WHEEL OF FORTUNE - UML

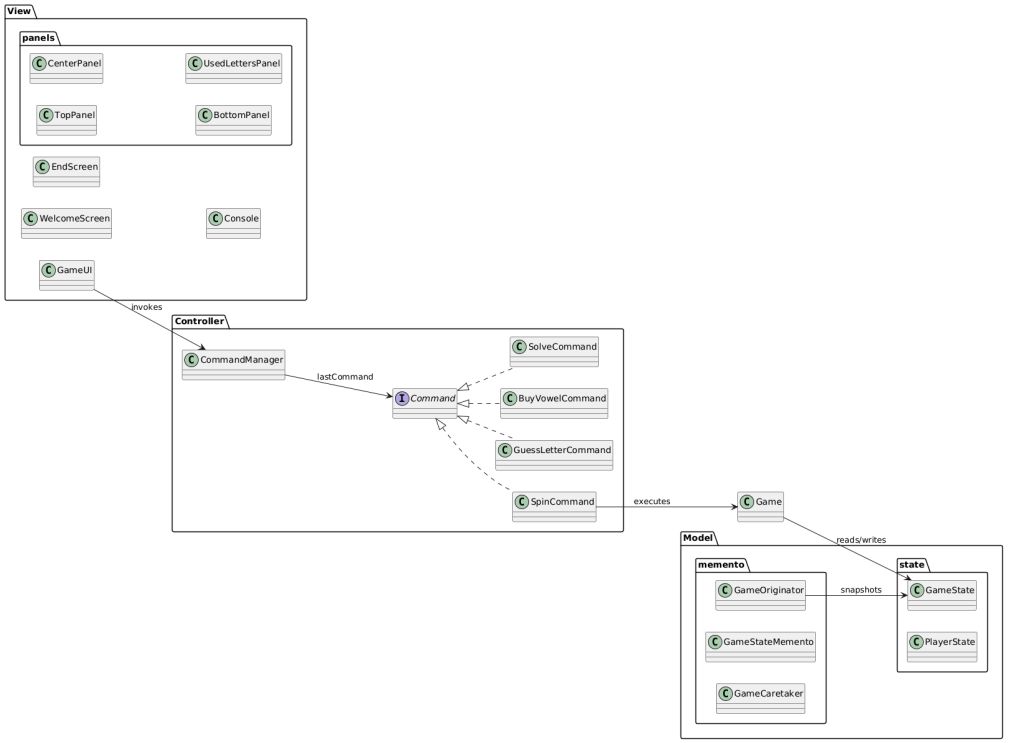
Adrián Cervantes, Gonzalo Fdez.-Martos,
Alejandro Serrano,Eva Sierra, Bogdan Shchepny
Software Engineering II-UCM

INDEX

1.Final architecture

Overall, the game's architecture cleanly separates concerns into two orthogonal views. In the MVC view, the Model encapsulates all of the game's data and persistence logic—GameState and PlayerState hold the current play information, while the memento classes (GameOriginator, GameStateMemento, GameCaretaker) provide snapshot and restore capabilities. The View layer, comprised of GameUI, the various screen classes, and the UI panels, is responsible solely for rendering the game state and capturing user input. The Controller is realized via the Command pattern: each user action (spin, guess, buy vowel, etc.) is wrapped in a Command object (SpinCommand, GuessLetterCommand, etc.) and executed through a central CommandManager, which also enables undo functionality.

In the distributed (client–server) view, each Client App runs a local UI (GameUI) and a GameClient component that sends and receives serialized NetworkMessage objects via MessageSender and MessageReceiver. The Server App hosts the authoritative GameServer, which maintains the master game state, processes incoming messages, and broadcasts updates back to all connected clients. This split ensures a responsive, local user experience while preserving a single, consistent source of truth on the server for multiplayer synchronization.



This package-level MVC diagram shows how our game cleanly separates UI, control flow, and data:

- View (top-left) houses all GUI screens and panels (GameUI, WelcomeScreen, TopPanel, etc.), responsible for rendering GameState and capturing user input.

- Controller (center) implements the Command pattern: CommandManager invokes concrete commands (SpinCommand, GuessLetterCommand, BuyVowelCommand, SolveCommand), each of which “executes” on the core Game logic.

- Model (bottom-right) splits into the state subpackage (in-memory GameState/PlayerState) and memento subpackage (GameOriginator, GameStateMemento, GameCaretaker) for snapshot/undo.

Arrows highlight that View “invokes” the controller, commands “execute” against the Game, and the memento layer “snapshots” the GameState, ensuring a clear, maintainable separation of concerns.

2. Final Structure

The final structure of our project is based on a modular and object-oriented architecture. We organized the application into clear subsystems, each responsible for a key aspect of the game logic. Through this structure, we support flexibility, extensibility, and maintainability.

The core components are distributed into different packages, each aligned with a specific functionality such as user interaction, AI player logic, game persistence, and multiplayer networking. Each class is clearly defined in terms of responsibility and connected through meaningful relationships such as aggregation, dependency, and interface implementation.

The figure below displays the final UML class diagram for our application, reflecting attributes, methods, and relationships.

Final Structure UML

Package	Responsibility
controller	Implements all user actions as commands.
game	Contains the central Game class which manages the entire game flow.
model.memento	Implements game state persistence.
model.state	Defines the state of the game and players.
players	implements AI players and strategy patterns.
ui	provides the graphical user interface using Swing components.

network	Handles multiplayer features via TCP communication.
utils	Provides utility functions and custom UI configuration.

3.User Stories Evolution

1. **As a player, I want to enter my name before starting the game.**

   Sprint 1

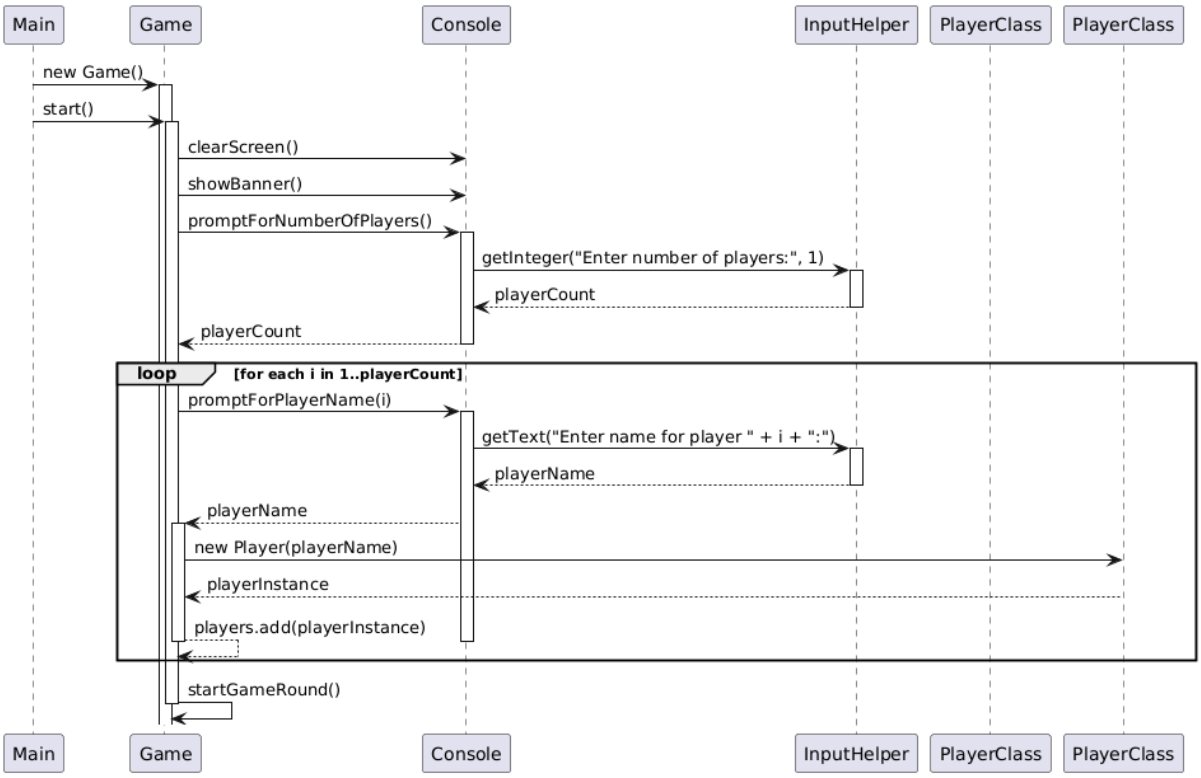
   Initial design

   - **Name entry in model:** `Game.start()` calls `registerPlayers()`, which first invokes `Console.promptForNumberOfPlayers()` → `InputHelper.getInteger("Enter number of players:", 1)`, then loops from 1 to playerCount calling `Console.promptForPlayerName(i)` → `InputHelper.getText("Enter name for player " + i + ":")`, constructs `new Player(name)`, and adds it to `game.players`.

   - **No validation:** empty or duplicate names are accepted as-is—there's no check on name length or uniqueness.

   - **Push-based I/O:** the model itself "pushes" prompts and reads input via `Console`/`InputHelper`.

   - **Why:** minimal logic to capture player identities before the game starts, cleanly separating core flow from console I/O.



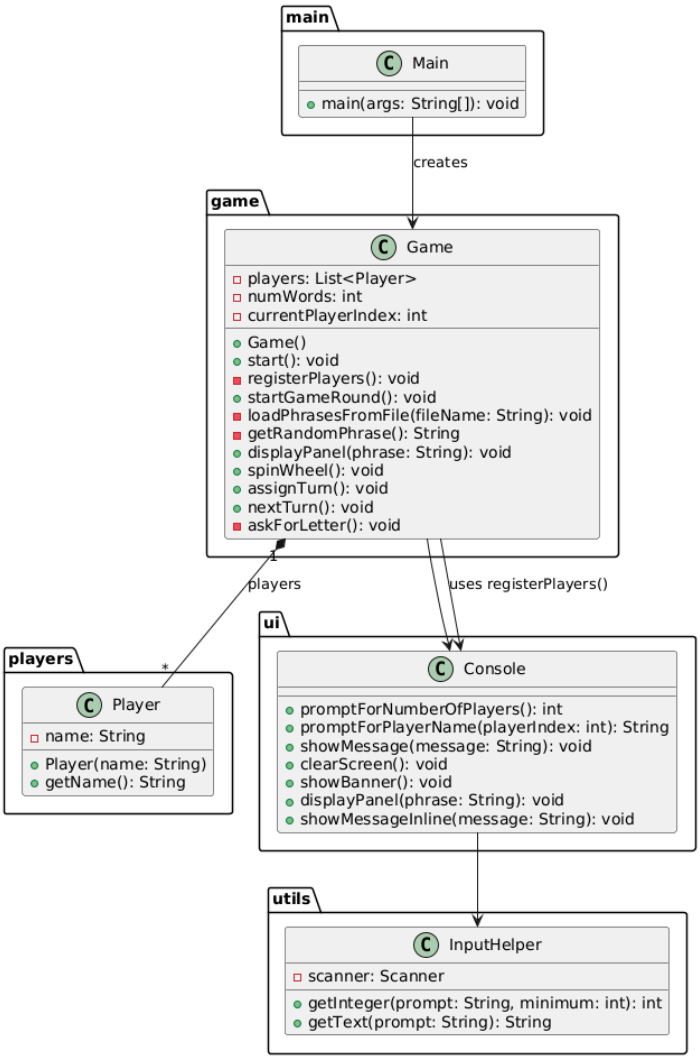
Interpretation & Design Rationale

- **Orchestration:** The flow begins in `Main`, which instantiates `Game` and calls `start()`. All player setup lives in `Game.start()`, keeping the application entry point focused solely on launch.

- **Responsibility separation:** `Game` handles only game-flow and player registration (`registerPlayers()`), while `Console` and `InputHelper` manage all I/O and basic parsing—adhering to the Single Responsibility Principle.

- **Push-based notification:** `Game` directly invokes console methods to prompt and display messages. This couples the model to the console layer, a trade-off accepted here for rapid prototyping.

- **Naming policy:** Player names are taken exactly in input order with no defaults or random placeholders—ensuring predictable registration.



Structure & Design Rationale
Class roles:

- **Game:** encapsulates `start() → registerPlayers() → startGameRound()`.
- **Player:** simple data holder for the player's `name`.
- **Console:** static façade for displaying prompts and messages.
- **InputHelper:** static utility that reads from `Scanner` and returns typed values.

Associations:

- `Game "1" ── "*" Player` models the player roster.

- `Game` → `Console`; `Console` → `InputHelper` indicate invocation dependencies, with no ownership of each other's data.

Evolution groundwork:

- Starts without design patterns to avoid premature complexity.

- Clear separation of concerns paves the way to introduce interfaces (e.g. `IPlayerRegistrar`), add name-validation, or swap in a GUI framework in future iterations.

Sprint 2

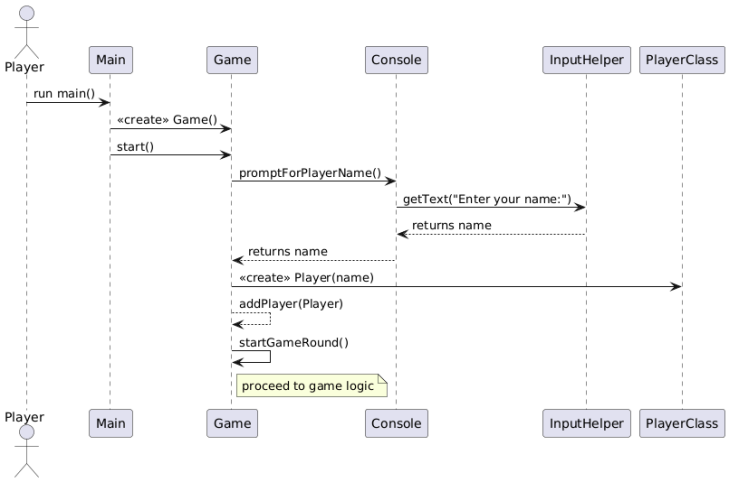
Name Entry Flow

- **Name entry in model**: `Game.start()` invokes `Console.promptForPlayerName()` → `InputHelper.getText("Enter your name:")`, then constructs `new Player(name)` and adds it to `game.players`.

- **No validation**: empty or blank names are accepted as-is—there's no check on length or disallowed characters.

- **Push-based I/O**: the core model "pushes" prompts and reads input via `Console`/`InputHelper`.

- **Why**: keep the minimum logic to capture the player's identity before any game logic runs, cleanly separating game flow from console I/O.



Interpretation & Design Rationale

- **Orchestration**: the entry point remains in `Main`; after instantiating `Game`, `Main` calls `start()`. All name-entry logic lives inside `Game.start()`, so the launch sequence is straightforward.

- **Responsibility separation**:

    - `Game` is responsible only for controlling the startup sequence and registering the player.

    - `Console` and `InputHelper` manage all user interaction and parsing. This adheres to the Single Responsibility Principle.

- **Push-based notification**: `Game` directly invokes console methods to prompt and receive the name. This does couple the domain to the console layer, but that trade-off is acceptable for rapid prototyping.

- **Naming policy**: the player's name is taken verbatim, with no defaults or fallbacks—ensuring predictable behavior and easy later insertion of validation or default-name logic.

