARS}$" : $return new BedWarsGame(mode); //...$

4. Configuration Enhancements

[language=yaml] minigames: $\mathrm{tnt}_{run} : modes : -CLASSIC-HARDCORE rewards : CLASSIC : win : 100 HARDCORE : win : 200$

5. GUI Enhancements

Add a sub-menu in the game selection GUI to let players choose their desired game mode:

- Clicking a minigame (e.g. TNT Run) shows available modes (Classic, Hardcore).
- Selecting a mode starts that game instance.

Benefits

- Adds variety and replayability.

- Makes it easy to tweak gameplay styles.
- Keeps the code modular and scalable.

7 ArenaManager

7.1 Purpose

The `ArenaManager` module is responsible for managing all arenas available to minigames. It ensures that each minigame instance has a dedicated, conflict-free region to operate in. It also handles arena resets, integrates with WorldEdit (for schematic pasting), and defines spawn points, chest locations, and other arena-specific details.

7.2 Responsibilities

- **Arena Definitions & Metadata:** Load arena configurations from `arenas.yml` (or a database). Store metadata: name, world, region boundaries, spawn points, loot chests, NPC locations, etc.
- **Arena Allocation:** Provide an available arena to a new `GameInstance`. Lock the arena during use to prevent conflicts.
- **Arena Resetting:** Reset arena blocks after each game. Use WorldEdit schematics, block snapshots, or custom reset algorithms.
- **Dynamic Integration:** Integrate with WorldEdit API for advanced region management. Fire events (e.g. `ArenaResetEvent`) for hooks like dynamic quest objectives or external plugin integrations.

7.3 Class Design

7.3.1 Class Signature

```
public class ArenaManager {
    private final Map<String, List<Arena>> arenasByType;
    private final WorldEditIntegration worldEditIntegration;

    public void loadArenas();
    public Arena allocateArena(String gameType);
    public void resetArena(Arena arena);
    public List<Arena> getAvailableArenas(String gameType);
}
```

7.4 Arena Definition

`arenas.yml` Example:

```
arenas:
  tnt_run:
    - id: tnt_run_1
      world: minigames_world
      region:
        min: { x: 100, y: 60, z: 100 }
        max: { x: 150, y: 90, z: 150 }
      spawns:
        - { x: 125, y: 65, z: 125 }
      schematic: tnt_run_1.schem
  bed_wars:
    - id: bed_wars_1
      world: minigames_world
      region:
        min: { x: 200, y: 60, z: 200 }
        max: { x: 250, y: 90, z: 250 }
      team_spawns:
        red: { x: 210, y: 65, z: 210 }
        blue: { x: 240, y: 65, z: 240 }
      schematic: bed_wars_1.schem
```

7.5 Arena Lifecycle

1. **Load Arenas:** On plugin startup, load all arenas from `arenas.yml`. Validate region data (WorldEdit integration can help here).
2. **Allocate Arena:** When `GameManager` requests an arena, select an available one for that game type. Lock the arena to prevent concurrent use.
3. **Reset Arena:** On game end, call `resetArena()`. WorldEdit pastes schematic or resets blocks using snapshots. Fire `ArenaResetEvent` for integrations.
4. **Release Arena:** Unlock the arena for future use.

7.6 WorldEdit Integration

- Use WorldEdit API to paste schematics.
- Detect region overlaps to prevent multiple games in the same space.
- Optionally support dynamic arena generation in the future.

7.7 Events and Hooks

- `ArenaAllocatedEvent` fires when an arena is assigned to a game.
- `ArenaResetEvent` fires when an arena reset completes.

These events allow external plugins (like quests) to react dynamically.

7.8 Usage Example

```
// Allocate an arena for a TNT Run game
Arena arena = arenaManager.allocateArena("tnt_run");
GameInstance tntRunGame = new TntRunGame(arena, players);
// Reset arena on game end
arenaManager.resetArena(arena);
```

7.9 Testing and Debugging

- Validate region boundaries and coordinate conversion.
- Ensure arenas don't overlap with others in use.
- Test schematic resets with WorldEdit in a test environment.

7.10 Summary

The `ArenaManager` module is crucial for ensuring each game has its own dedicated, resettable region. It integrates tightly with WorldEdit and provides a consistent API for managing arenas across all minigames.

8 StatsManager

8.1 Purpose

The `StatsManager` module is responsible for tracking, storing, and exposing player statistics for all minigames. It allows server administrators and players to view progress and achievements, and integrates with PlaceholderAPI to display dynamic stats on scoreboards, menus, and holograms.

8.2 Responsibilities

- **Track Player Statistics:** Store core stats: wins, losses, kills, deaths, and minigame-specific metrics. Track per-game stats as well as global stats.
- **Data Persistence:** Store stats in YAML files or a database (SQLite/MySQL). Load stats on player join, save on quit and at regular intervals.
- **Stats Queries:** Provide methods for getting player stats by player UUID or name. Aggregate stats for leaderboards.
- **PlaceholderAPI Integration:** Expose dynamic stats like `%minigame_tnt_run_wins%`. Enable scoreboard or GUI integration.

