Phase 3: Testing

Features to be Unit-Tested

I. Controller and Runtime Flow

1) **Input-driven movement pipeline** – `GameController.tryPlayerMove` only allows moves while the `GameState` is running, routes moves through `Board.step`, applies the resulting collectible, and handles collisions or blocked moves differently. Tests should assert that each `MoveResult` branch triggers the right scoreboard and view side-effects, especially the win/lose transitions invoked from `evaluateEndStates` and `onTick`.

2) **Game loop ticking** – `GameController.onTick` advances the `Board`, gives the `Spawner` a chance to spawn bonuses, and checks for enemy catches vs. end-state evaluation. Unit tests can simulate ticks with mocked board summaries to ensure state transitions, timer usage, and repaint requests occur at the right times.

3) **Runtime state tracking** – `GameState` encapsulates status, tick count, and entity positions, and automatically stops the `Scoreboard` when switching to `WON`/`LOST`.

II. Board and Cell Mechanics

1) **Board construction and terrain ownership** – `Board` consumes a `BoardGenerator.Output` to create `Cell` instances, seeds the player at the start, and keeps authoritative lists of enemies/rewards. Tests should assert that `cellAt`, `isInBounds`, and registration helpers maintain consistent terrain and occupant references.

2) **Movement resolution** – `Board.step` enforces bounds, terrain walkability, per-character entrance rules, manages occupants, and returns `MoveResult.MOVED`/`BLOCKED`/`COLLISION`. Unit tests can cover player vs. enemy moves, collisions, and wall/barrier interactions.

3) **Tick processing** – `Board.tick` iterates enemies, checks for player capture, and expires temporary bonuses via `BonusReward.onTickAndAlive`. Tests should verify that expired bonuses are removed and that `TickSummary.playerCaught` flags catches accurately.

4) **Collectible extraction** – `Board.collectAt` removes any item from a cell and prunes the corresponding tracking list, so tests must ensure optional and required rewards (and punishments) are removed consistently.

5) **Cell occupancy rules** – `Cell.isEnterableFor`, `addOccupant`, and `symbol()` enforce single-player/single-enemy limits, keep start/exit enemy-free, and render priority order. Focus tests on enforcing terrain constraints and occupant conflicts.

III. Board Generation and Helper Logic

1) **Barrier-mode permutations** – `BoardGenerator.generate` dispatches to `NONE`/`PROVIDED`/`TEXT`/`RANDOM` algorithms, each of which has unique validation and setup paths (perimeter walls, resource parsing, random placement constrained by Chebyshev distance). Unit tests should cover each branch, particularly error handling for malformed text files and insufficient board sizes.

2) **Random barrier validation** – The `RANDOM` branch relies on `GeneratorHelper.isBarrierConfigurationValid` to ensure reachability, no isolated cells, and start/exit accessibility while iteratively placing barriers. Tests can feed crafted wall/barrier arrays to this helper to ensure invalid states are rejected.

3) **Reusable helper utilities** – `GeneratorHelper` supplies BFS counting, perimeter-wall generation, random start/exit positioning, and text resource parsing; these utilities impact every generated board, so unit tests should validate edge cases like minimal board sizes, inconsistent maps, and isolated floors.

IV. Spawning

1) **Bonus quota scheduling** – `Spawner.spawnBonusRewards`, `onTick`, `notifyBonusCollected`, and `scheduleNextBonusSpawn` enforce quota counts, spawn/life timing, and capacity validation. Tests should simulate varying tick intervals to ensure quotas decrease only after collection and spawn windows honor `spawnMinSec`/`spawnMaxSec`.

2) **Regular reward placement** – `spawnRegularRewards` demands enough free cells and shuffles placement, so test the exception path when capacity is insufficient and verify that each reward is registered onto the board.

3) **Punishment safety checks** – `spawnPunishments` removes start/exit positions, tests reachability to exit and every regular reward before placing, and keeps track of blocked cells. Unit tests should mock small boards to ensure punishments never seal off required paths.