Structure & Design Rationale

- **Class roles**:

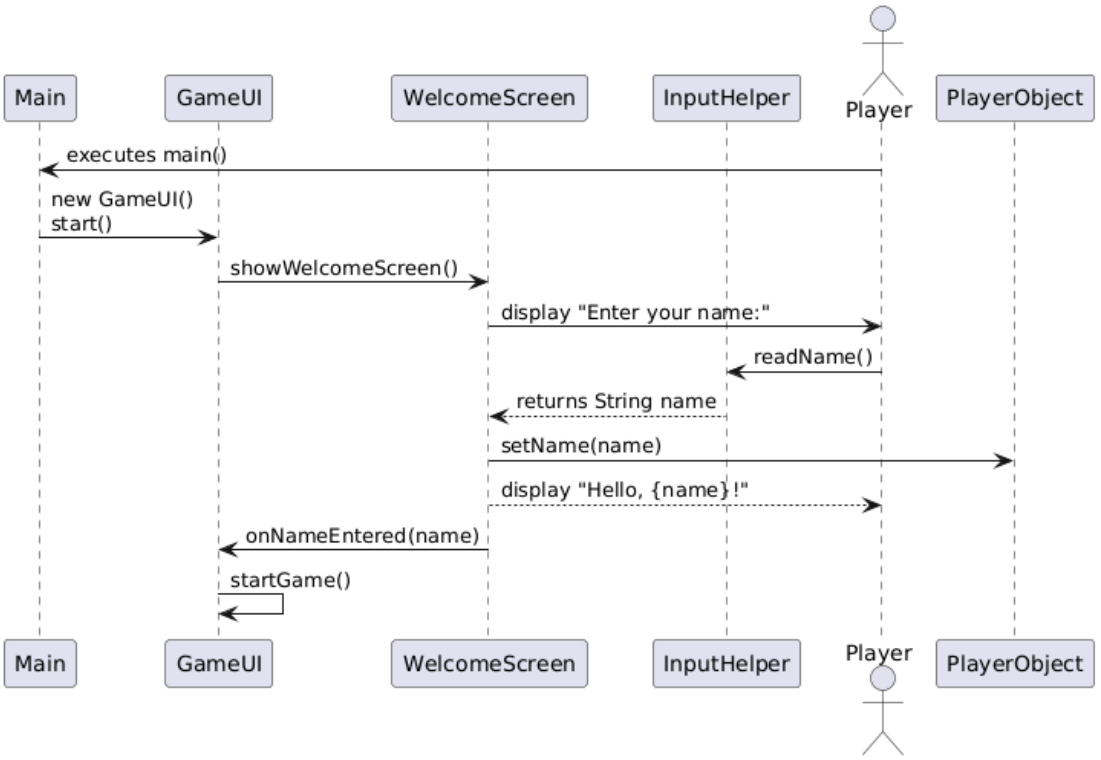
        - **Game**: orchestrates `start()` → `promptForPlayerName()` → `startGameRound()`.
        - **Player**: simple data holder for the name (and later, score/money).
        - **Console**: façade for displaying prompts and messages.
        - **InputHelper**: utility for reading typed input from the user.
    - **Associations**:
        - `Game "1" ── "*" Player` models the (currently single-element) player roster.
        - `Game → Console` and `Console → InputHelper` indicate invocation dependencies without data ownership.
    - **Evolution groundwork**:
        - Begins with minimal coupling and no design patterns to avoid over-engineering.

Sprint 3

Unified Player Name Registration Flow

- **GUI path:**
    - WelcomeScreen.show() invokes GameUI.registerPlayers() → JOptionPane.showInputDialog("Enter your name:")
    - Constructs `new Player(name)` and calls `Game.getInstance().addPlayer(player)`
- **Console path:**
    - Game.start() invokes Console.promptForPlayerName() → InputHelper.getText("Enter your name:")
    - Constructs `new Player(name)` and adds it to `game.players`
- **No validation:** empty or blank names are accepted as-is—no length or character checks
- **Push-based I/O:** UI and model "push" prompts and read input via JOptionPane (GUI) or Console/InputHelper (console)

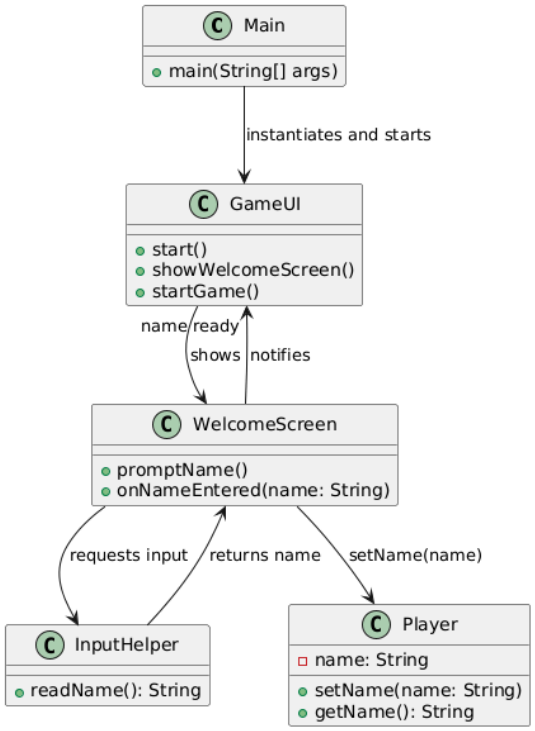
- **Why:** keep minimal logic to capture the player's identity before any game logic runs, cleanly separating game flow from Swing UI or console I/O



Interpretation & Design Rationale

- **Orchestration:**
    - Main remains the single entry point: after instantiating WelcomeScreen/GameUI or Console, Main calls show()/start().
    - All name-entry logic lives in `GameUI.registerPlayers()` or `Game.start()`, making launch sequence straightforward.
- **Responsibility separation:**
    - **Game** controls startup sequence and holds the player list.
    - **WelcomeScreen** & **GameUI** manage all Swing dialogs.
    - **Console** & **InputHelper** manage all text prompts and parsing.
    - Adheres to the Single Responsibility Principle.

- **Push-based notification:**
    - GameUI directly invokes JOptionPane dialogs; Game directly invokes console methods—acceptable coupling for rapid prototyping.
- **Naming policy:**
    - Player's name is taken verbatim, with no defaults or fallbacks—ensuring predictable behavior and easy insertion of validation or defaults later.



Structure & Design Rationale

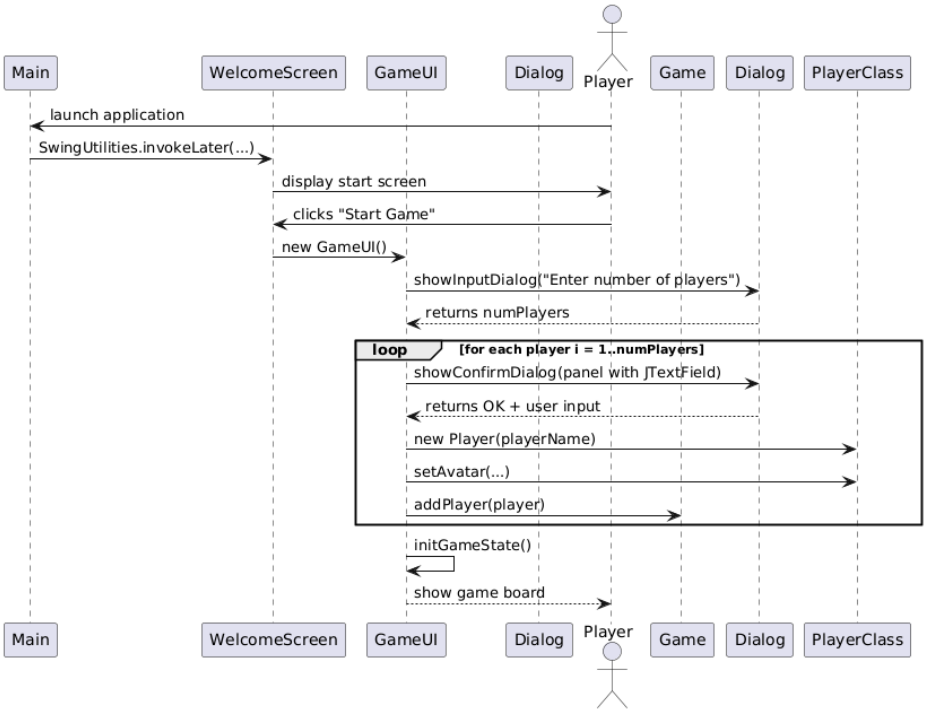
- **Class roles:**
    - **Main:** chooses GUI or console and kicks off registration.
    - **WelcomeScreen:** builds initial Swing window with name field and "Start" button.
    - **GameUI:** orchestrates Swing dialogs, constructs Player, and adds to Game.
    - **Game:** orchestrates start() → promptForPlayerName() → initGameState() and holds players.

    - **Player:** data holder for name (and later score/money/avatar).
    - **Console:** façade for text prompts, delegates to InputHelper.
    - **InputHelper:** utility for reading console input.
- **Associations:**
    - Main → (WelcomeScreen → GameUI) or (Game.start()) → Player
    - Game “1” ── “*” Player models the player roster
    - GameUI → Game and Game → Player show invocation dependencies
    - Console → InputHelper indicates console I/O dependency
- **Evolution groundwork:**
    - Minimal coupling and no heavy design patterns to avoid over-engineering.
    - Supports both GUI and console flows, ready for future validation, multi-player extension, or default-name logic.

Sprint 4

Simple Name Registration

- **GUI path:** `WelcomeScreen.show()` → `GameUI.registerPlayers()` → `JOptionPane.showInputDialog("Enter your name:")` → `new Player(name)` → `Game.getInstance().addPlayer(player)`
- **Console path:** `Game.start()` → `Console.promptForPlayerName()` → `InputHelper.getText("Enter your name:")` → `new Player(name)` → added to `game.players`
- **No validation:** empty or blank names are accepted as-is—no length or character checks
- **Push-based I/O:** the model “pushes” prompts and reads input via Swing (`JOptionPane`) or console (`Console`/`InputHelper`)
- **Why:** keep the minimum logic to capture the player’s identity before any game logic runs, cleanly separating game flow from I/O



Interpretation & Design Rationale

Orchestration

- **Main remains the single entry point:** after instantiating either `WelcomeScreen`/`GameUI` (GUI) or `Game` (console), it calls `show()` or `start()`.
- **All name-entry lives in exactly two methods—**`GameUI.registerPlayers()` and `Game.start()/Console.promptForPlayerName()`—making launch straightforward.

Responsibility separation

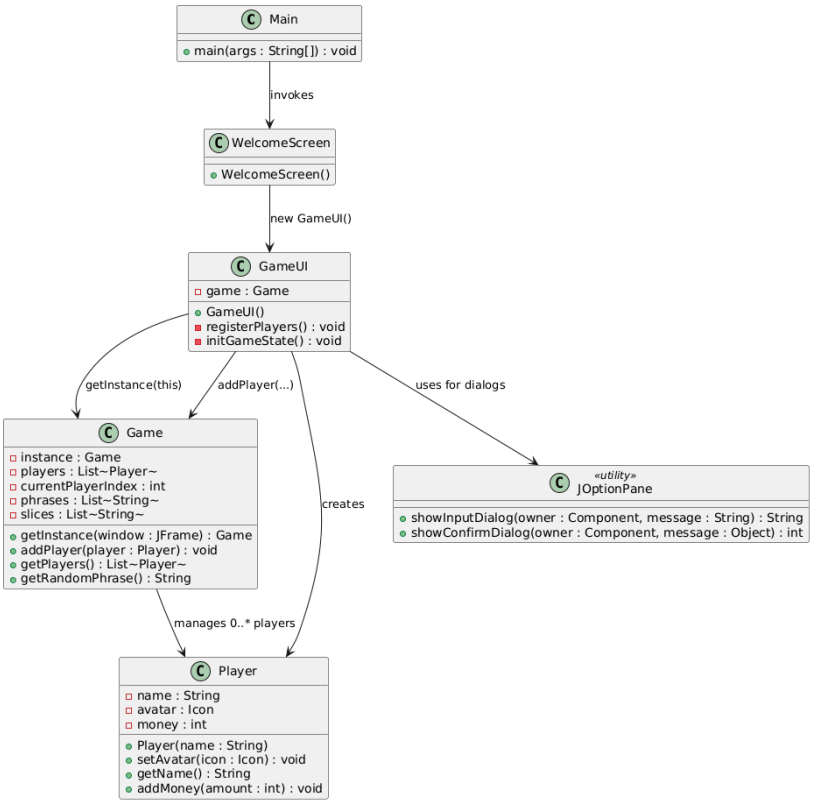
- Game: owns startup sequence and holds the player list.
- WelcomeScreen & GameUI: own Swing dialogs.
- Console & InputHelper: own text prompts and parsing.
- Adheres to the Single Responsibility Principle.

Push-based notification

- UI classes directly invoke `JOptionPane`; console path directly invokes `Console`/`InputHelper`—acceptable for rapid prototyping.

Naming policy

- Player names are taken verbatim—no defaults, trimming, or fallbacks—to ensure predictable behavior and easy future validation insertion.



Structure & Evolution Groundwork

Class roles

- Main: selects mode (GUI vs console), kicks off registration.
- WelcomeScreen: builds initial Swing window.
- GameUI: orchestrates Swing name-entry and adds `Player` to `Game`.

- Console: façade for console prompts.
- InputHelper: utility for reading console input.
- Game: singleton, holds `List<Player>`, controls startup.
- Player: simple data holder for `name`.

Associations

- `Main` → (`WelcomeScreen` → `GameUI` → `Game` → `Player`)
- `Main` → (`Game` → `Console` → `InputHelper` → `Game` → `Player`)
- `Game` "1"—"*" `Player` models the roster.

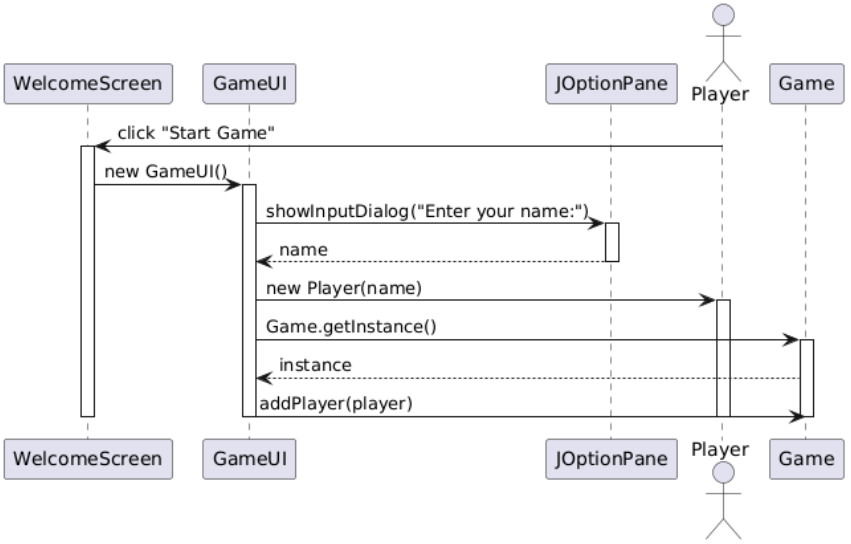
Evolution groundwork

- Minimal coupling and no heavy patterns to avoid over-engineering.
- Supports both GUI and console flows, ready for future validation rules, multiplayer counts, or default-name logic.

Sprint 5

Player Name Registration

• **GUI path:** `WelcomeScreen.show() → GameUI.registerPlayers() → JOptionPane.showInputDialog("Enter your name:") → new Player(name) → Game.getInstance().addPlayer(player)`
• **Console path:** `Game.start() → Console.promptForPlayerName() → InputHelper.getText("Enter your name:") → new Player(name) → added to Game.getInstance().getPlayers()`
• **No validation:** Empty or blank names are accepted as-is—no length checks, no trimming, no character restrictions.
• **Push-based I/O:** Prompts are "pushed" via Swing (JOptionPane) or console (Console/InputHelper), keeping the name-capture logic isolated in two spots.
• **Why:** Keep the minimum logic to capture the player's identity **before** any game rules execute, cleanly separating *when* we collect a name from *how* the game flow proceeds.



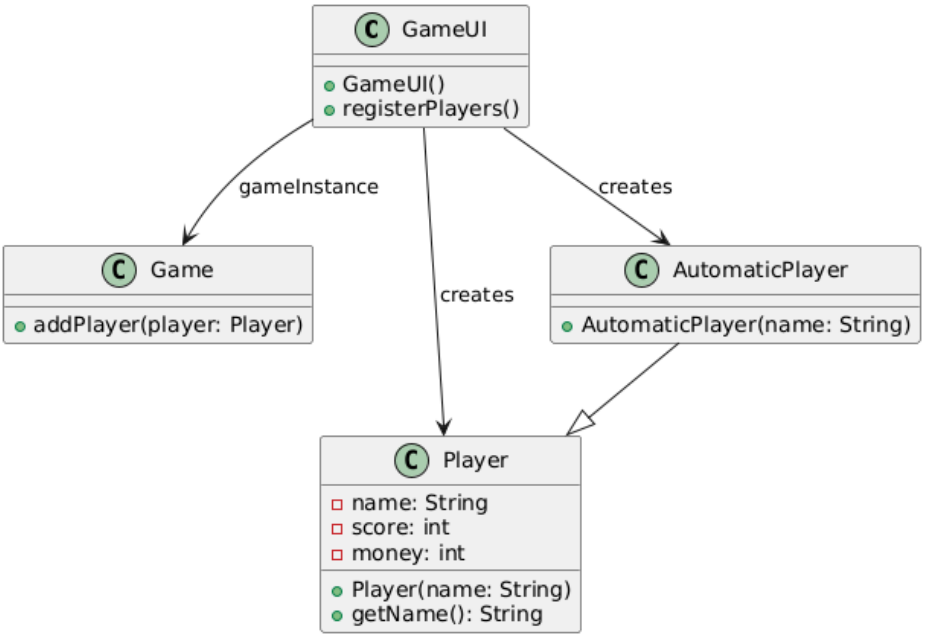
Interpretation & Design Rationale

• **Orchestration:**
Main remains the single entry point: it decides GUI vs. console mode and calls either WelcomeScreen.show() or Game.start(). All name entry lives in exactly two methods—GameUI.registerPlayers() and Game.start()/Console.promptForPlayerName()—making startup predictable and easy to trace.