8.3 Class Design

8.3.1 Class Signature

```
public class StatsManager {
    private final Map<UUID, PlayerStats> statsMap;
    private final StorageAdapter storageAdapter;

    public void loadStats(Player player);
    public void saveStats(Player player);
    public PlayerStats getStats(Player player);
    public void incrementStat(Player player, String statType, int amount);
}
```

8.3.2 PlayerStats Class

```
public class PlayerStats {
    private final UUID playerId;
    private final Map<String, Integer> globalStats;
    private final Map<String, Map<String, Integer>> gameStats;

    public void incrementGlobalStat(String statType, int amount);
    public void incrementGameStat(String gameType, String statType, int amount);
    public int getGlobalStat(String statType);
    public int getGameStat(String gameType, String statType);
}
```

8.4 Data Storage

- **StorageAdapter** interface with implementations:
  - **YamlStorageAdapter:** uses player-specific YAML files for quick deployment.
  - **DatabaseStorageAdapter:** uses SQLite or MySQL for scalable, server-friendly stats.

8.5 Typical Workflow

1. **Player Join:** Load stats from storage. Populate in-memory `PlayerStats` object.
2. **During Gameplay:** Increment stats on relevant game events (kills, wins). `StatsManager` handles in-memory tracking.
3. **Game End or Player Quit:** Save stats to storage.
4. **PlaceholderAPI:** `PlaceholderAPIHook` queries `StatsManager` for dynamic values.

8.6 PlaceholderAPI Integration

- Dynamic Placeholders:
  - %minigame_<gameType>_<statType>% e.g. %minigame_tnt_run_wins%, %minigame_bed_wars_kills%.
  - %minigame_total_playtime%.
  - %minigame_global_wins%.
- Refresh Interval: Configurable (default every 10 seconds).
- Implementation: Uses `PlaceholderExpansion` to hook into PlaceholderAPI.

8.7 Example Usage

```
// Incrementing a stat
statsManager.incrementStat(player, "tnt_run_wins", 1);
// Getting stats for display
int wins = statsManager.getStats(player).getGameStat("tnt_run", "wins");
```

8.8 Testing and Debugging

- Test stat persistence across server restarts.
- Validate PlaceholderAPI placeholders.
- Use mock `StorageAdapter` for unit tests.

8.9 Summary

The `StatsManager` ensures accurate and reliable stat tracking for players across all minigames. It integrates with PlaceholderAPI, supports multiple storage backends, and provides APIs for easy access to stats—making it essential for leaderboards, rewards, and player engagement.

9 PartyManager

9.1 Purpose

The `PartyManager` module provides a system for players to form temporary social groups (parties) so they can play minigames together. Parties improve the social experience by ensuring friends stick together, share rewards, and coordinate strategies.

9.2 Responsibilities

- **Party Creation & Management:** Create, join, leave, and disband parties. Assign a party leader with special permissions.
- **Invites & Acceptance:** Manage party invitations (invite, accept, deny, cancel). Time-limited invites to prevent spam.
- **Party Queuing:** Keep party members in the same minigame instance. Guarantee teams (e.g. Bed Wars) are assigned together.
- **Rewards Sharing:** Allow parties to share or split rewards (coins, XP). Configurable in `config.yml` (e.g. shared or individual).
- **Party Chat:** Enable a private chat channel for party members.

9.3 Class Design

9.3.1 PartyManager Class

```
public class PartyManager {
    private final Map<UUID, Party> playerPartyMap;
    private final List<Party> activeParties;

    public Party createParty(Player leader);
    public void invitePlayer(Party party, Player invitee);
    public void joinParty(Player player, Party party);
    public void leaveParty(Player player);
    public void disbandParty(Party party);
    public Party getParty(Player player);
}
```

9.3.2 Party Class

```
public class Party {
    private final UUID leader;
    private final Set<UUID> members;
    private final Map<String, Object> settings;

    public void addMember(UUID playerId);
    public void removeMember(UUID playerId);
    public boolean isLeader(UUID playerId);
    public Set<UUID> getMembers();
}
```

9.4 Party Lifecycle

1. **Player Creates a Party:** Player uses `/party create` or GUI. Player is assigned as party leader.

2. **Invite Players:** `/party invite <player>`. Target player gets a message and can `/party accept`.
3. **Join Games Together:** `PartyManager` ensures all members are queued into the same minigame instance. `PartyManager` interacts with `GameManager` to assign teams where needed.
4. **Game Ends:** Rewards are shared or distributed based on config.
5. **Party Disbanded:** Party ends if leader leaves or uses `/party disband`.

9.5 GUI Integration

- Party GUI shows:
  - Members with online/offline status.
  - Invite/kick buttons (for leaders).
  - Toggle shared rewards (configurable).
  - Leave/disband party button.
- Integrated with `GUIManager`.

9.6 Configuration Options

Example in `config.yml`:

```
party:
  shared_rewards: true
  max_members: 5
  invite_timeout: 60 # seconds
```

9.7 Events & Hooks

- `PartyJoinEvent`
- `PartyLeaveEvent`
- `PartyDisbandEvent`
- `PartyQueueEvent`

Other modules (e.g. `StatsManager`) can listen to these events to apply bonuses or adjust stats.