4) **Enemy spawn spacing** – `spawnEnemies` filters out cells within Chebyshev distance <3 from start/exit and enforces move-period injection. Tests must verify filtering logic and that `Board.registerEnemy` is invoked the expected number of times.

5) **Spawner utilities** – `SpawnerHelper.freeFloorCells` and `canReach` provide deterministic cell listings and BFS reachability, critical to spawning correctness; they warrant tests covering occupancy filtering and blocked-set behavior.

V. **Collectibles** and **Scoring**
1) **Scoreboard accounting** – `Scoreboard` tracks score, required collectibles remaining, and elapsed time, with dedicated methods for required, optional, and penalty pickups plus lifecycle timing. Tests can validate value clamping, timer behavior, and the "required remaining" countdown.
2) **Bonus reward lifetimes** – `BonusReward` has constructors for persistent vs. timed bonuses and exposes `onTickAndAlive`. Unit tests should cover countdown behavior reaching zero and staying alive when lifetime <= 0.
3) **Punishment value safeguards** – The `Punishment` constructor clamps penalties to non-positive and exposes the symbol used for ASCII rendering, which should be verified to avoid accidental positive "punishments."

VI. **Enemy** Behavior
1) **Enemy move cadence** – `MovingEnemy.tick` enforces a cooldown (move period) while delegating to `Enemy.tick`, so tests can check that enemies skip the appropriate number of ticks and then move.
2) **Greedy pathing** – `MovingEnemy.decide` calculates direction priority based on horizontal vs. vertical distance, tries legal moves in order, and falls back to null when stuck. Unit tests should cover approach ordering, tie cases, and respecting board bounds/walkability.

VII. **User Interface**
1) **GamePanel render-mode controls and painting pipeline** – `GamePanel` exposes `setRenderMode`/`toggleRenderMode` and paints two distinct pipelines (image sprites vs. ASCII glyphs) when `paintComponent` iterates every cell, layering terrain, collectibles, enemies, and the player while respecting explosion overlays. Tests can simulate a stub `Board`/`Cell` matrix to assert that toggling modes triggers repaints and that the draw order honors the explosion short-circuit and item layering rules.
2) **ASCII symbol/color mapping** – In `SYMBOL` mode, the panel selects foreground colors based on the `Cell.symbol()` character, so a regression could easily leave actors invisible. Snapshot-based tests (using a fake Graphics2D or verifying the switch table) should cover each branch in the switch (sym) mapping to ensure every symbol (player, enemies, rewards, punishments, walls, start/exit, etc.) stays associated with its intended color.
3) **Game-over banner overlay** – When `onGameOver` stores a message, the painter computes board dimensions and draws a translucent rounded rectangle plus centered text over the playfield. UI tests can verify that a banner becomes visible only when non-blank and that its geometry centers relative to the board size (important when changing tile counts).
4) **HUD text formatting** – The `HUD` renders score, remaining required collectibles, elapsed time, and the current `GameState` status using dynamic layout calculations. Tests should assert that `paintHud` aligns these strings correctly (e.g., score on the left, time centered, status right-aligned) and that it pulls the expected values from `Scoreboard`/`GameState`.

5) **Image asset loading and error signaling** – `loadImage` resolves resources from the **classpath** and throws a descriptive exception when assets are missing. Unit tests can inject a **ClassLoader** with known paths (or stub the method) to ensure the method throws **IllegalStateException** for absent resources and wraps IO failures consistently.

6) **Game wiring/bootstrap logic** – The **App** class wires the entire MVC stack: it configures board size, reward counts, spawn timing, constructs the **Board** via **BoardGenerator**, seeds the **Spawner**, initializes **Scoreboard** / **GameState**, and finally starts **GameController**. Smoke-style tests (e.g., using dependency injection or verifying mocked collaborators) can ensure each configuration path (**NONE**/**PROVIDED**/**TEXT**/**RANDOM** barrier modes) produces a consistent board, that spawning calls honor the configured counts, and that the controller is started exactly once.