• **Responsibility separation:**
– Game owns the startup sequence and holds the List<Player>.
– WelcomeScreen & GameUI own Swing dialogs.
– Console & InputHelper own console prompts and parsing.
Adheres to the Single Responsibility Principle, so UI concerns never bleed into game-logic classes.

• **Push-based notification:**
UI path invokes JOptionPane; console path invokes Console/InputHelper. This “push” style avoids callbacks or event hubs—appropriate for rapid prototyping and keeps the flow straightforward.

• **Naming policy:**
Player names are taken verbatim—no defaults, trimming, or fallbacks—to ensure predictable behavior and easy future insertion of validation rules.



Structure & Evolution Groundwork

• **Class roles:**
– Main: selects mode (GUI vs. console) and kicks off registration.
– WelcomeScreen: builds the initial Swing window.
– GameUI: orchestrates Swing name-entry and adds each Player to Game.
– Console: façade for console prompts.
– InputHelper: utility for reading and parsing console input.
– Game: singleton, holds List<Player>, controls startup.
– Player: simple data holder for name.

• **Associations:**
– Main → WelcomeScreen → GameUI → Game → Player
– Main → Game → Console → InputHelper → Game → Player
– Game “1”—“*” Player (roster modeled as one-to-many)

• **Evolution groundwork:**
Minimal coupling, no over-engineered patterns. Supports both GUI and console flows out of the box. Ready for future enhancements: player-count limits, name validation, default-name schemes, etc.

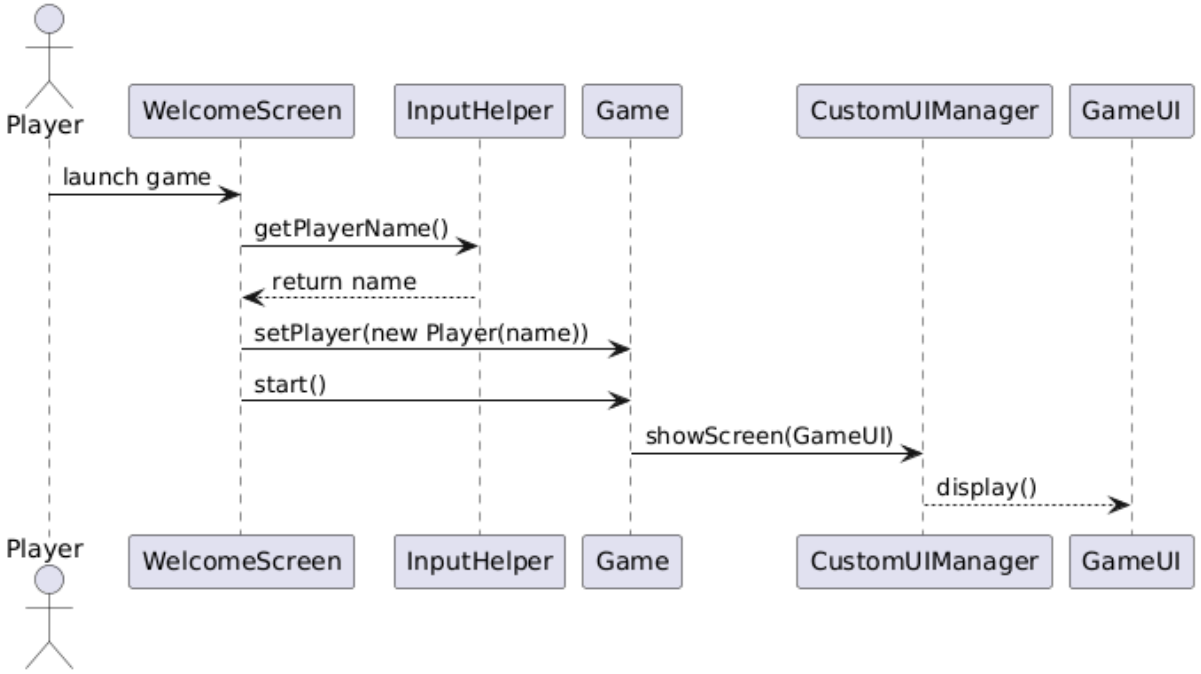
Sprint 6

Player Name Registration

- **Console path**:

    - **Entry**: `Main.main()` → decides console mode → calls `Game.start()`
    - **Prompt**: `Console.promptForPlayerName()` → internally calls `InputHelper.getText("Enter your name:")`
    - **Construction**: `new Player(name)` → added to `Game.getInstance().getPlayers()`

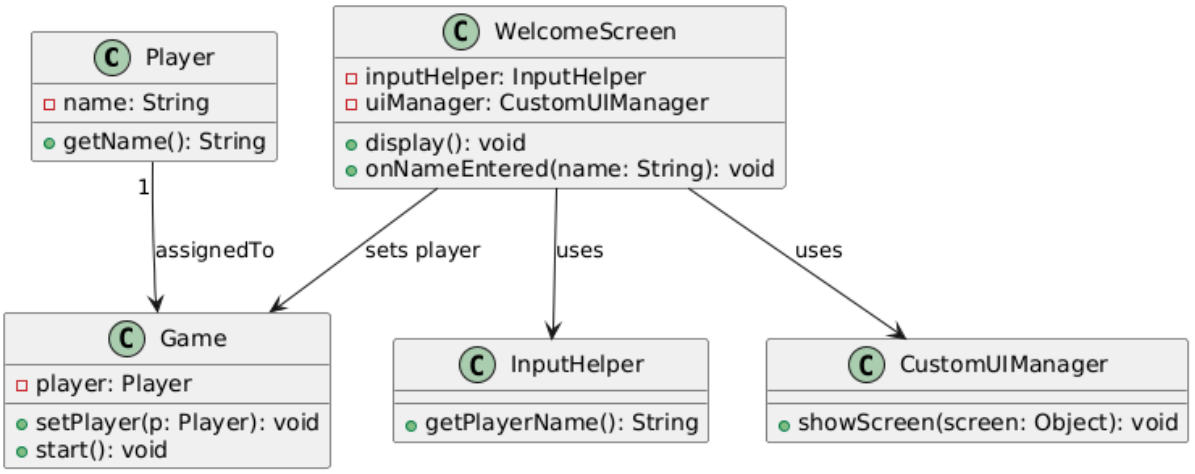
- **No validation**: Empty or blank names are accepted without checks (no trimming, length or character restrictions).
- **Push-based I/O**: The console "pushes" a prompt via `Console`/`InputHelper`, isolating name-capture logic in one location.
- **Why**: Keep startup logic minimal—collect the player's identity upfront before any game rules run, and isolate I/O from game flow.



Interpretation & Design Rationale

- **Orchestration**

    - **Single entry point**: `Main` decides mode (console) and calls `Game.start()`.
    - **Predictability**: All name entry lives in `Game.start()` → `Console.promptForPlayerName()`, making startup easy to trace.
- **Responsibility separation**
    - `Game`: owns the startup sequence and holds the `List<Player>`.
    - `Console`: façade for all console-based prompts.
    - `InputHelper`: low-level text input and parsing.
    - `Player`: simple data holder.
    - **Result**: UI concerns (console I/O) never bleed into core game-logic classes, adhering to the Single Responsibility Principle.
- **Push-based notification**
    - The console path directly invokes `Console.promptForPlayerName()` and `InputHelper.getText(...)`.
    - No callbacks or event buses—ideal for a first, rapid-prototype version with straightforward flow.
- **Naming policy**
    - Player names are accepted verbatim—no defaults, trimming, or fallbacks—to ensure fully predictable behavior and make it trivial to add validation later.



Structure & Evolution Groundwork

- **Class roles**
  - **Main**: selects console mode and kicks off the game.
  - **Console**: handles all console prompts.
  - **InputHelper**: utility for reading and parsing console input.
  - **Game**: singleton, holds `List<Player>`, controls startup.
  - **Player**: immutable name holder.
- **Associations**
  - `Main` → `Game` → `Console` → `InputHelper` → (`new Player(name)`) → `Game.getPlayers()`
  - `Game` "1" —— "*" `Player` (the roster)
- **Evolution groundwork**
  - **Minimal coupling** and no over-engineered patterns—easy to extend.
  - Ready for enhancements:
    - name validation or trimming rules
    - GUI flow addition

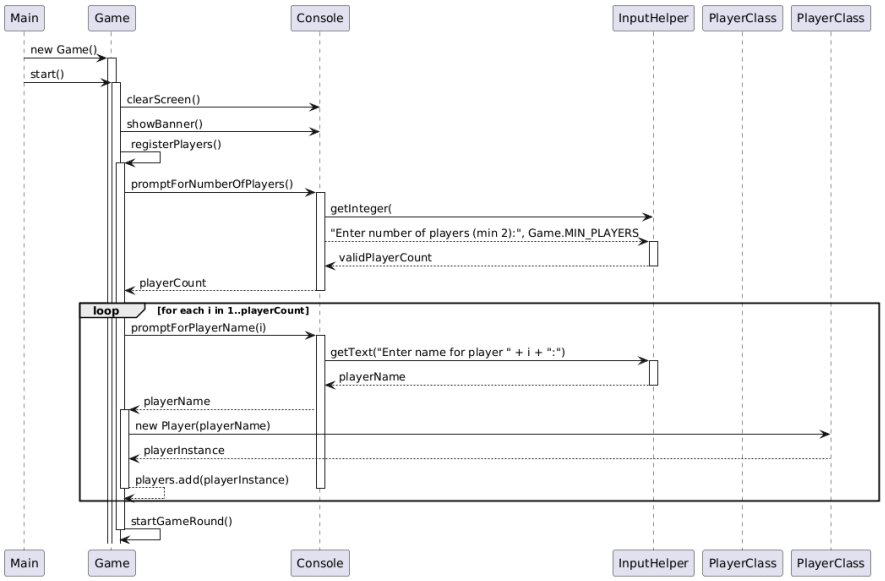
        - multi-player support
        - default-name fallbacks
        - pluggable I/O (e.g., file input, networked names)

2. **As a developer, I want the game to have at least two players**

   Sprint 1

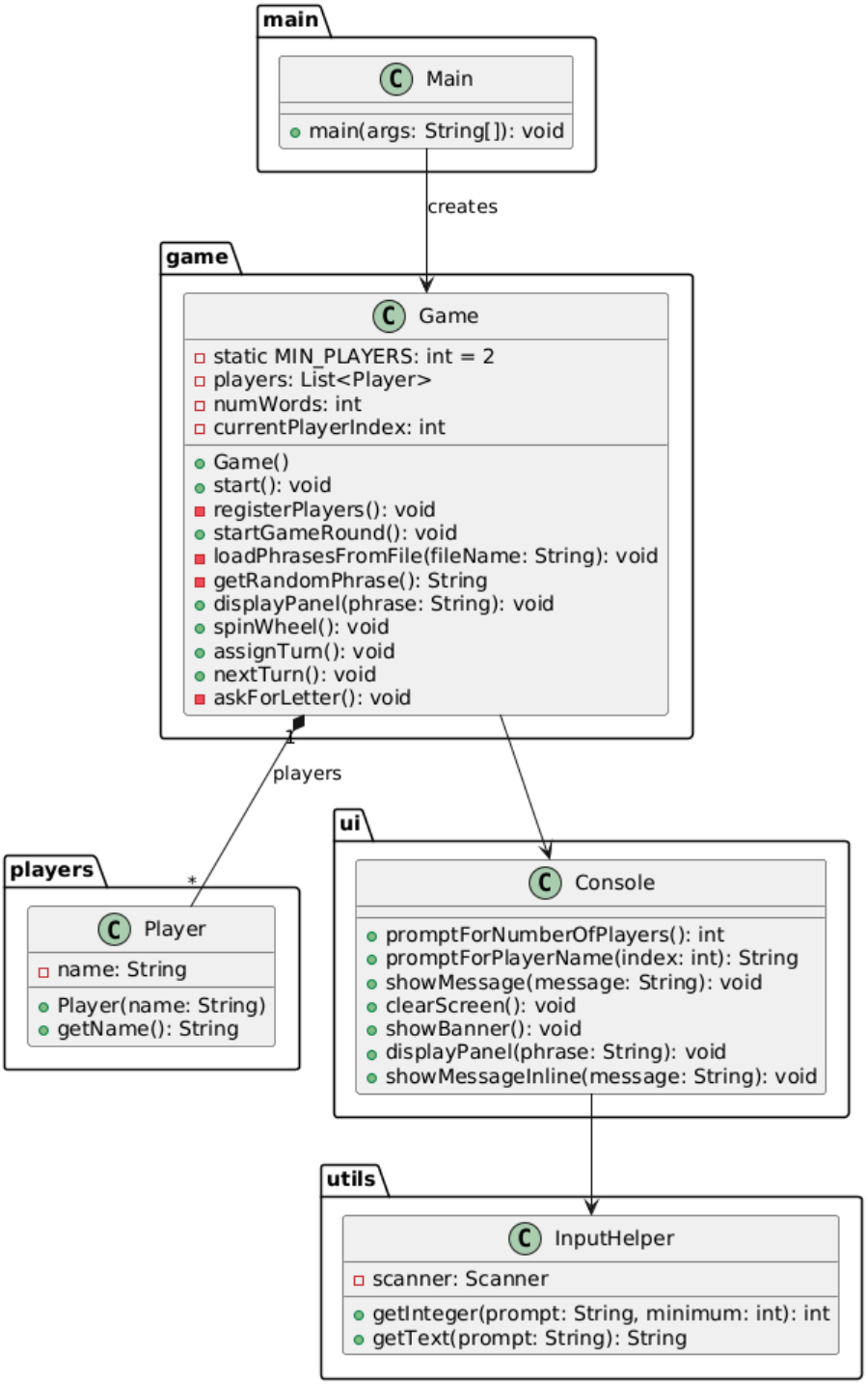
   Initial design

   - **Minimum-players enforcement in model:** `Game.start()` calls `registerPlayers()`, which invokes
     - `Console.promptForNumberOfPlayers()` → `InputHelper.getInteger("Enter number of players (min 2):", 2)`
     - Loops from 1 to playerCount exactly as before, calling `Console.promptForPlayerName(i)` → `InputHelper.getText("Enter name for player " + i + ":")`, constructing `new Player(name)` and adding it to `game.players`.
   - **Enforced constraint only at input time:** by passing `2` as the minimum to `getInteger()`, the model guarantees at least two players. No further checks on maximum count or duplicate names.
   - **Push-based I/O:** the model "pushes" prompts and reads input via `Console`/`InputHelper`, just like in earlier iterations.
   - **Why:** minimal change to existing flow—reuse the same registration logic but require two or more participants before game start.



Interpretation & Design Rationale

- **Orchestration:** Retains the same entry-point setup: `Main` instantiates `Game` and calls `start()`, which now ensures the game can't proceed until at least two players are registered. Startup logic remains centralized.
- **Responsibility separation:**
  - **Game** still handles only game-flow and player registration.
  - **Console** and **InputHelper** remain responsible for all I/O and input validation (including the new minimum constraint).
  - No extra validation logic is added to `Game` itself—delegation keeps `Game` focused.
- **Push-based notification:** As before, `Game` directly invokes console prompts. The tighter coupling is accepted for rapid prototyping.
- **Minimum-player policy:** By embedding the minimum into the prompt, we enforce the business rule at the earliest point and prevent invalid game states (i.e., single-player or zero-player games).