9.8 Example Usage

```
Party party = partyManager.createParty(leader);
partyManager.invitePlayer(party, invitee);
```

9.9 Testing & Debugging

- Validate invite and join flows.
- Ensure parties queue together into minigames.
- Test reward sharing logic.

9.10 Summary

The `PartyManager` module fosters community play, teamwork, and strategy, enriching the player experience by allowing friends to stick together across all minigames. Its design integrates tightly with `GameManager`, `StatsManager`, and `GUIManager` for a seamless experience.

10 GUIManager

10.1 Purpose

The `GUIManager` module is responsible for creating and managing all in-game menus that players and admins interact with. These menus are built using the Bukkit Inventory API and/or GUI libraries, and they make it easy to select minigames, manage parties, view stats, and configure settings.

10.2 Responsibilities

- **Build Dynamic GUIs:** Construct menus like:
  - Minigame selection (browse and join games)
  - Party management (view members, invite/kick, settings)
  - Stats display (player wins, kills, leaderboards)
  - Settings menus (global and arena-specific)
- **Handle Click Events:** Listen for player clicks in GUI menus. Route actions to the appropriate manager (`GameManager`, `PartyManager`, etc.).
- **Support PlaceholderAPI:** Display dynamic data using placeholders (e.g. game status, player stats).
- **Integrate with Config:** Allow customization of menu layouts via `guis.yml`. Use icons, names, lores, and dynamic placeholders in GUI items.

10.3 Class Design

10.3.1 GUIManager Class

```
public class GUIManager {
    private final MinigamesPlugin plugin;

    public void openGameSelection(Player player);
    public void openPartyMenu(Player player);
    public void openStatsMenu(Player player);
    public void openSettingsMenu(Player player, String gameType);
}
```

10.4 GUI Types

10.4.1 Game Selection Menu

- Shows available minigames.
- Displays:
  - Game name and icon.
  - Current player count.
  - Status (waiting, running).
- Click to join or spectate.

10.4.2 Party Management Menu

- Shows party members.
- Buttons to:
  - Invite, kick, promote members.
  - Toggle party chat.
  - Queue party into a minigame.

10.4.3 Statistics Menu

- Shows player stats for all minigames.
- Uses PlaceholderAPI for dynamic data.
- Includes leaderboards.

10.4.4 Settings Menu

- Allows admins to adjust global and arena-specific settings:
  - Enable/disable minigames.
  - Set max instances.
  - Configure rewards.

10.5 Integration Points

- **GameManager:** Joins games when a player clicks a minigame item.
- **PartyManager:** Handles party invites and kicks via GUI.
- **StatsManager:** Pulls stats for display in menus.
- **PlaceholderAPI:** Displays dynamic stats and game data in item lores.

10.6 Configuration Example

`guis.yml`:

```
game_selection:
  rows: 3
  title: "Select a Minigame"
  items:
    - type: TNT_RUN
      icon: TNT
      display_name: "&cTNT Run"
      lore:
        - "Players: %minigame_tnt_run_players%"
        - "Status: %minigame_tnt_run_status%"
    - type: BED_WARS
      icon: BED
      display_name: "&bBed Wars"
      lore:
        - "Teams: 4"
        - "Beds remaining: %minigame_bed_wars_beds%"
```

10.7 Example Usage

```
// Open game selection menu
guiManager.openGameSelection(player);
```

10.8 Testing & Debugging

- Test GUI click handlers and menu refresh logic.
- Validate PlaceholderAPI placeholders render properly.
- Test performance under high player load.

10.9 Summary

The `GUIManager` module is the user interface backbone of the plugin, making it easy for players to interact with all the plugin's features. It integrates closely with other managers to create a seamless experience and uses PlaceholderAPI for dynamic, real-time data.

11 NPCManager

11.1 Purpose

The `NPCManager` module integrates with the Citizens2 plugin to spawn and manage NPCs in the Minecraft world. These NPCs serve as interactive join points, display game information, and provide a more immersive way for players to enter minigames or access party features.

11.2 Responsibilities

- **NPC Spawning & Configuration:** Spawn NPCs at predefined locations defined in `arenas.yml` or `npcs.yml`. Assign skins, names, and holograms (optional).
- **Interactive Actions:** On NPC click:
  - Join a minigame or open a GUI menu.
  - Show game status or player count.
  - Provide quick access to party management.
- **Dynamic Data Display:** Use PlaceholderAPI or direct plugin calls to update NPC names or holograms dynamically (e.g. TNT Run: 12 Players).
- **NPC Lifecycle Management:** Remove NPCs on plugin disable. Update NPCs on plugin reload or arena resets.