Structure & Design Rationale
Class roles:

- **Game:** orchestrates `start() → registerPlayers() → startGameRound()`, now with a minimum-player requirement.

- **Player:** holds the player's `name`.
- **Console:** static façade for displaying prompts and messages, now including the "min 2" prompt.
- **InputHelper:** static utility that reads from `Scanner` and enforces the `minimum` parameter passed by `Game`.

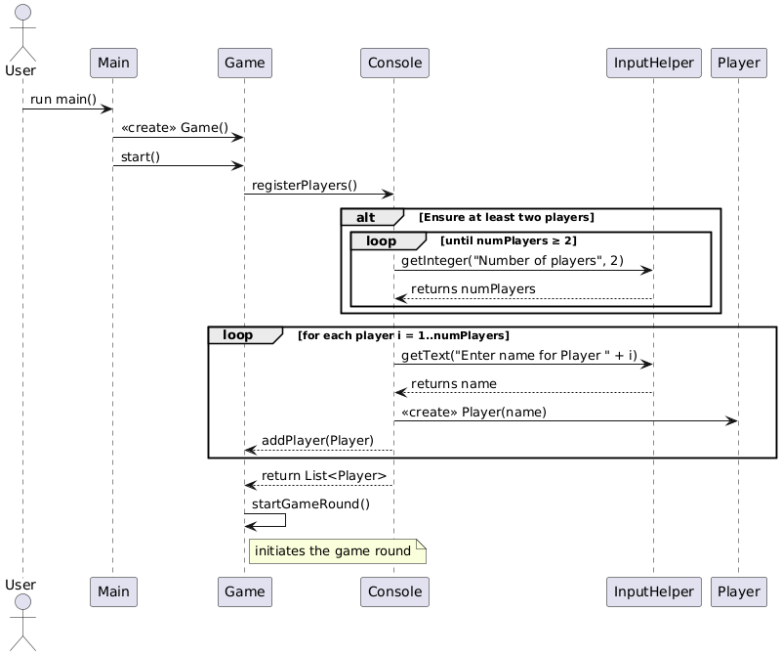
Associations:

- `Game "1" ── "*" Player` models the player roster unchanged.
- `Game` → `Console`; `Console` → `InputHelper` denote invocation dependencies, maintaining low coupling.

Sprint 2

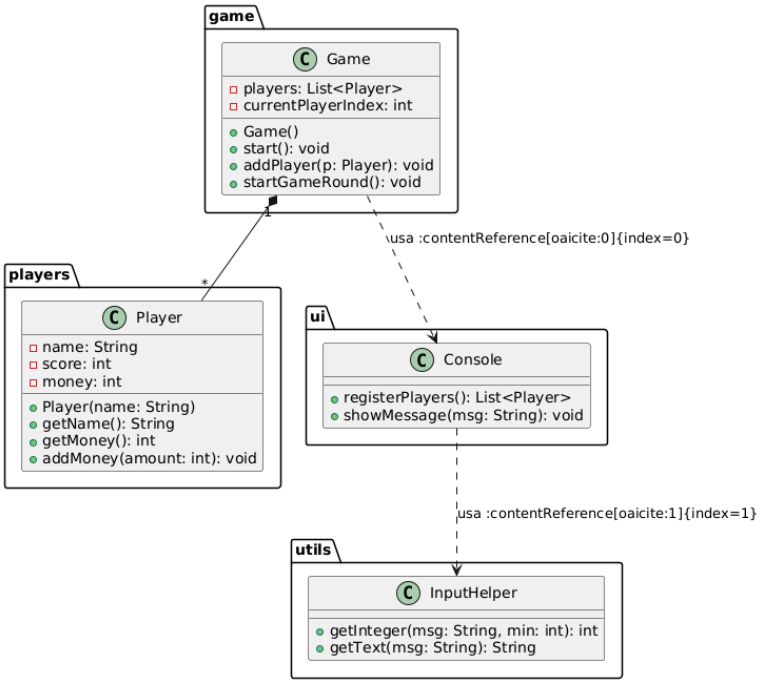
Multi-Player Registration

- **Player count entry in model**: `Game.start()` invokes `Console.promptForNumberOfPlayers()` → `InputHelper.getInteger("Enter number of players:", 2)`, ensuring at least two players. Then it loops from 1 to playerCount calling `Console.promptForPlayerName(i)` → `InputHelper.getText("Enter name for player " + i + ":")`, constructs `new Player(name)`, and adds it to `game.players`.
- **Minimal name handling**: while the count is validated (≥2), individual names are accepted verbatim—empty or duplicate names pass through with no checks.
- **Push-based I/O**: the core model "pushes" prompts and reads input via `Console`/`InputHelper`.
- **Why**: to capture at least two player identities before any game logic runs, keeping the domain flow separate from console I/O.



Interpretation & Design Rationale

- **Orchestration**: `Main` remains responsible for launch; it instantiates `Game` and calls `start()`. All multi-player setup lives in `Game.start()`, so the entry point is clean.
- **Single Responsibility**:
  - `Game` controls startup flow and enforces the two-player minimum.
  - `Console` and `InputHelper` handle all user interactions and parsing.
- **Coupling trade-off**: `Game` directly invokes console methods to register players—this couples domain logic to the console layer but accelerates prototyping.
- **Predictable behavior**: requiring a minimum of two players up front avoids downstream errors, while deferring name validation simplifies initial implementation.



Structure & Evolution

- **Class roles**:
    - **Game**: orchestrates `start()` → `registerPlayers()` → `startGameRound()`.
    - **Player**: data holder for a name (and later, scores/money).
    - **Console**: façade for prompts and messages.
    - **InputHelper**: utility for reading typed input.
- **Associations**:
    - `Game "1" ── "*" Player` models the roster.
    - `Game → Console` and `Console → InputHelper` denote invocation dependencies without data ownership.
- **Future-ready**: clean separation allows easy introduction of interfaces (e.g. `IPlayerRegistrar`), addition of name-validation rules, or swapping in a GUI without major refactoring.

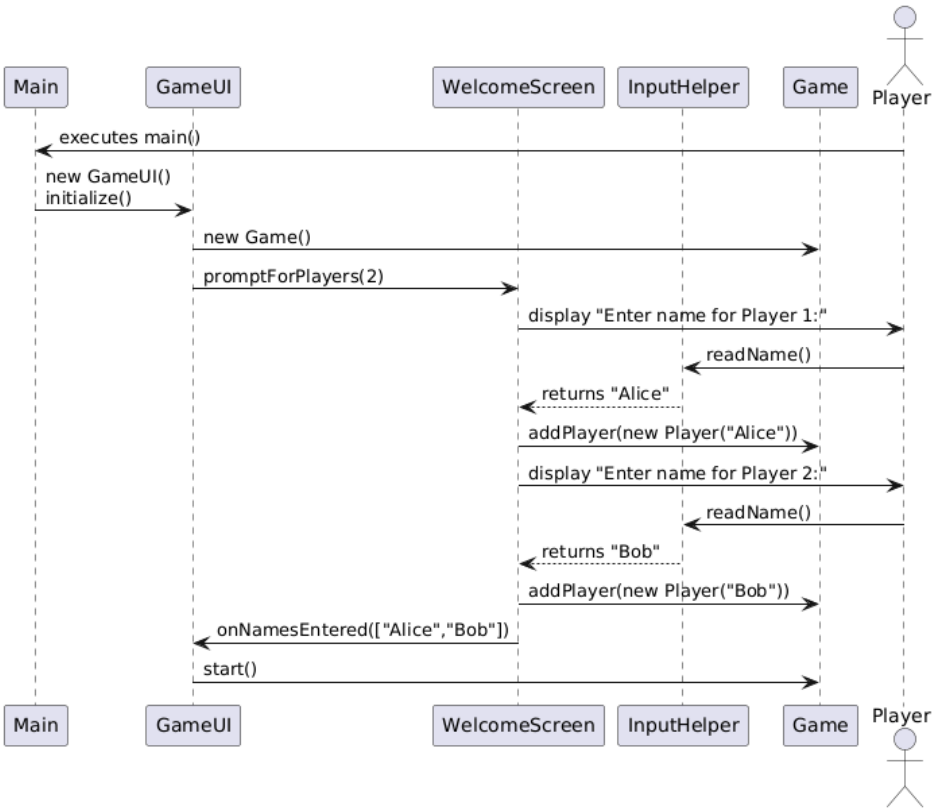
Sprint 3

Mandatory Two-Player Initialization Flow

- **GUI path:**
    - WelcomeScreen.show() invokes GameUI.registerPlayers() →
        - Dialog asks “Enter number of players:” via `JOptionPane.showInputDialog(...)`.
        - If number < 2, show error and re-prompt.
        - Loop for each player i=1…numPlayers (numPlayers ≥ 2):
            - Prompt `JOptionPane.showInputDialog("Enter name for Player " + i + ":")`
            - Construct `new Player(name)` and call `Game.getInstance().addPlayer(player)`
- **Console path:**
    - Game.start() invokes `Console.promptForPlayerCount()` → `InputHelper.getInteger("Enter number of players:", 2)`
    - Loop for i=1…count:
        - Call `Console.promptForPlayerName(i)` → `InputHelper.getText("Enter name for Player " + i + ":")`
        - Construct `new Player(name)` and add to `game.players`
- **Validation:**
    - Enforce minimum of **two** players.
    - Reject counts < 2 with re-prompt.
    - Still accept empty or blank names as entered (no name-content validation yet).
- **Push-based I/O:**

- GameUI or Game directly **push** prompts to the user; they handle enforcement of the two-player rule.

- **Why:**
  - Guarantee at least two participants before any game rounds run.
  - Centralize the multi-player requirement in the startup flow, keeping game logic free of player-count checks afterward.



Interpretation & Design Rationale

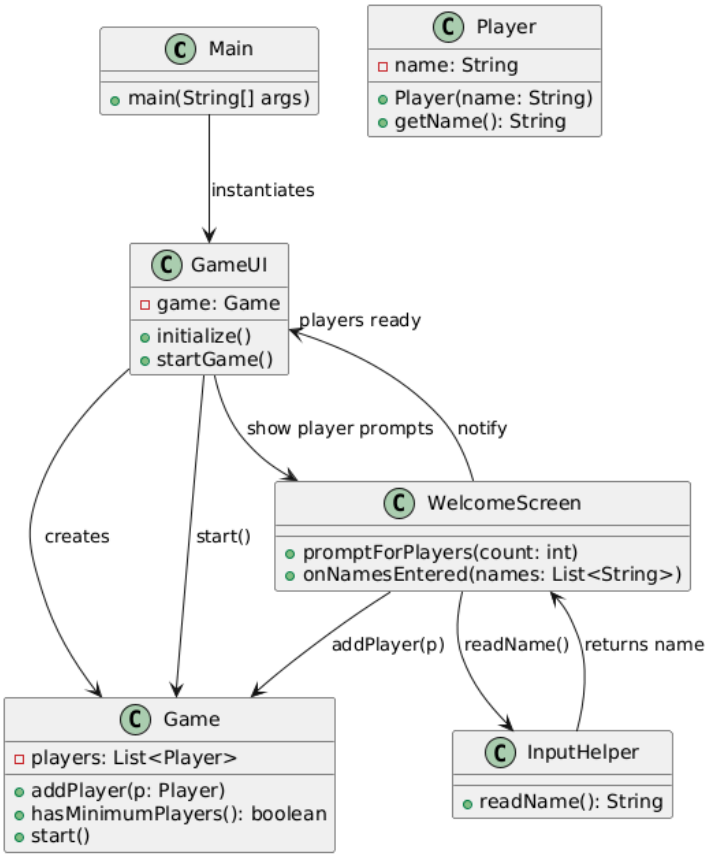
- **Orchestration:**
  - Main remains entry point: instantiates UI or console mode, then calls `show()` or `start()`.
  - Registration now collects a player count first, then collects that many names—ensuring the two-player minimum before proceeding.

- **Responsibility separation:**

- **Game** still controls the high-level flow but no longer needs to check player count—UI or Console enforces it.
- **GameUI** & **WelcomeScreen** manage Swing dialogs for count and names.
- **Console** & **InputHelper** manage text-based prompts and integer parsing.
- Adheres to Single Responsibility: each layer handles its own validation concerns.

- **Push-based notification:**
  - UI layer reports errors (e.g., “Must enter at least 2 players”) and re-launches prompts; Game only sees valid input.
  - Avoids coupling game logic to validation loops.

- **Evolution readiness:**
  - Future rules (max players, name uniqueness) can slot into the same UI/Console flows without touching Game.



Structure & Design Rationale

- **Class roles:**
    - **Main:** chooses GUI or console, then kicks off registration.
    - **WelcomeScreen:** displays count-and-name dialogs, hands off to GameUI.
    - **GameUI:** orchestrates count prompt, name prompts, constructs `Player`, calls `Game.addPlayer()`.
    - **Game:** holds list of players and proceeds to rounds once registration completes.
    - **Player:** holds name (and later score/money/avatar).
    - **Console:** façade for count and name prompts, delegates numeric/text parsing to InputHelper.
    - **InputHelper:** utility for reading integers (enforcing minimum) and text input.
- **Associations:**
    - Main → (WelcomeScreen → GameUI) or (Game.start()) → Player
    - Game “1” ── “*” Player models the roster (now guaranteed ≥ 2).
    - UI/Console layers depend on Game only via `addPlayer()`.

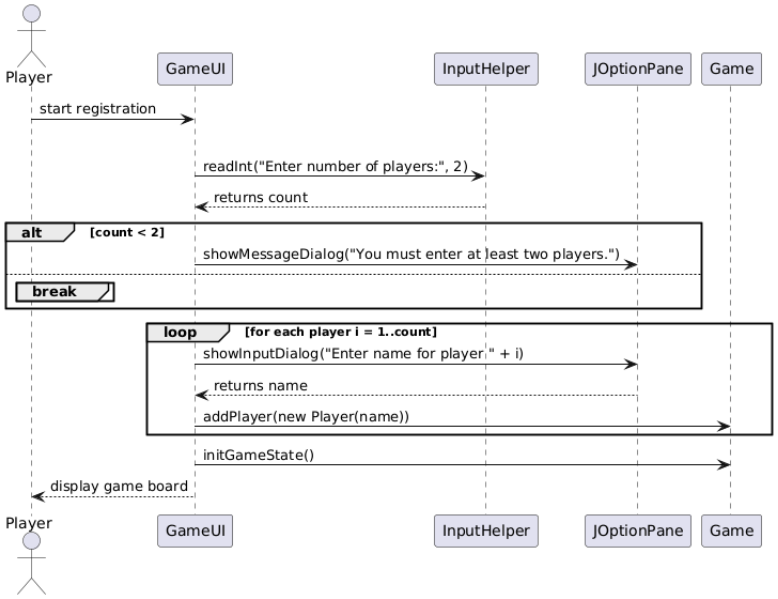
Sprint 4

Minimum Two-Player Enforcement

- **GUI path:**
  `GameUI.registerPlayers()` →
  loop: `InputHelper.readInt("Enter number of players:", 2)` → **if count < 2:** `JOptionPane.showMessageDialog("You must enter at least two players.")` → **repeat** → **once valid, for each i:** `JOptionPane.showInputDialog("Enter name for player " + i)` → `new Player(name)` → `Game.getInstance().addPlayer(player)`
- **Console path:**
  `Game.start()` →

loop: `InputHelper.readInt("Enter number of players:", 2)` → if count < 2: `Console.showMessage("You must enter at least two players.")` → repeat → once valid, for each i: `Console.promptForPlayerName("Enter name for player " + i)` → `new Player(name)` → added to `game.players`

- **Validation:** enforces a minimum of two players before proceeding—no other checks on name content or length
- **Push-based I/O:** the model "pushes" prompts and reads input via Swing dialogs (`JOptionPane`) or console helpers (`Console`/`InputHelper`**)**
- **Why:** ensure a meaningful multiplayer experience by guaranteeing at least two participants, while still keeping I/O logic separate from game rules



Interpretation & Design Rationale

Orchestration

- Main still drives mode selection and calls either `GameUI.registerPlayers()` or `Game.start()`.
- Name-and-count entry is now centralized in those two methods, making setup flow clear.