11.3 Class Design

11.3.1 NPCManager Class

```
public class NPCManager {
    private final MinigamesPlugin plugin;
    private final Map<Integer, NPC> npcMap; // Citizens2 NPCs

    public void loadNPCs();
    public void spawnNPC(String gameType, Location location);
    public void removeNPC(int npcId);
    public void handleClick(Player player, NPC npc);
}
```

11.4 NPC Definition Example

`npcs.yml`:

```
npcs:
  - id: 1
    game_type: "tnt_run"
    location: { world: "minigames_world", x: 100.5, y: 65, z: 100.5, yaw: 0, pitch: 0 }
    skin: "Notch"
    name: "&cTNT Run"
    hologram:
      lines:
        - "&ePlayers: %minigame_tnt_run_players%"
        - "&aClick to Join!"
```

11.5 Typical NPC Flow

1. **Load NPCs:** On plugin startup or reload, load NPCs from `npcs.yml`. Use Citizens2 API to spawn NPCs at defined locations.
2. **Player Clicks NPC:** `NPCManager` detects click event from Citizens2. Calls `handleClick()` to:
   - Join game via `GameManager`.
   - Open game selection GUI (optional).
   - Show game status (PlaceholderAPI).
3. **Dynamic Updates:** `NPCManager` periodically updates holograms and names. Uses PlaceholderAPI to show real-time data.
4. **Plugin Shutdown:** Remove all NPCs cleanly to prevent duplication on restart.

11.6 Integration Points

- **GameManager:** Starts game when a player interacts with an NPC.
- **PlaceholderAPI:** Provides dynamic data for NPC names and holograms.
- **GUIManager:** Opens menus if NPC is configured as a menu link.

11.7 Example Usage

```
npcManager.spawnNPC("tnt_run", new Location(world, 100.5, 65, 100.5));
```

11.8 Testing & Debugging

- Validate NPC clicks trigger correct actions.
- Ensure dynamic data updates properly in holograms.
- Test plugin reloads to ensure NPCs persist and update correctly.

11.9 Summary

The `NPCManager` module is key for creating an immersive and interactive player experience. By integrating with Citizens2 and PlaceholderAPI, it allows dynamic, real-time NPCs that guide players into minigames and enhance engagement.

12 PlaceholderAPI Integration

12.1 Purpose

The PlaceholderAPI Integration module allows other plugins (like scoreboard or hologram plugins) to access real-time statistics and game data from the Minigames Plugin. This enhances the player experience by displaying dynamic stats and game info seamlessly across the server.

12.2 Responsibilities

- **Expose Dynamic Placeholders:** Provide placeholders for:
  - Player stats (wins, kills, deaths).
  - Active game counts.
  - Arena statuses (enabled, in-use).
  - Party stats (size, leader).
- **Dynamic Data Refresh:** Update placeholder values on a configurable interval (e.g. every 5 seconds).
- **Register Placeholders:** Hook into PlaceholderAPI on plugin enable. Unregister on plugin disable.

12.3 Class Design

12.3.1 PlaceholderAPIHook Class

```
public class PlaceholderAPIHook extends PlaceholderExpansion {
    private final StatsManager statsManager;
    private final GameManager gameManager;
    private final PartyManager partyManager;

    @Override
    public String onPlaceholderRequest(Player player, String identifier) {
        // Example: minigame_tnt_run_wins
        if (identifier.startsWith("minigame_")) {
            return handleMinigamePlaceholder(player, identifier);
        }
        return "";
    }

    private String handleMinigamePlaceholder(Player player, String identifier) {
        // Split identifier and fetch stats
        // E.g., identifier = minigame_tnt_run_wins
    }
}
```

12.4 Placeholder Structure

- **Per-Game Stats:** %minigame_<gameType>_<statType>% Example: %minigame_tnt_run_wins%
- **Global Stats:** %minigame_global_<statType>%
- **Active Games:** %minigame_<gameType>_active_games%
- **Arena Status:** %minigame_<gameType>_<arenaId>_status%
- **Party Stats:** %minigame_party_<statType>% Example: %minigame_party_size%

12.5 Typical Usage Scenarios

- **Scoreboards:** Display players' wins, kills, or deaths in a minigame.
- **NPC Holograms:** Show real-time player counts, game statuses, or party sizes.
- **GUIs:** Use placeholders in lores to show stats or status dynamically.

12.6 Configuration Example

In config.yml:

```
placeholders:
  refresh_interval: 5 # seconds
```

12.7 Integration Points

- **StatsManager:** Supplies player-specific stats.
- **GameManager:** Supplies active game counts and statuses.
- **PartyManager:** Supplies party sizes and leader info.

12.8 Example Usage

```
// Placeholder usage in a scoreboard plugin
%minigame_tnt_run_wins%
```

12.9 Testing & Debugging

- Validate that placeholders update at the correct interval.
- Test placeholder rendering in scoreboards, GUIs, and holograms.
- Check behavior when a stat or game type is missing.

12.10 Summary

The PlaceholderAPI Integration module bridges the Minigames Plugin with the rest of the server ecosystem, enabling dynamic, real-time stats to enhance scoreboards, GUIs, NPCs, and other features. It empowers admins to create a rich, interactive experience for players.

13 Configuration Files

13.1 Purpose

The Configuration System ensures that server administrators and developers can easily customize and fine-tune the plugin's behavior without modifying the code. It also supports dynamic reloading to allow real-time changes during gameplay, reducing downtime and making administration easier.

13.2 Core Configuration Files

`config.yml`**:** Global plugin settings (minigames, rewards, PlaceholderAPI, economy).

`arenas.yml`**:** Defines arena regions, spawn points, chests, and NPC locations.

`guis.yml`**:** Menu layouts and icons.

`npcs.yml`**:** NPC definitions, locations, and skins.

`stats.yml` **(optional):** Player stats for non-database storage (YAML fallback).

13.3 `config.yml`

Purpose: Controls plugin-wide settings like:

- Minigame enabling/disabling.
- Max instances.
- Economy integration.
- PlaceholderAPI refresh rates.
- Party settings.

Example:

```
minigames:
  tnt_run:
    enabled: true
    max_instances: 5
    rewards:
      win: 100
      participation: 10
  bed_wars:
    enabled: true
    max_instances: 3
economy:
  enabled: true
  vault: true
placeholderapi:
  refresh_interval: 5
party:
  shared_rewards: true
  max_members: 5
  invite_timeout: 60
```

13.4 `arenas.yml`

Purpose: Defines arena boundaries, spawn points, team spawns, and schematic files. Used by `ArenaManager` to allocate and reset arenas.

Example:

```
arenas:
  tnt_run:
    - id: "tnt_run_1"
      world: "minigames_world"
      region:
```

```
      min: { x: 100, y: 60, z: 100 }
      max: { x: 150, y: 90, z: 150 }
    spawns:
      - { x: 125, y: 65, z: 125 }
    schematic: "tnt_run_1.schem"
bed_wars:
  - id: "bed_wars_1"
    world: "minigames_world"
    region:
      min: { x: 200, y: 60, z: 200 }
      max: { x: 250, y: 90, z: 250 }
    team_spawns:
      red: { x: 210, y: 65, z: 210 }
      blue: { x: 240, y: 65, z: 240 }
    schematic: "bed_wars_1.schem"
```

13.5 guis.yml

Purpose: Define layout, rows, titles, icons, display names, and lores for GUIs. `GUIManager` uses this file to render menus dynamically.

Example:

```
game_selection:
  rows: 3
  title: "Select a Minigame"
  items:
    - type: TNT_RUN
      icon: TNT
      display_name: "&cTNT Run"
      lore:
        - "Players: %minigame_tnt_run_players%"
        - "Status: %minigame_tnt_run_status%"
    - type: BED_WARS
      icon: BED
      display_name: "&bBed Wars"
      lore:
        - "Teams: 4"
        - "Beds remaining: %minigame_bed_wars_beds%"
```

13.6 npcs.yml

Purpose: Define NPCs with Citizens2. Set game type, skin, name, location, and optional holograms.

Example:

```
npcs:
  - id: 1
    game_type: "tnt_run"
    location: { world: "minigames_world", x: 100.5, y: 65, z: 100.5, yaw: 0, pitch: 0 }
    skin: "Notch"
    name: "&cTNT Run"
    hologram:
      lines:
        - "&ePlayers: %minigame_tnt_run_players%"
        - "&aClick to Join!"
```

13.7 `stats.yml` (optional)

Only used for YAML-based stats storage when no database is configured. Contains stats per player UUID.

13.8 Dynamic Configuration & Reloading

Hot Reload: Admins can use `/minigames reload` to apply changes without server restart. Plugin reloads all YAML files and updates dynamic content (e.g. GUIs, NPCs).

Dynamic Updates: `GUIManager` and `PlaceholderAPIHook` listen for configuration changes and update menus/placeholders live.

13.9 Summary

The Configuration System gives admins and developers full control over the plugin's functionality, enabling them to customize everything from minigame settings to GUI layouts. The system is designed to be dynamic, flexible, and easy to use, ensuring smooth integration with the rest of the plugin ecosystem.

14 Plugin Integrations

14.1 Purpose

The Plugin Integrations module connects the Minigames Plugin to the wider Minecraft plugin ecosystem. By hooking into external plugins like Citizens2, WorldEdit, PlaceholderAPI, and Vault, this plugin enhances functionality, automates processes, and provides a polished, professional experience.

14.2 Supported Integrations

14.2.1 Citizens2

Purpose: NPC integration for interactive game join points and status displays.

Integration Points:
- `NPCManager` spawns NPCs at configured locations.
- Supports dynamic names, skins, and holograms.
- Click actions to join games or open menus.

14.2.2 WorldEdit

Purpose: Arena region selection, schematic pasting, and arena resets.

Integration Points:
- `ArenaManager` uses WorldEdit API to paste schematics.
- Supports schematic-based resets for complex arena builds.
- Optional custom wand for defining regions in-game.

14.2.3 PlaceholderAPI

Purpose: Dynamic, real-time stats and game info for scoreboards, holograms, and GUIs.

Integration Points:
- `PlaceholderAPIHook` registers custom placeholders.
- Used by `GUIManager` and `NPCManager` for dynamic displays.
- Integrates with external plugins like DeluxeMenus or hologram plugins.

14.2.4 Vault

Purpose: Economy integration for rewards, shops, and minigame participation fees.

Integration Points:
- `EconomyManager` (if implemented) interfaces with Vault to deposit rewards.
- Rewards can be set per minigame in `config.yml`.