Responsibility separation

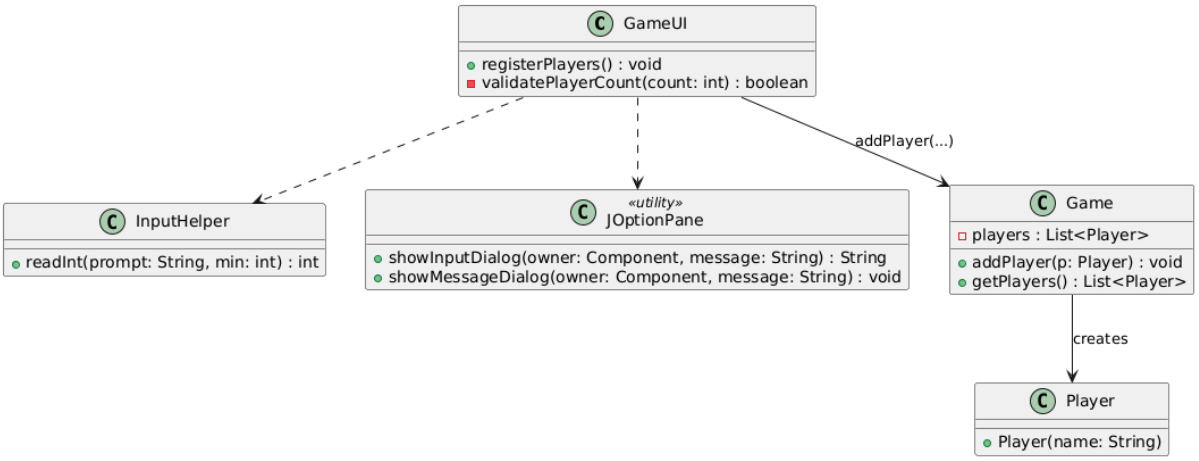
- GameUI/Game handle only orchestration and player addition.
- InputHelper reads and enforces the minimum count.
- JOptionPane/Console display prompts and error messages.
- Adheres to Single Responsibility Principle.

Push-based notification

- Registration methods directly invoke UI or console prompts—acceptable trade-off for prototyping.

Naming policy

- Player names remain verbatim. Count validation is orthogonal to name handling.



Structure

Class roles

- GameUI: orchestrates GUI count+name flow and adds players.
- Console: orchestrates console count+name flow and adds players.
- InputHelper: utility for reading integers with a lower bound.
- Game: singleton, holds `List<Player>`, controls startup.

- Player: data holder for `name`.

Associations

- `GameUI` → `InputHelper` & `JOptionPane` → `Game` → `Player`
- `Console` → `InputHelper` & `Console` UI → `Game` → `Player`

Sprint 5

Minimum Player Enforcement

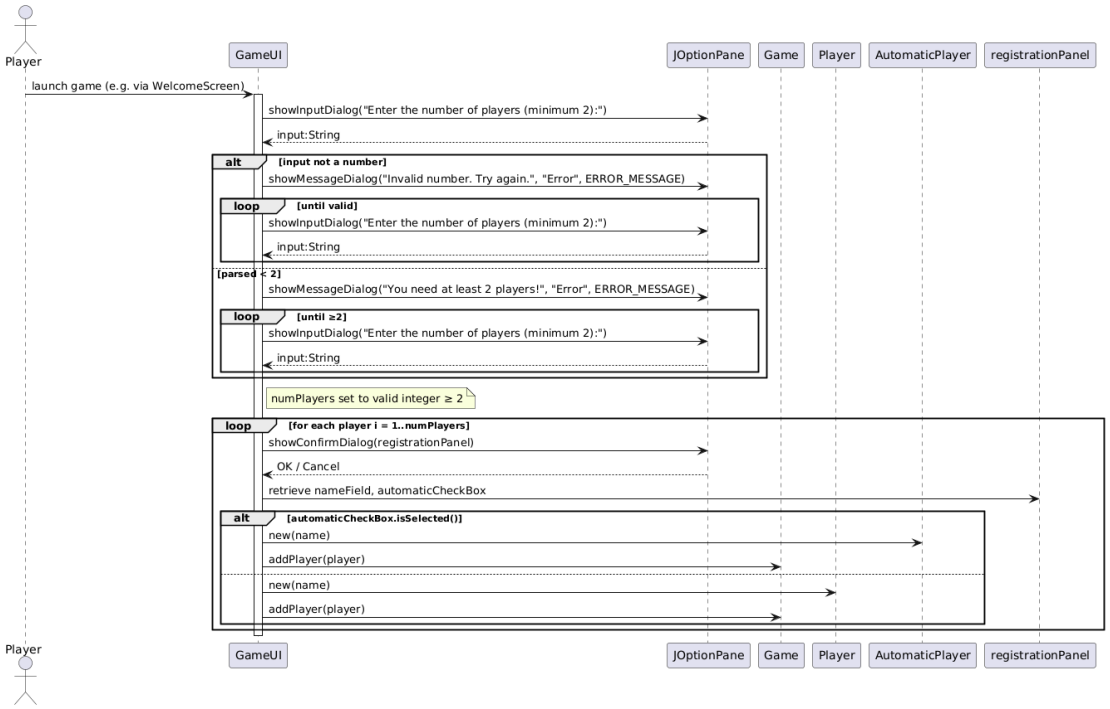
• **GUI path:**
WelcomeScreen.show() → GameUI.registerPlayers() → JOptionPane.showInputDialog("Enter number of players (minimum 2):") → if input < 2 showMessageDialog("At least two players required") → repeat prompt → for each player i: JOptionPane.showInputDialog("Enter your name:") → new Player(name) → Game.getInstance().addPlayer(player)

• **Console path:**
Game.start() → Console.promptForPlayerCount(minimum 2) → InputHelper.getInteger("Enter number of players:", 2) → for each player i: InputHelper.getText("Enter your name:") → new Player(name) → added to Game.getInstance().getPlayers()

• **Validation:**
Enforces a minimum of two participants. Any attempt to start with fewer than two causes an immediate, user-friendly error and reprompt.

• **Push-based I/O:**
The UI path pushes dialogs via Swing; the console path pushes prompts via Console/InputHelper. Both maintain linear, imperative flows.

• **Why:**
Many game features (e.g., turn rotation, competition) require at least two players. Enforcing this upfront prevents invalid game states and keeps core logic focused on gameplay rather than edge cases.



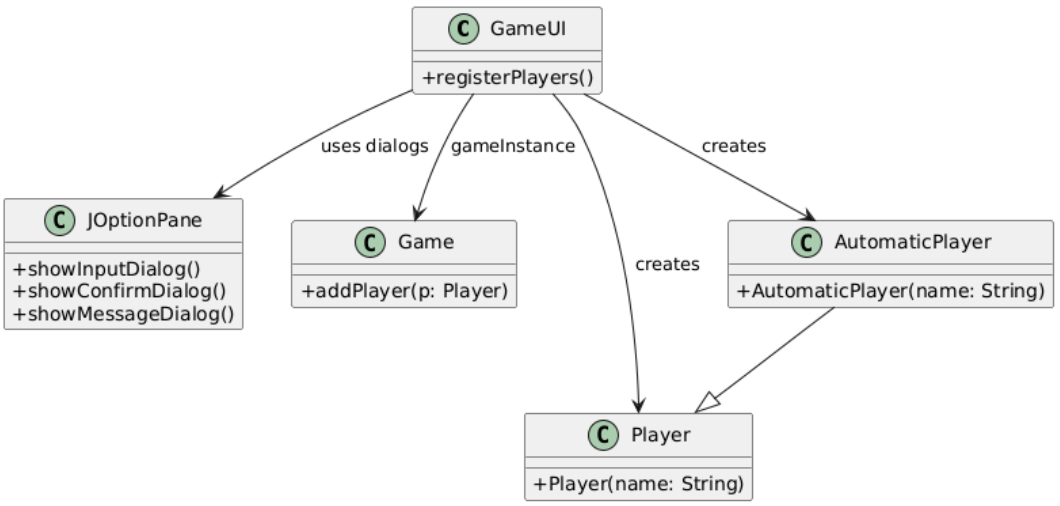
Interpretation & Design Rationale

• **Orchestration:**
Main remains the single launch point. GUI mode calls WelcomeScreen.show() → registerPlayers(); console mode calls Game.start() → promptForPlayerCount(). Both flows then loop to register exactly the validated number of players, ensuring startup consistency.

• **Responsibility separation:**
– Game owns the player roster and overall startup sequence.
– WelcomeScreen & GameUI handle all Swing dialogs and validation UI.
– Console & InputHelper handle all console prompts and numeric/text validation.
This keeps UI/validation concerns out of Game and Player classes, adhering to the Single Responsibility Principle.

• **Push-based notification:**
By invoking JOptionPane or Console/InputHelper directly, the design avoids callback complexity. The flow remains straightforward, suitable for rapid prototyping and future extension.

• **Validation policy:**
Number-of-players validation is centralized in two methods (`GameUI.registerPlayers()` and `Console.promptForPlayerCount()`). Names remain verbatim for simplicity, but count checks guarantee a valid player set before proceeding.



Structure & Evolution Groundwork

• **Class roles:**
– Main: selects mode (GUI vs. console) and kicks off registration.
– WelcomeScreen: constructs initial Swing window.
– GameUI: orchestrates GUI validation (player count ≥ 2), name entry, and adds each Player to Game.
– Console: façade for console-mode prompts, including getInteger(min 2) for count.
– InputHelper: utility for reading and validating console input.
– Game: singleton, holds List<Player>, controls startup.
– Player: simple data holder for name.

• **Associations:**
– Main → WelcomeScreen → GameUI → Game → Player
– Main → Game → Console → InputHelper → Game → Player
– Game “1”—“*” Player (roster modeled as one-to-many)

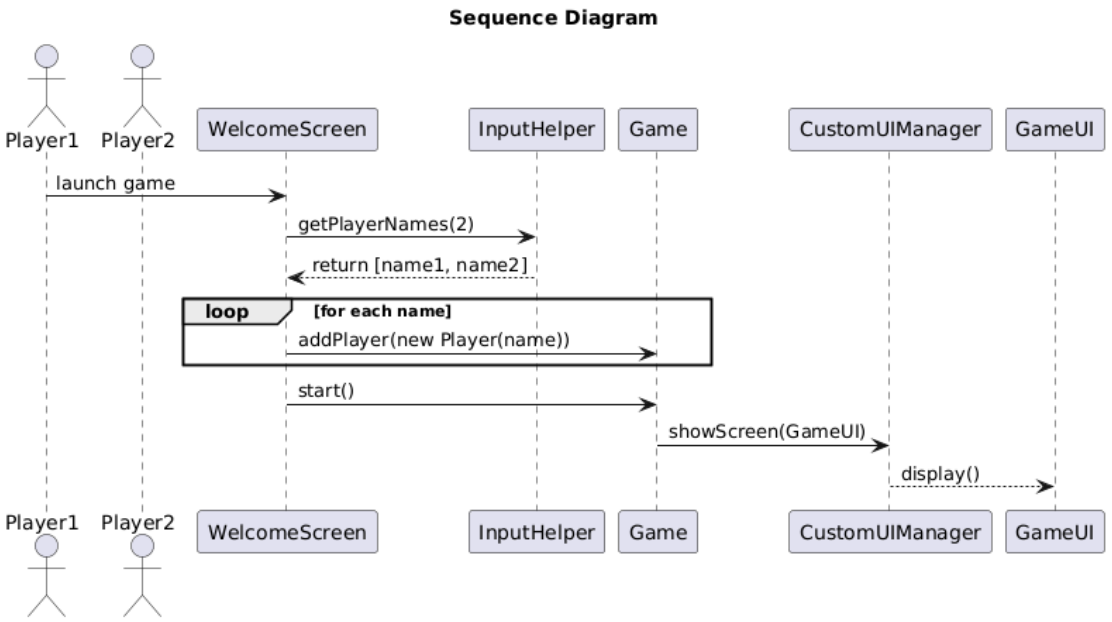
• **Evolution groundwork:**
Minimal coupling and clear separation allow easy addition of a maximum player limit, per-player configuration screens, or default-name logic. Validation rules can evolve independently in the respective UI or console helper classes.

Sprint 6

Multi-Player Registration

- **GUI path**:
    - **Entry**: `Main.main()` → decides GUI mode → calls `WelcomeScreen.show()`

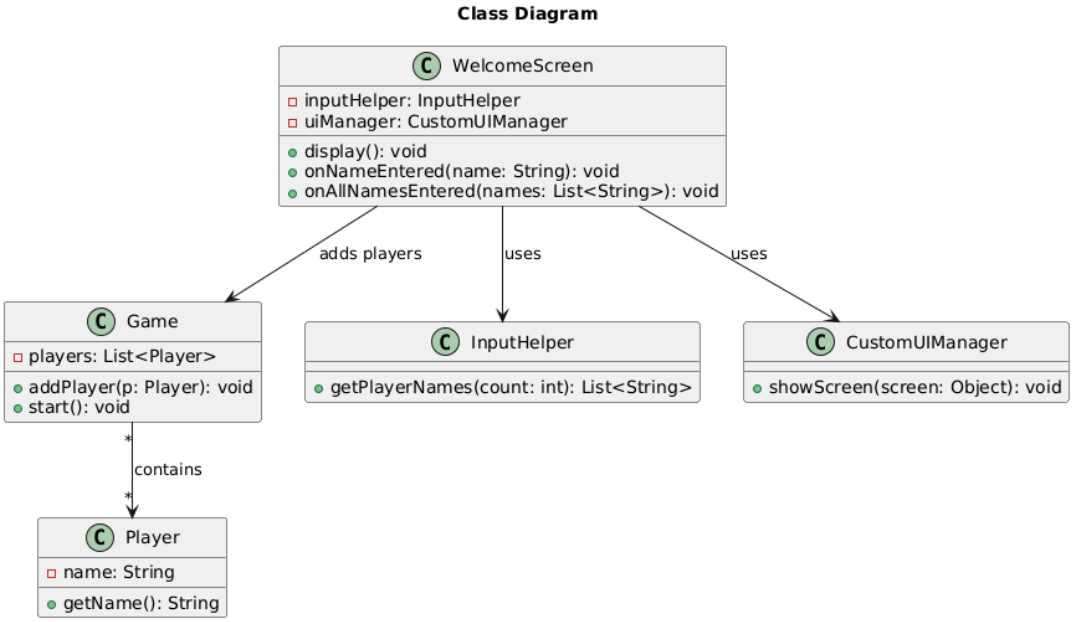
    - **Prompt loop**: `GameUI.registerPlayers(2)` → for each of the two players:
        - `JOptionPane.showInputDialog("Enter player name:")`
        - `new Player(name)` → `Game.getInstance().addPlayer(player)`
    - **Count enforcement**: exactly two prompts; the game will not start until both names are entered.
- **Console path**:
    - **Entry**: `Main.main()` → decides console mode → calls `Game.start()`
    - **Prompt loop**: inside `Game.start()` →
        - call `Console.promptForPlayerName(1)` and `Console.promptForPlayerName(2)` (or a simple `for` loop from 1 to 2)
        - each invokes `InputHelper.getText("Enter player name:")`
        - `new Player(name)` → added to `Game.getInstance().getPlayers()`
    - **Count enforcement**: two sequential prompts; game logic assumes two players are present before proceeding.
- **No name validation**: Names may be empty or blank; no trimming or character restrictions are applied at this stage.
- **Push-based I/O**: Both GUI and console paths “push” prompts directly where needed, keeping the looping and I/O logic localized in two methods (one for GUI, one for console).
- **Why**:
    - Enforce minimum player count at startup so that all downstream game rules (turn order, scoring, etc.) can assume a two-player context.
    - Keep prompting logic simple and isolated from core game flow.



Interpretation & Design Rationale

- **Orchestration**
  - `Main` remains the single entry point and decides GUI vs. console.
  - Exactly two name-entry calls live in `GameUI.registerPlayers(2)` and `Game.start()` → console prompts, making the startup flow obvious and predictable.
- **Responsibility separation**
  - **Game**: owns the `List<Player>` and the logic of "we need two players before starting."
  - **GameUI** & **WelcomeScreen**: handle Swing dialogs and the loop for two players.
  - **Console** & **InputHelper**: handle console prompts in a simple `for` loop.
  - **Player**: immutable holder of a name.
  - This division keeps UI concerns out of the game-logic classes, adhering to the Single Responsibility Principle.

- **Push-based notification**
    - No callbacks or event buses—each path directly invokes its I/O routine twice.
    - Simplifies prototyping and makes it trivial to swap in a different I/O mechanism later (e.g., networked registration).
- **Naming policy**
    - Still accepts names verbatim—no defaults, trimming, or fallbacks—so behavior remains predictable and future validation rules can slot in cleanly.