14.3 Integration Architecture

- **Modular Adapters:** Each integration is built as a modular adapter within the plugin's `integrations` package. Example:
  - `VaultIntegration.java`
  - `WorldEditIntegration.java`
  - `PlaceholderAPIHook.java`

  These adapters abstract away the specific plugin APIs, making testing and future replacements easier.
- **Dynamic Loading:** Plugin uses `Bukkit.getPluginManager().getPlugin()` to check if the external plugin is installed. If an integration is missing, plugin disables that feature gracefully.

- **Event-Driven Hooks:** Plugin fires events (e.g. `GameStartEvent`, `GameEndEvent`) that integrations can listen to. External plugins or integrations can react to these events dynamically.

14.4 Example Integration: Citizens2

```
if (Bukkit.getPluginManager().getPlugin("Citizens") != null) {
    // Register NPC click events
    // Spawn NPCs with Citizens2 API
}
```

14.5 Example Integration: WorldEdit

```
if (Bukkit.getPluginManager().getPlugin("WorldEdit") != null) {
    // Use WorldEdit API to paste schematics
}
```

14.6 Example Integration: PlaceholderAPI

```
if (Bukkit.getPluginManager().getPlugin("PlaceholderAPI") != null) {
    new PlaceholderAPIHook(this).register();
}
```

14.7 Example Integration: Vault

```
if (Bukkit.getPluginManager().getPlugin("Vault") != null) {
    // Setup Vault economy
}
```

14.8 Testing & Debugging

- Validate integrations are enabled/disabled based on external plugin availability.
- Use log messages to alert admins when integrations are missing or misconfigured.
- Unit test integrations with mock plugin dependencies.

14.9 Summary

The Plugin Integrations system connects the Minigames Plugin to the broader Minecraft plugin ecosystem, enhancing functionality and creating a seamless, feature-rich experience for players. Its modular design ensures that each integration is easy to maintain, test, and expand.

15 Commands System

15.1 Purpose

The Commands System enables players and server admins to interact with the Minigames Plugin through textual commands in the Minecraft chat. These commands allow for managing games, parties, stats, and plugin configuration—providing powerful administrative and player controls.

15.2 Responsibilities

- **Command Registration:** Register plugin commands in `plugin.yml` and in `MinigamesPlugin.java`. Use a central command manager to manage subcommands.
- **Permissions Handling:** Assign permissions for each command and subcommand. Use `minigames.admin`, `minigames.play`, `minigames.party`, etc.
- **Dynamic Help & Tab Completion:** Provide contextual help for players and admins.
- **Command Routing:** Route commands to appropriate managers (`GameManager`, `PartyManager`, etc.).

15.3 Command Overview

`/minigames` Main command, shows help menu.

`/minigames join <game>` Join a minigame by type.

`/minigames leave` Leave current game.

`/minigames stats <player>` View player stats.

`/minigames reload` Reload configuration files.

`/party` Party management (create, invite, leave, disband).

`/arena` Arena management (create, edit, save).

15.4 Command Structure

15.4.1 CommandManager Class

```
public class CommandManager implements CommandExecutor, TabCompleter {
    private final MinigamesPlugin plugin;

    public boolean onCommand(CommandSender sender, Command command, String label, String[] a
    public List<String> onTabComplete(CommandSender sender, Command command, String alias, S
}
```

15.4.2 Subcommands

Use a subcommand pattern to route commands:

- `/minigames join TNT_RUN`
- `/minigames leave`
- `/party invite <player>`
- `/arena create <gameType> <arenaId>`

15.5 Permissions Example

In `plugin.yml`:

```
permissions:
  minigames.admin:
    description: "Access to all minigames admin commands."
    default: op
  minigames.play:
    description: "Allows players to join and play minigames."
    default: true
  minigames.party:
    description: "Allows players to create and manage parties."
    default: true
```

15.6 Example Command Flow

1. Player types: `/minigames join tnt_run`
2. `CommandManager` parses command, validates permissions.
3. `CommandManager` calls `GameManager.startGame()` for TNT Run.
4. `GameManager` allocates an arena and starts the game.
5. Player is teleported to the arena.

15.7 Tab Completion

Suggest minigame types, arena names, player names, etc. Improves usability for players and admins.

15.8 Dynamic Help Menu

Provide help text when no subcommand is entered: `/minigames` shows:

- `/minigames join <game>` Join a minigame.
- `/minigames stats <player>` View stats.
- `/minigames reload` Reload config.

15.9 Integration Points

- **GameManager:** Start/stop games.
- **PartyManager:** Party creation and management.
- **ArenaManager:** Create/edit arenas via commands.
- **StatsManager:** View and reset stats.

15.10 Testing & Debugging

- Validate permission checks and feedback messages.
- Test subcommands and tab completion with multiple players.
- Confirm dynamic help menus display as expected.

15.11 Summary

The Commands System is the backbone of the plugin's player and admin interaction. It ensures easy access to all features, supports permissions, and integrates seamlessly with the plugin's managers—empowering both players and administrators.

16 Events & Listeners

16.1 Purpose

The Events & Listeners system is a core part of the plugin's architecture, enabling loose coupling and modularity. It allows different modules (like `GameManager`, `StatsManager`, `PartyManager`) and external plugins to react to key moments in the game lifecycle—such as when a game starts, ends, or when players join or leave.

16.2 Responsibilities

- **Define Custom Events:** Create events like `GameStartEvent`, `GameEndEvent`, `PlayerJoinGameEvent`, etc.
- **Fire Events at Key Moments:** `GameManager` fires `GameStartEvent` when a game begins. `GameInstance` fires `GameEndEvent` when a game ends. `PartyManager` fires `PartyJoinEvent` and `PartyDisbandEvent` as needed.
- **Provide Listeners:** Register event listeners to respond to events internally (e.g. `StatsManager` updating stats on `GameEndEvent`). Allow external plugins (like quests or leaderboards) to register their own listeners.

16.3 Event Definitions

16.3.1 Example: `GameStartEvent`

```
public class GameStartEvent extends Event {
    private final GameInstance gameInstance;

    public GameStartEvent(GameInstance gameInstance) {
        this.gameInstance = gameInstance;
    }

    public GameInstance getGameInstance() {
        return gameInstance;
    }

    @Override
    public HandlerList getHandlers() {
        return handlers;
    }
}
```

16.3.2 Example: `GameEndEvent`

```
public class GameEndEvent extends Event {
    private final GameInstance gameInstance;

    public GameEndEvent(GameInstance gameInstance) {
        this.gameInstance = gameInstance;
    }

    public GameInstance getGameInstance() {
        return gameInstance;
    }

    @Override
    public HandlerList getHandlers() {
        return handlers;
    }
}
```

16.4 Key Events

`GameStartEvent` When a new game starts.

`GameEndEvent` When a game ends.

`PlayerJoinGameEvent` When a player joins a game.

`PlayerLeaveGameEvent` When a player leaves a game.

`PartyJoinEvent` When a player joins a party.

`PartyDisbandEvent` When a party is disbanded.

`ArenaResetEvent` When an arena is reset after a game.

16.5 Listener Registration

16.5.1 Plugin Example

```
public class StatsListener implements Listener {
    private final StatsManager statsManager;

    @EventHandler
    public void onGameEnd(GameEndEvent event) {
        statsManager.updateStats(event.getGameInstance());
    }
}
```

16.5.2 Registration in Main Plugin

```
getServer().getPluginManager().registerEvents(new StatsListener(statsManager), this);
```

16.6 External Plugin Integration

Other plugins can listen to events to:

- Award quest progress.
- Update leaderboards.
- Send Discord notifications.

16.7 Benefits of Event-Driven Design

- **Loose Coupling:** Modules don't depend directly on each other.
- **Extendibility:** New features can be added without modifying existing modules.
- **Integration Friendly:** External plugins can hook into events for seamless synergy.

16.8 Testing & Debugging

- Validate that all events fire at the correct times.
- Write unit tests to simulate event triggers.
- Use debug logging to confirm event data (e.g. game instance details).

16.9 Summary

The Events & Listeners system is a cornerstone of the plugin's modular, extensible design. It enables each module to react to key moments while remaining decoupled, and it empowers external plugins to integrate tightly with the minigames system.

17 Development Workflow

17.1 Purpose

This section outlines the recommended workflow and development practices for contributing to the Minigames Plugin. It aims to make the process seamless for both human developers and AI contributors, ensuring consistent quality, maintainability, and collaboration.

17.2 Best Practices

Use a Version Control System (Git):
- Use a service like GitHub or GitLab.
- Commit changes in small, meaningful chunks with clear messages.

Branching Strategy:
- `main` branch: stable, production-ready code.
- `develop` branch: active development.
- Feature branches: `feature/feature-name` for new features.
- Bugfix branches: `bugfix/issue-name` for fixes.

Code Reviews & Pull Requests:
- Submit pull requests for all feature and bugfix branches.
- Use code review tools to catch issues early.

17.3 Tools & IDE Setup

- **Recommended IDE:** IntelliJ IDEA (supports Java and Minecraft plugin development).
- **Dependencies:**
  - Spigot/Paper API.
  - Citizens2 API (optional).
  - WorldEdit API (optional).
  - PlaceholderAPI API (optional).
  - Vault API (optional).
- **Build System:** Maven or Gradle (recommended) to handle dependencies and builds. Allows integration testing with Spigot/Paper API.

17.4 Development Process

1. **Select a Task:** Choose a task (e.g. add a new minigame, create a GUI, implement an event). Check for open issues or feature requests in the repository.
2. **Create a Branch:** Use a feature or bugfix branch for the task. Example: `feature/add-bed-wars`.
3. **Write Code & Tests:** Follow modular design (each manager or minigame in its own package). Write unit tests and/or integration tests as appropriate.
4. **Document Code:** Use JavaDocs for public classes and methods. Update the design document if architectural changes are made.
5. **Submit a Pull Request:** Merge to `develop` after review and testing.

17.5 Testing & Debugging

Unit Testing:
- Use mocking frameworks (e.g. Mockito) to isolate modules.