Structure

- **Class roles**
    - **Main**: selects mode and initiates registration.
    - **WelcomeScreen** → **GameUI**: prompts twice via Swing.
    - **Console** → **InputHelper**: prompts twice via console.

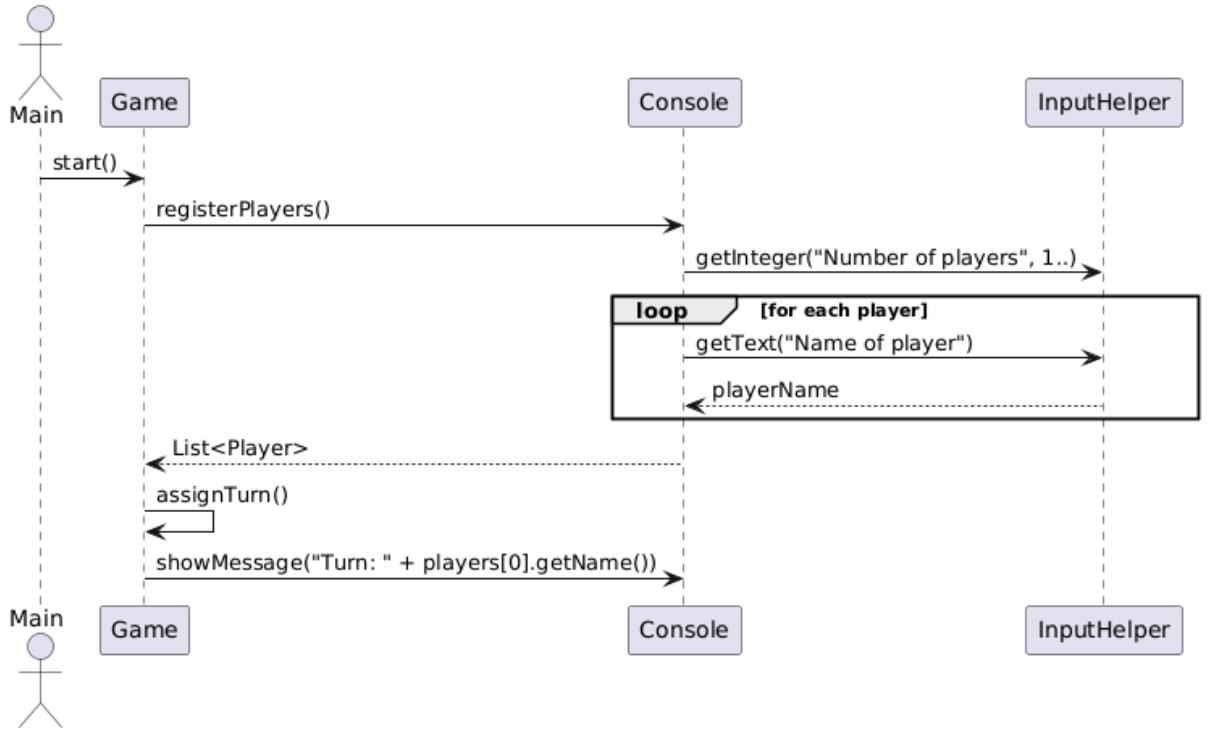
    - **Game**: singleton, holds a `List<Player>`; enforces that `players.size() == 2` before `start()` proceeds.
    - **Player**: simple name holder.
- **Associations**
    - `Main` → (`WelcomeScreen` → `GameUI` → `Game.addPlayer`)
    - `Main` → (`Game` → `Console` → `InputHelper` → `Game.addPlayer`)
    - `Game` "1" ── "*" `Player` (but for this sprint, usage is exactly two players)

3. **As a player, I want the game to assign me a turn automatically.**

Sprint 1

Initial design

- **Turn assignment** lives in the model: `Game.start()` registers players (via `Console`/`InputHelper`) then calls `assignTurn()`, which unconditionally picks `players[0]`.
- **Rotation**: `nextTurn()` simply advances `(currentPlayerIndex + 1) % players.size()`.
- **Push-based**: the model itself calls `Console.showMessage("Turn: …")`.
- **Why**: minimal logic to get the game running; no randomness, no UI framework, a clear single-responsibility split (game logic vs. console I/O).



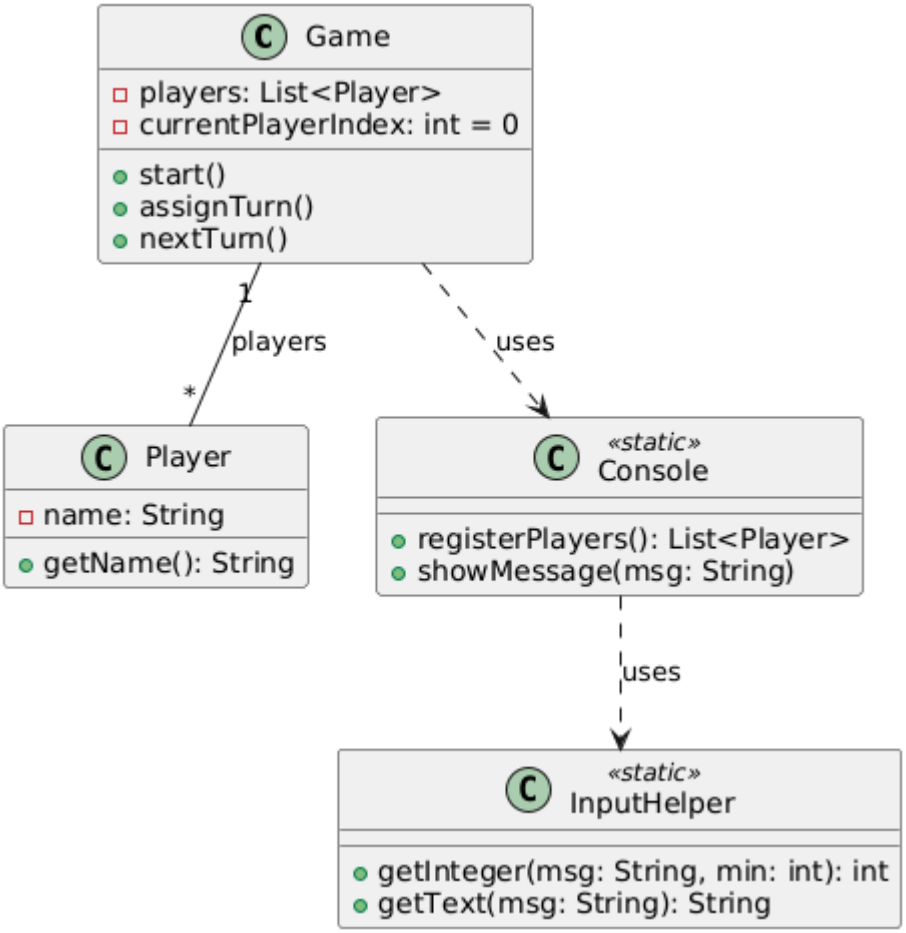
Interpretation & Design Rationale

- **Orchestration**: The flow begins in `Main` so the application entry point has full control over launching the game. This keeps startup logic out of lower-level classes.

- **Responsibility separation**: `Game` handles only game-flow logic (`registerPlayers()`, `assignTurn()`, `nextTurn()`), while `Console` and `InputHelper` manage all I/O and validation. This clear division adheres to the **Single Responsibility Principle**.

- **Push-based notification**: In this first iteration, `assignTurn()` directly calls `Console.showMessage(...)`. Although simple, this couples the model to the console; we accept that trade-off initially for rapid prototyping.

- **First-player policy**: Always selecting `players[0]` removes randomness complexity and guarantees a predictable start. This decision prioritizes simplicity over game fairness at this stage.



Structure & Design Rationale

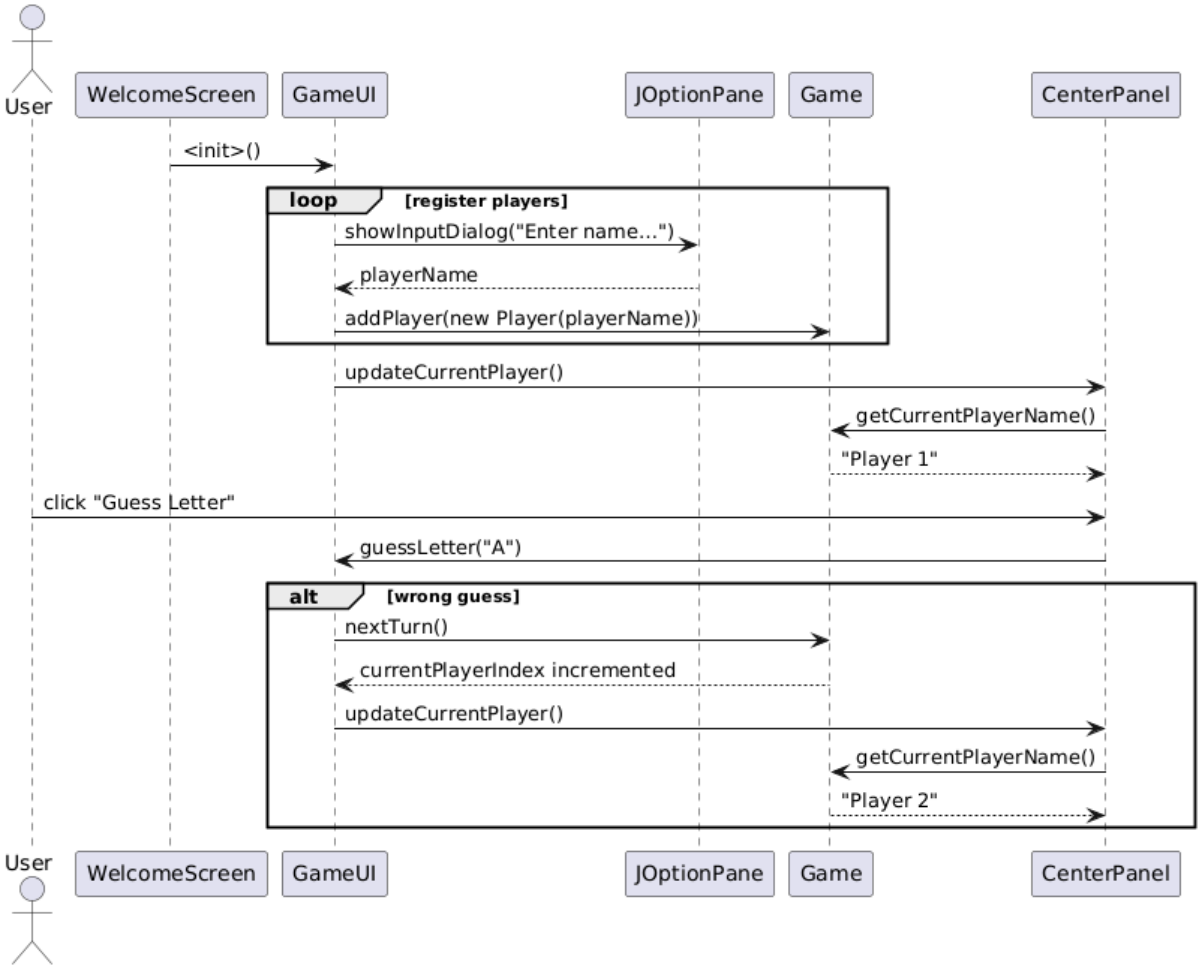
- **Class roles**:
  - `Game` encapsulates the turn-assignment logic and holds the list of `Player`.
  - `Player` is a simple data holder for the player's `name`.
  - `Console` and `InputHelper` are utility classes for user interaction, marked `<<static>>` because they have no instance state.
- **Associations**:
  - A one-to-many link (`1` — `*`) from `Game` to `Player` models the player roster.
  - Dependency arrows (`..>`) indicate that `Game` invokes `Console` methods, and in turn `Console` invokes `InputHelper`, but none own each other's data—this keeps coupling low.

- **Evolution groundwork**:
    - We start without design patterns to avoid premature complexity.
    - The current structure already separates concerns, paving the way for later refactorings (MVC, Singleton, Command, etc.).

Sprint 2

Swing UI & pull-based update

- **Model/UI separation**: introduce `GameUI` (with `TopPanel`/`CenterPanel`/`BottomPanel`) and move all console code out of the turn-assignment path.
- **Pull-based**: instead of `Game` pushing messages, the UI now calls `Game.getCurrentPlayerName()` to display “Turn: …” in a label.
- **First-player policy** remains (no random choice yet).
- **Why**: decouple presentation from logic, prepare for richer UX and future features.



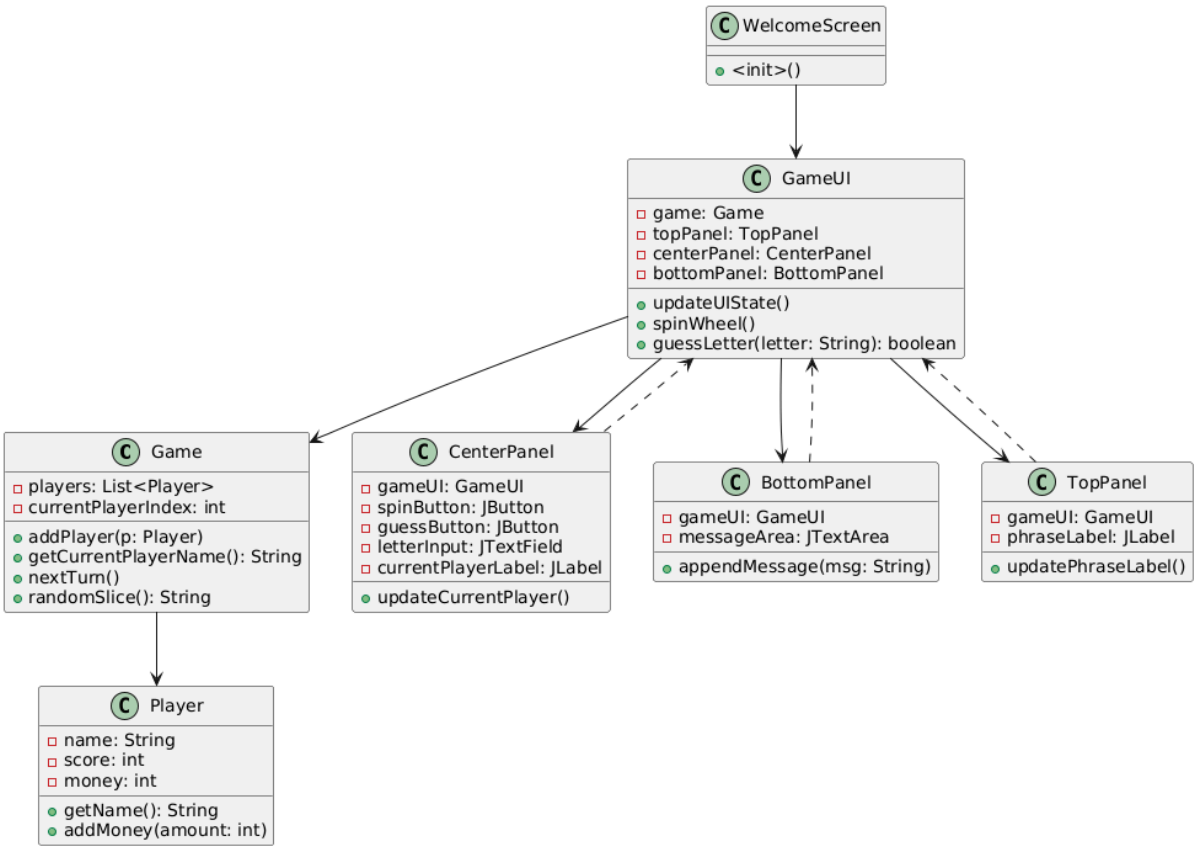
Interpretation & Design Rationale

1. **Decoupling UI from model**: Initialization and player registration move entirely into `GameUI`, which serves as a façade between Swing dialogs (`JOptionPane`) and the `Game` model. This shift exemplifies the **Model-View-Controller** principle: `Game` holds state and rules, `GameUI` handles orchestration, and panels (e.g., `CenterPanel`) render data.

2. **Pull-based turn display**: After registration, `GameUI` calls `CenterPanel.updateCurrentPlayer()`, which "pulls" the active player's name via `Game.getCurrentPlayerName()`. This avoids the model pushing messages to the UI, enhancing separation and testability.