Integration Testing:
- Use Spigot or Paper test servers.
- Validate arena allocation, stats tracking, GUI functionality, etc.

Debugging:
- Use detailed logging (plugin logger).
- Enable debug mode in `config.yml` to get verbose logs.

17.6 Documentation

- **Design Document:** Update this document as new modules or integrations are added.
- **Code Comments:** Use JavaDocs for classes, methods, and important logic.
- **ReadMe.md:** Keep the plugin's repository README updated with usage instructions and installation guides.

17.7 Collaboration & Communication

- Use issues and discussions in the repository for planning and support.
- Clearly label issues as bug, enhancement, question, etc.

17.8 AI Developer Integration

Guidelines for AI Contributions:

- AI assistants should follow the same branching and documentation practices.
- Human reviewers should verify AI-generated code.
- AI can help with:
  - Generating starter classes (`GameInstance`, `GUIManager`).
  - Writing test cases.
  - Drafting documentation.

17.9 Summary

A clear, consistent workflow ensures high-quality contributions and a smooth onboarding process. By following these guidelines, both human and AI developers can effectively collaborate to build, extend, and maintain the Minigames Plugin.

18 Testing & Debugging

18.1 Purpose

The Testing & Debugging section ensures the Minigames Plugin remains reliable, maintainable, and bug-free throughout its lifecycle. It outlines the recommended approaches for testing individual modules, minigame logic, and integrations with other plugins. This section also covers debugging tools and practices that streamline development and improve overall stability.

18.2 Testing Strategies

18.2.1 Unit Testing

- **Purpose:** Test individual classes (e.g. `GameInstance`, `StatsManager`) in isolation.
- **Tools:** JUnit + Mockito (recommended).
- **Guidelines:**
  - Mock dependencies (e.g. `ArenaManager`) to isolate logic.
  - Validate expected outputs and side effects (e.g. state changes).

18.2.2 Integration Testing

- **Purpose:** Test interactions between modules (e.g. `GameManager`, `PartyManager`).
- **Tools:** PaperTest or SpigotTest frameworks, or set up a local Spigot/Paper test server.

- **Guidelines:**
  - Use real plugin interactions (player joins, arena allocation).
  - Validate game lifecycles and event triggers.

18.2.3 System Testing

- **Purpose:** Test the plugin in a real server environment.
- **Tools:** Local Spigot/Paper server with plugins installed.
- **Guidelines:**
  - Test complete game flows: joining games, ending games, resetting arenas.
  - Validate GUI interactions and PlaceholderAPI integrations.
  - Simulate multiple players for concurrency tests.

18.3 Testing Focus Areas

Game Lifecycles: Game start, tick updates, win conditions, game end, arena allocation, and reset.

Event Firing: `GameStartEvent`, `GameEndEvent`, `PlayerJoinGameEvent`, etc. Verify that events are fired at the correct times.

Party System: Invites, joins, leaves, and shared rewards. Party queueing into games.

Stats Tracking: Accurate recording of wins, kills, deaths, etc. Persistence across server restarts.

GUI Interactions: Dynamic menu updates. Click handlers and permissions.

External Plugin Integrations: PlaceholderAPI values. NPC interactions via Citizens2. WorldEdit region management.

18.4 Debugging Tools

- **Plugin Logger:** Use `plugin.getLogger().info(...)` for key actions. Enable or disable debug mode in `config.yml` for verbose output.
- **Minecraft Console:** Watch for error messages and stack traces. Use color-coded logs for different modules.
- **In-Game Messages:** Use admin-only messages for important plugin actions (e.g. game starts).

18.5 Enabling Debug Mode

Example in `config.yml`:

```
debug:
  enabled: true
```

When enabled:

- Verbose logs for key modules.
- Log module names and method calls.

18.6 Testing & Debugging Workflow

1. **Write Tests Early:** Write unit tests as you develop modules.
2. **Test in Isolation:** Mock dependencies (`ArenaManager`, `StatsManager`) during unit tests.
3. **Validate Event Flow:** Use test listeners to capture and assert events.
4. **Run Integration Tests:** Use a local test server for live testing.
5. **Debug Using Logs:** Add debug logs to modules with module names (e.g. [`GameManager`]). Keep logs concise and meaningful.

18.7 Example Unit Test (JUnit + Mockito)

```
@Test
public void testGameStartTriggersEvent() {
    ArenaManager arenaManager = mock(ArenaManager.class);
    StatsManager statsManager = mock(StatsManager.class);
    GameManager gameManager = new GameManager(arenaManager, statsManager, ...);

    // register a test listener
    PluginManager pluginManager = mock(PluginManager.class);
    doNothing().when(pluginManager).callEvent(any(GameStartEvent.class));

    GameInstance gameInstance = mock(GameInstance.class);
    gameManager.startGame("TNT_RUN", Arrays.asList(player));

    // verify event is called
    verify(pluginManager).callEvent(any(GameStartEvent.class));
}
```

18.8 Summary

The Testing & Debugging section ensures that the Minigames Plugin is stable, reliable, and ready for production use. By combining unit, integration, and system testing with structured debugging tools, the development team (including AI) can deliver a polished and maintainable plugin.