3. **Event-driven flow**: User interactions (button clicks) drive the sequence. When the player guesses incorrectly, the UI again delegates to `Game.nextTurn()` and then re-queries the model for the next active player.

4. **Preparation for randomness**: Although not yet implemented, the presence of `randomSlice()` in `Game` (not shown in this sequence) foreshadows later addition of spin-wheel logic, without altering turn-assignment mechanics here.



Structure & Design Rationale

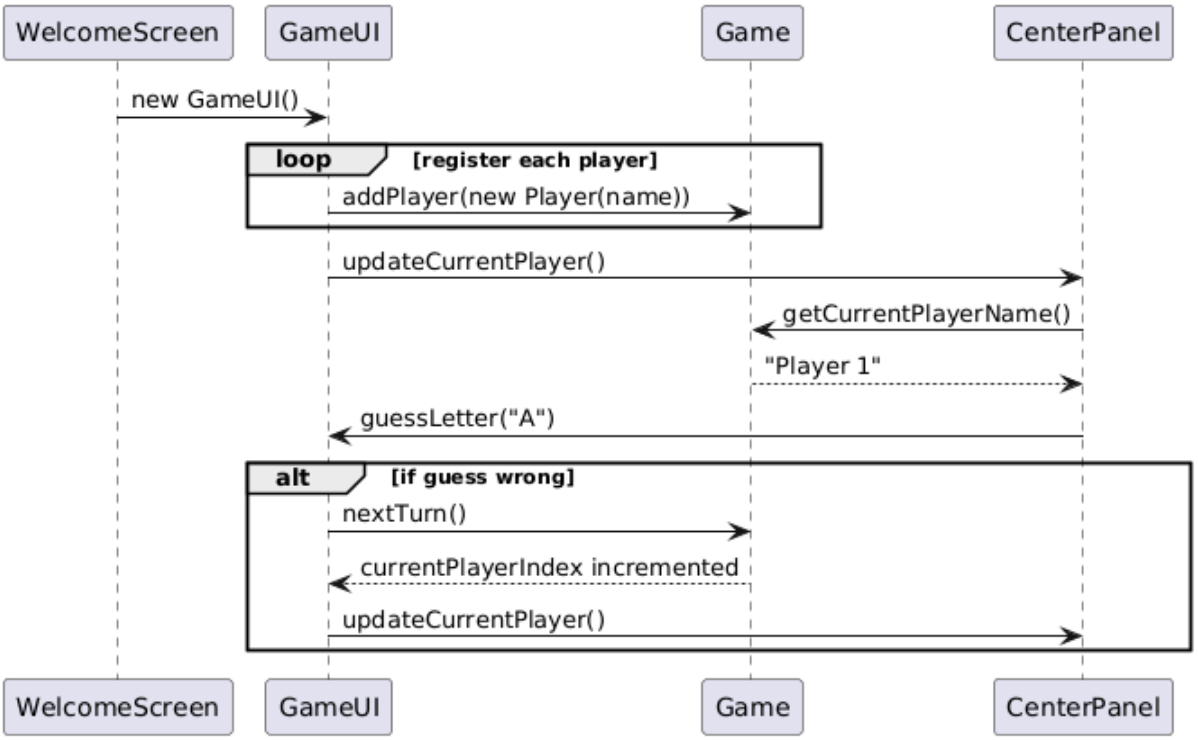
1. **MVC-like layering**:
    - **Model**: `Game` and `Player` encapsulate game state and rules.
    - **View/Controller**: `GameUI` plus panels manage user interactions and rendering.
    - **Support**: `WelcomeScreen` initializes the UI, keeping `Main` out of the details.
2. **Single Responsibility & Low Coupling**:
    - `Game` knows nothing about Swing; it only manages players and turn logic.
    - Each panel handles a specific UI concern: `CenterPanel` shows the current player, `BottomPanel` logs messages, `TopPanel` displays the phrase in play.
3. **Extensibility**:

- `randomSlice()` is already present, hinting at upcoming spin functionality without modifying existing associations.
- New UI components or features can be added by extending `GameUI` and its panels, without touching `Game`.

Sprint 3

Singleton model & pure pull

- **Singleton**: `Game.getInstance(...)` guarantees a single shared game state.
- **No more `assignTurn()`**: UI always “pulls” the active player on demand.
- **Legacy console code** remains but is no longer part of the main UI flow.
- **Why**: ensure one authoritative state, simplify turn logic by having the UI fully



in charge of refreshing.

Interpretation & Design Rationale

1. **Singleton access**: Rather than instantiating `Game` each time, `GameUI` holds a reference to the single `Game` instance (via `getInstance()`). This guarantees a unified game state across the entire UI without passing around `Game` as a parameter.

2. **Pull-based turn display**: Whenever the UI needs to show whose turn it is—both at startup and after a wrong guess—it explicitly asks `Game.getCurrentPlayerName()`. By removing any `assignTurn()` push from the model, we keep the control flow clear: the view decides *when* to refresh, and the model simply provides state.

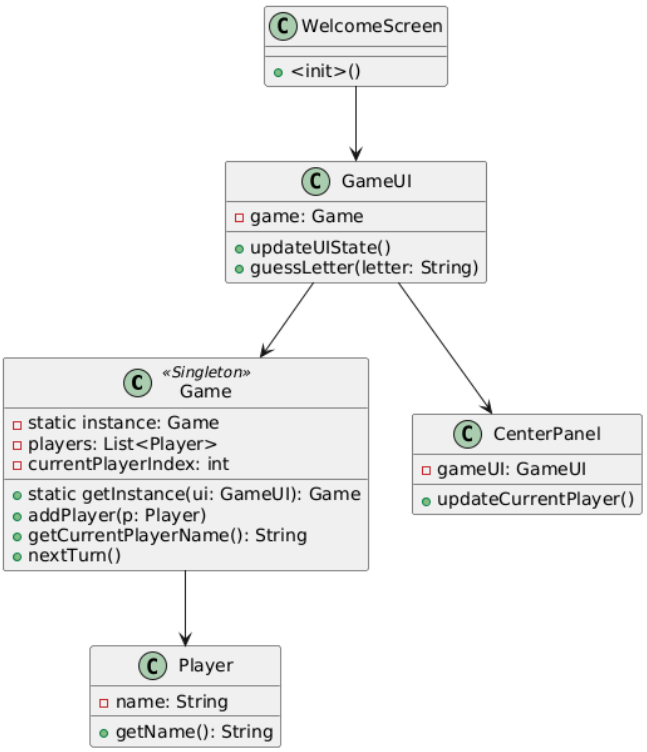
3. **SRP & low coupling**:

    - `WelcomeScreen` solely bootstraps the Swing window.

    - `GameUI` orchestrates registration and delegates turn logic to `Game`.

    - `CenterPanel` updates UI elements based on model queries.
      This separation makes each component easy to test in isolation.

4. **Evolution step**: We've eliminated direct console calls and deprecated the old `assignTurn()` method. The focus here is on UI-driven, pull-based updates using the Singleton `Game` as the canonical source of truth.



Structure & Design Rationale

- **Singleton pattern** on `Game`: The `static instance` field and `getInstance(...)` method enforce a single shared game state, simplifying access from anywhere in the UI without passing references.
- **Clear responsibilities**:
    - `Game` holds only the list of players and the logic to advance turns (`nextTurn()`).
    - `GameUI` manages high-level orchestration: initializing the game, handling user commands (e.g., `guessLetter(...)`), and invoking `updateUIState()`.
    - `CenterPanel` is responsible exclusively for refreshing the display of the current player's name.
    - `Player` remains a simple data carrier for the player's `name`.
    - `WelcomeScreen` bootstraps the UI.
- **Loose coupling**:

- Dependencies flow *toward* `Game` rather than vice versa. The model (`Game`) knows nothing about Swing or panels.

- UI classes depend on the `Game` interface but not on each other beyond necessary interactions.

- **Preparation for future patterns**: By introducing the Singleton now, subsequent sprints can layer additional patterns (e.g., Command, Strategy, Memento) on top of this stable, centralized model.

Sprint 4

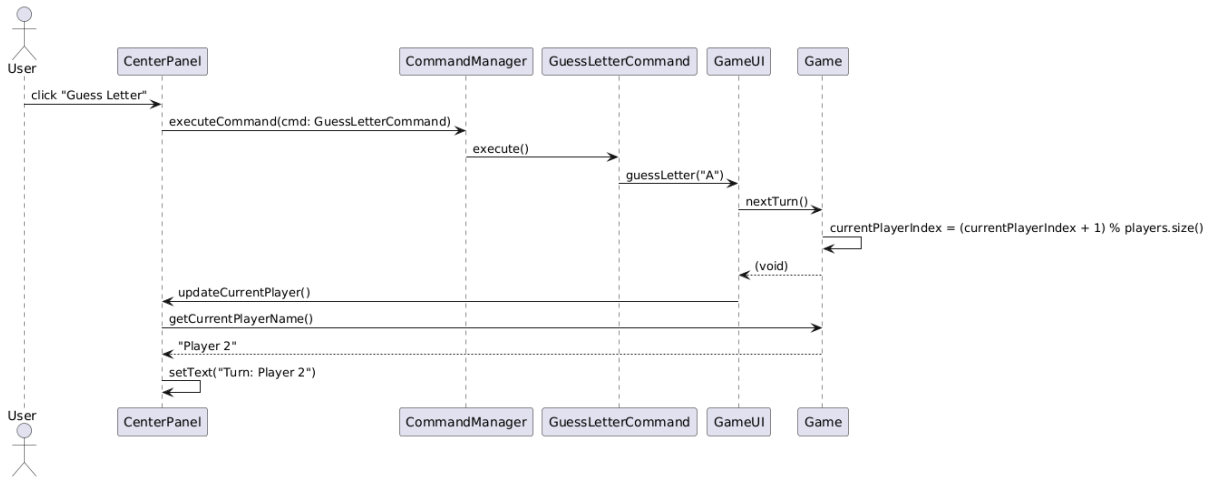
Command pattern integration

- **`Command` + `CommandManager`**: every user action (spin, guess, buy vowel, solve) becomes a command object.

- **Encapsulated turn logic**: each command's `execute()` invokes `Game.nextTurn()` when needed, then UI calls `updateUIState()`.

- **Undo support**: built-in via the `Command` interface (not yet used for turn but ready).

- **Why**: decouple UI events from business rules, open the door to undo/redo and easier testing.



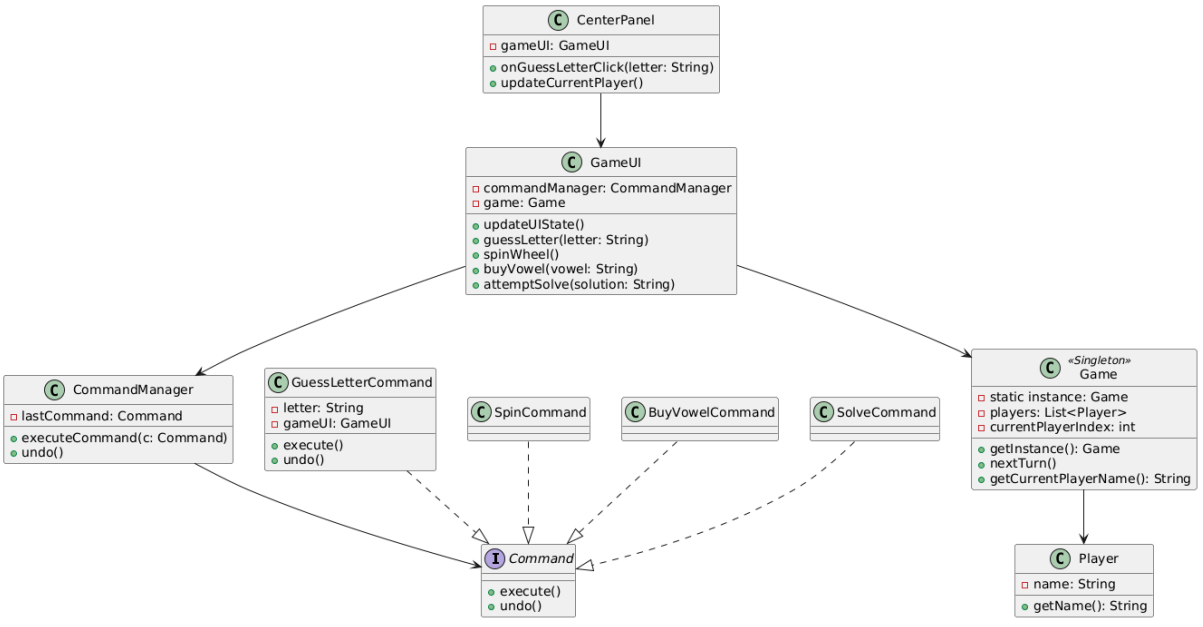
Interpretation & Design Rationale

- **Encapsulation of user actions**: When the user clicks “Guess Letter,” the `CenterPanel` doesn’t call `Game` directly but instead creates a `GuessLetterCommand` and hands it to the `CommandManager`. This isolates UI code from game-logic details, adhering to the **Open/Closed Principle**: new commands can be added without modifying existing UI classes.

- **Command execution and turn logic**: Inside `GuessLetterCommand.execute()`, the command delegates to `GameUI.guessLetter()`, which in turn invokes `Game.nextTurn()` if the guess is wrong. This layered approach keeps the turn-advancement logic (`nextTurn()`) centralized in the `Game` model, preserving **Single Responsibility** across classes.

- **Pull-based UI update**: After state changes, `GameUI` instructs `CenterPanel` to `updateCurrentPlayer()`, which queries `Game.getCurrentPlayerName()` and updates the label. The model never pushes UI updates, maintaining a clear separation of concerns.

- **Undo readiness**: Although not shown here, each `Command` implements an `undo()` interface, laying the groundwork for future “undo” functionality without entangling the model or UI in rollback details.



Structure & Design Rationale

- **Command pattern**: The `Command` interface and `CommandManager` decouple UI events from business logic. Each concrete command class (`GuessLetterCommand`, `SpinCommand`, etc.) encapsulates its own execution and undo logic, supporting **Open/Closed Principle** and potential undo/redo features without modifying the core `Game` or UI classes.

- **Singleton model**: `Game` is marked `<<Singleton>>`, ensuring a single source of truth for turn state across the application. The static `getInstance()` method centralizes access and prevents accidental multiple game instances.

- **Clear separation of concerns**:

    - `Game` holds only player data and turn-advancement logic.

    - `GameUI` orchestrates command execution and overall UI refresh (`updateUIState()`).

    - `CenterPanel` focuses exclusively on handling the “guess letter” button and displaying the current player.

- **Low coupling**: Arrows flow toward abstractions; the `Game` model does not depend on UI classes, and UI classes rely only on the `Command` and `Game` interfaces. This structure maximizes testability and maintainability.

Sprint 5

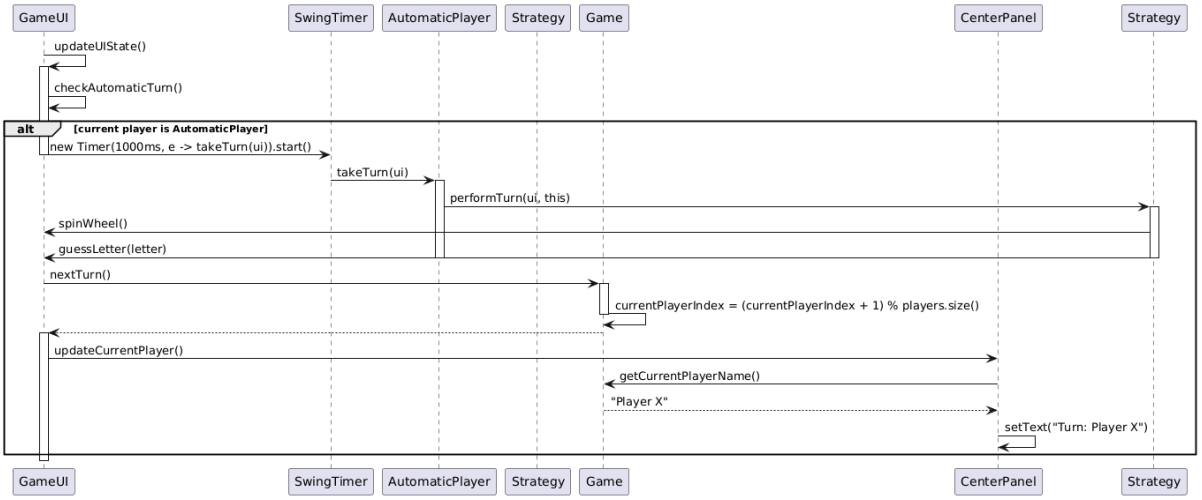
Automated players & Strategy pattern

- **`AutomaticPlayer`** subclass of `Player` plus `AutomaticPlayerStrategy` (Easy/Medium/Hard).

- **Detection**: after `updateUIState()`, `GameUI` calls `checkAutomaticTurn()`. If the current player is AI, a 1 s `Timer` triggers `takeTurn()`.

- **Strategy** drives calls to `spinWheel()` and `guessLetter()`, which still delegate to `Game.nextTurn()`.

- **Why**: extend turn assignment to non-human actors without polluting core logic; easily swap or add new AI behaviours.



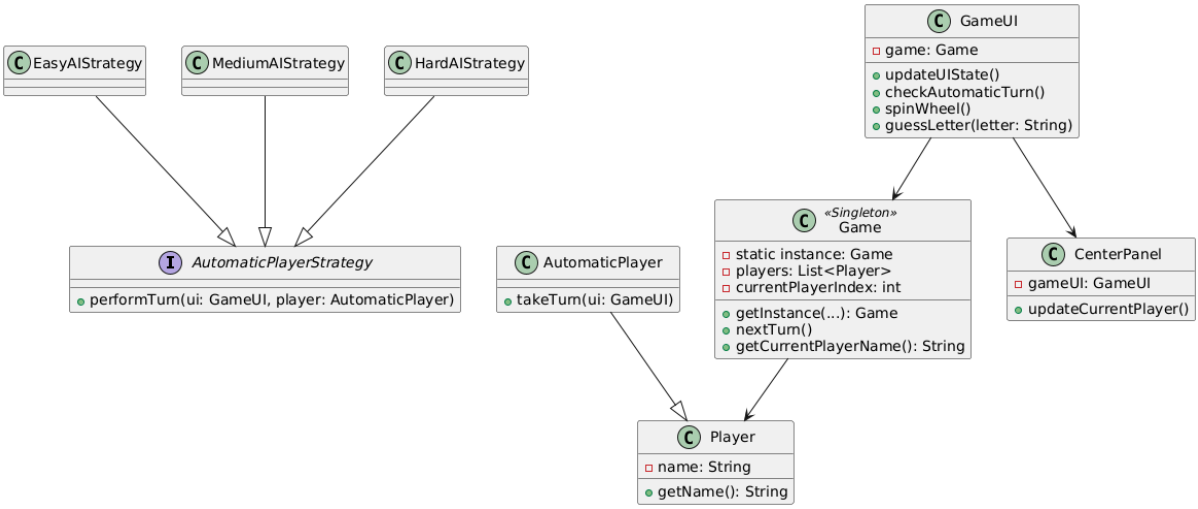
Interpretation & Design Rationale

1. **Automatic-turn detection**: After every UI refresh (`updateUIState()`), the `GameUI` invokes `checkAutomaticTurn()`, which inspects if the current player is an `AutomaticPlayer`. This pull-based check ensures that human and AI flows can share the same update path.

2. **Delayed AI action**: Launching a Swing `Timer` with a 1 s delay gives the player a moment to see the UI update before the AI "moves," improving the user experience.

3. **Strategy delegation**: `AutomaticPlayer` delegates decision-making to its `AutomaticPlayerStrategy` (Easy/Medium/Hard), which encapsulates the choice of spin and guess without polluting the player or UI classes. This honours the **Open/Closed Principle** by allowing new AI behaviours to plug in without modifying existing code.

4. **Consistent turn advancement**: Regardless of human or AI, the turn always advances via `Game.nextTurn()`, and the UI then re-queries the singleton `Game` for `getCurrentPlayerName()`. The model remains the single source of truth, preserving **Single Responsibility** in both model and UI.



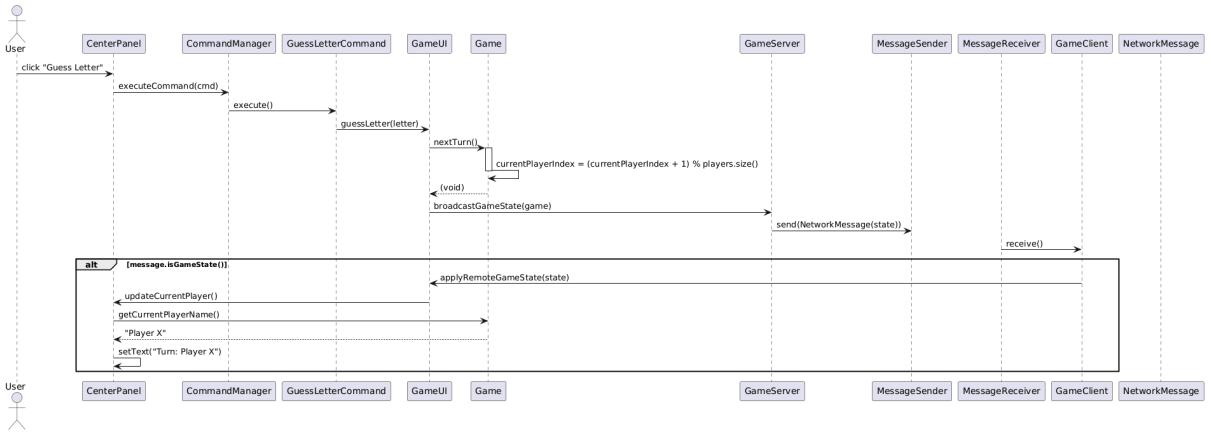
Structure & Design Rationale

- **Strategy pattern**: The `AutomaticPlayerStrategy` interface and its three implementations decouple AI logic from both `AutomaticPlayer` and `GameUI`, allowing you to introduce new difficulty levels without altering existing classes (OCP).
- **Singleton model**: `Game` remains a thread-safe singleton, centralizing turn state so both human and AI flows refer to the same `currentPlayerIndex`.
- **MVC continuity**:
    - *Model* (`Game`, `Player`, `AutomaticPlayer`) contains only game rules and state.
    - *View/Controller* (`GameUI`, `CenterPanel`) drives the interaction and rendering.
- **Single Responsibility & Low Coupling**:
    - `AutomaticPlayer` is responsible only for deciding *when* to play, not *how*—that's the strategy's job.
    - UI classes never contain game logic; they simply orchestrate calls to the model and strategies.

Sprint 6

Persistence & client-server sync

- **Memento pattern**: `GameOriginator` captures `currentPlayerIndex` (and full game state) into a `GameStateMemento`; `GameCaretaker` saves/loads it.
- **Multiplayer**: host (`GameServer`) broadcasts the serialized `GameState` after each `nextTurn()`; clients apply it and then pull `getCurrentPlayerName()` to refresh.
- **Singleton thread-safe** remains the source of truth.
- **Why**: ensure that when you load or share a game, the correct player "in turn" is restored and synchronized across all clients.

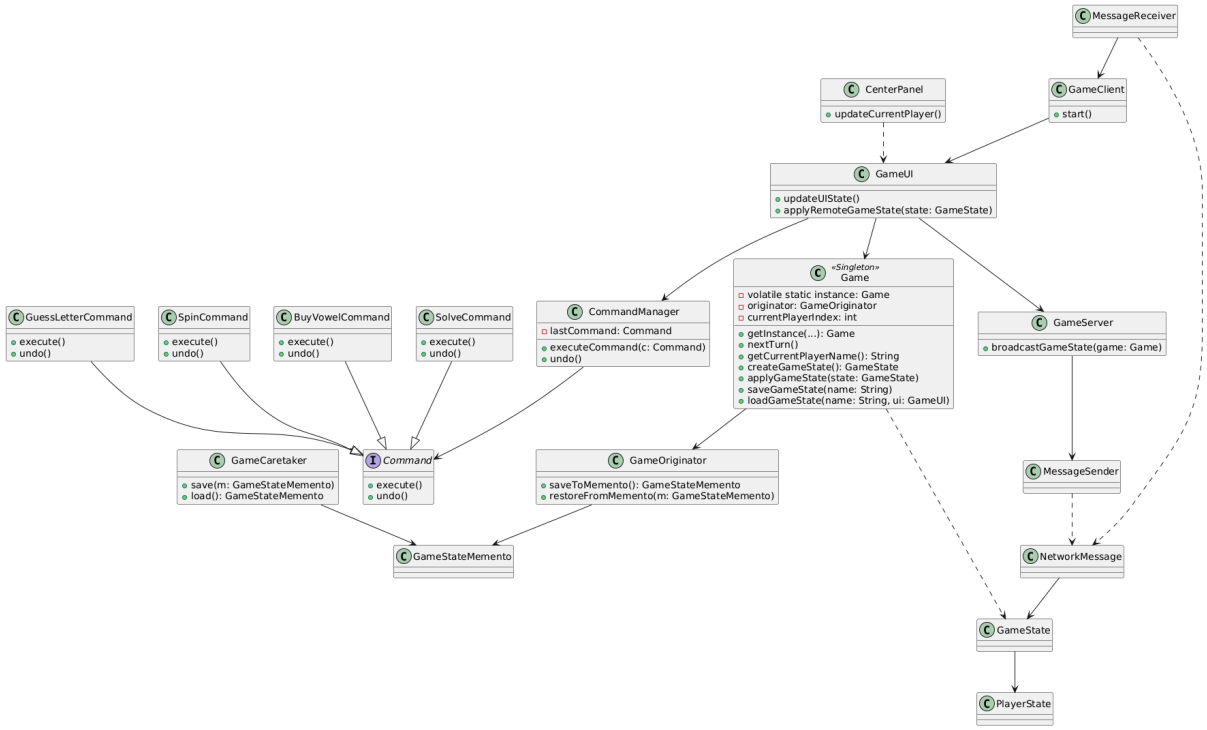


Interpretation & Design Rationale

1. **Command encapsulation remains**: A wrong guess is wrapped in `GuessLetterCommand`, executed by `CommandManager`, preserving the decoupled command infrastructure from Sprint 4.
2. **Turn advancement**: Within `Game.nextTurn()`, the singleton `Game` updates `currentPlayerIndex`. Keeping turn logic in the model ensures consistency and single responsibility.
3. **Host-driven state broadcast**: Immediately after updating the turn, `GameUI` (on the host) calls `broadcastGameState()`. This serializes the full `Game` state—including `currentPlayerIndex`—into a `NetworkMessage`, which `MessageSender` dispatches to all clients.
4. **Client synchronization**: Clients run a `MessageReceiver` loop; upon receiving a `NetworkMessage` containing a `GameState`, the client's `GameUI` applies it via `applyRemoteGameState()`. This restores the singleton model

(via Memento Pattern) on the client to exactly the host's state.

5. **Pull-based UI update**: After applying remote state, the client's `GameUI` invokes `CenterPanel.updateCurrentPlayer()`, which queries `Game.getCurrentPlayerName()` to display the correct player turn. Even in multiplayer, the UI never relies on pushes from the model—maintaining clear, testable flows.



Structure & Design Rationale

- **Memento Pattern**:
    - `GameOriginator` and `GameStateMemento` encapsulate state capture/restoration, ensuring that when `applyGameState()` is called (either on load or via network), `currentPlayerIndex` is correctly restored along with all other game data.
    - `GameCaretaker` handles persistence, isolating file I/O from both model and UI.

- **Singleton (thread-safe)**: `Game` remains the single source of truth. Its `getInstance()` method ensures that host and all clients operate on one authoritative game object, avoiding split-brain scenarios.

- **Command Pattern**: Commands continue to decouple UI events from game logic, supporting potential undo/redo even in a networked context.

- **Client-Server Synchronization**:

    - `GameServer.broadcastGameState()` packages the `GameState` into a `NetworkMessage`.

    - `MessageSender` and `MessageReceiver` manage low-level transport, while `GameClient` and `GameUI.applyRemoteGameState()` restore state at the application layer.

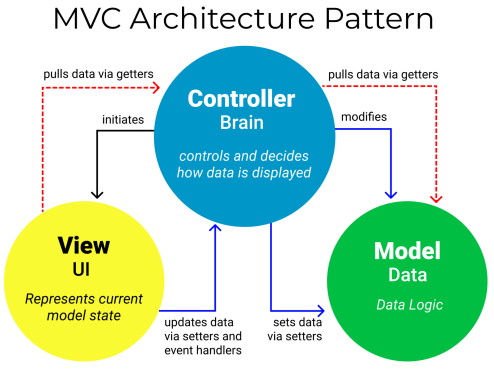
- **Pull-based UI**: UIs simply call `getCurrentPlayerName()` after state changes or synchronization, maintaining a unidirectional dependency flow (UI → model), which simplifies testing and maintenance.

4.Design Patterns

Pattern	Application	Effect
MVC	GameUI vs. Game logic	Separates UI from core game data and control
Singleton	Game class	One global game instance, centralizing state
Memento	GameOriginator, GameStateMemento, GameCaretaker	Allows saving/restoring game state without exposing internal details
Strategy	Player/MoveStrategy	Swappable human/AI and AI difficulty behaviors

Model View Controller

For the implementation of our project we chose to use the Model View Controller (MVC) design pattern. The MVC breaks the code into 3 main components (mode, view and controller) to make the app easier to maintain and expand. The Model is the one in charge of handling data, the View is



what is seen by users, being the controller the component used to connect the other two. The reason why we chose this design pattern is because we found really useful the way it divides front and back end programming.
Be that as it may, the implementation of the MVC was not done flawlessly. As this task was planned to be done late in the sprints, we did not have time to implement the pattern in its entirety.

Singleton

We adopted the Singleton pattern for our Game class to guarantee:

- One-and-only instance: Ensures all parts of the application—UI rendering, event handling and configuration—share the exact same Game object.

- Global access point: Eliminates the need to pass Game references around, simplifying method signatures and reducing boilerplate.

- Lazy initialization: The Game instance is created only when first needed, saving startup time and resources.

- Thread-safe instantiation: Double-checked locking around the synchronized block minimizes locking overhead while preventing concurrent creation.

- Centralized state management: Provides a clear, single location to manage windows, audio, settings and game logic, improving maintainability and cohesion.

Memento

We adopted the Memento pattern for our Game class to provide:

- State snapshots & restore
  Allows users to save the entire game state (players, current word, revealed/used letters, turn index) and reload it later or implement undo-like features.

- Encapsulation of state
  GameStateMemento hides the internal GameState structure, so the Game class doesn't expose or manipulate raw state objects directly.

- Separation of concerns

    - Originator (GameOriginator) handles the mechanics of creating and restoring snapshots.

    - Memento (GameStateMemento) holds the snapshot data.

    - Caretaker (GameCaretaker) deals exclusively with persistence (JSON serialization/deserialization).

- Flexible persistence
  By using Gson, deep copies are trivial, and the caretaker can easily switch file paths or formats without touching game logic.

- Maintainability & extensibility
  Adding new fields to GameState or supporting multiple save slots requires minimal changes—usually only to GameState and possibly the caretaker's file logic.

- Testability
  You can independently test snapshot creation and loading mechanics by exercising GameOriginator and GameCaretaker without involving UI or gameplay loops.

Strategy

The Strategy pattern defines a family of interchangeable algorithms, encapsulates each one behind a common interface, and allows the client to select the algorithm at runtime without altering its code. In our Wheel of Fortune implementation, the *Player* interface serves as the high-level strategy interface, declaring a *makeMove(GameState)* operation. Two concrete strategy families implement this interface:

1. Human vs. AI
   - Human player: Obtains its move by querying the user (through console or GUI).
   - Automatic player: Delegates move selection to a nested strategy.

2. AI Difficulty Levels
   Within *AutomaticPlayer*, we introduce a second strategy interface, *MoveStrategy*, whose *selectMove(GameState)* method encapsulates the algorithm for choosing letters. Three concrete *MoveStrategy* implementations realize progressively sophisticated behaviors:
   - EasyStrategy: pure random consonant picks
   - MediumStrategy: frequency-weighted random picks (bias toward common letters)
   - HardStrategy:full remaining-letter frequency analysis (always picks the highest-frequency unused consonant, buys a vowel if none left)

We chose this pattern because it cleanly separates what the game does (driving the play loop) from *how* a move is chosen, and it lets us vary that decision logic at runtime without touching the core game code.