Important Interactions Between Different Components

1. **Input→model→score pipeline** (**GameController** ↔ **Board**/**Scoreboard**): **GameController** binds **GamePanel** keystrokes to **tryPlayerMove**, calls **Board.step(...)**, processes the resulting **MoveResult**, and applies collectibles by mutating the shared Scoreboard before repainting the view. This interaction ensures that physics, collision, scoring, and UI refresh are tied to actual player inputs.

2. **World tick orchestration** (**GameController** ↔ **Board** ↔ **Spawner**): Every Swing-timer tick, the controller advances the **Board**, which moves enemies and ages timed bonuses, then delegates to the **Spawner** to decide whether to place new bonus rewards. Tick summaries feed back into the controller to trigger lose conditions, so timing, enemy AI, and bonus spawning are coupled in one loop.

3. **End-state handling & banner UI** (**GameController** ↔ **GamePanel**): Win/lose evaluations combine **Scoreboard** state and **Board** positions; when outcomes occur the controller stops time, optionally records explosions, and asks the **GamePanel** to show banner text before repainting. This connects logical victory/defeat to what the player sees.

4. **HUD data binding** (**GamePanel** ↔ **Scoreboard**/**GameState**): The **HUD** reads live score, required collectibles, elapsed time, and status directly from the shared **Scoreboard**/**GameState**, so controller lifecycle calls (start/stop) immediately affect **HUD** labels. This is the main flow of runtime data into the **UI** without extra adapters.

5. **Spawner safety guarantees** (**Spawner** ↔ **Board**): The **Spawner** uses helper reachability checks and board metadata to place collectibles, punishments, and enemies while preserving walkable paths and respecting Chebyshev distance constraints. Because it mutates the board directly, correctness relies on its coordination with **Board**'s registration helpers.

Integration Tests to Cover These Interactions
1. **"Move and collect" pipeline test** → Covers interaction #1
   Instantiate real `Board`, `Scoreboard`, `GameController`, and a spy `GamePanel`.
   Simulate a key binding action moving the player onto a regular reward.
   Assert `Board.step` moved the player, `Scoreboard.requiredRemaining()`
   decremented, and the view recorded a repaint. This exercises the input-to-model-to-UI
   chain.

2. **Collision-induced loss test** → Covers interaction #1 and #3
   Place an enemy adjacent to the player, invoke the bound action toward it, and verify the
   controller switches the `GameState` to `LOST`, calls `Board.setExplosion`, and
   `GamePanel.onGameOver` with the expected banner.

3. **Negative-score fail-safe test** → Covers interaction #1 and #3
   Feed a punishment collectible via `collectAt` to push the score below zero, call
   `evaluateEndStates()`, and ensure the controller stops the timer and banner text
   shows "Score below zero!".

4. **Tick-driven catch test** → Covers interaction #2
   Configure a moving enemy that reaches the player after `board.tick(...)`; call
   `onTick` repeatedly and assert the controller processes
   `TickSummary.playerCaught()` to trigger `lose(...)`.

5. **Bonus spawn cadence test** → Covers interaction #2
   Configure the `Spawner` with short spawn/lifetime windows, run several controller ticks,
   and assert bonuses only appear after previous ones are collected (via
   `notifyBonusCollected`) and disappear when lifetime expires.

6. **HUD lifecycle test** → Covers interaction #4
   Start the controller, wait a short duration, stop it, and render the `HUD` once; verify "Time"
   advances only while running and status text flips between `RUNNING`/`WON`/`LOST` per
   `GameState`.

7. **Win-condition UI test** → Covers interaction #1 and #3 and #4
   Collect all required rewards, move the player onto the exit, and assert the controller calls
   `view.onGameOver(...)`, `scoreboard.stop()`, and the `HUD` switches to `WON`.

8. **Punishment placement safety test** → Covers interaction #5
   Generate a dense board, run `spawnPunishments`, and then attempt pathfinding from
   start to every reward/exit to ensure reachability still holds, confirming `Spawner`
   respected board constraints.

9. **Enemy spawn distance test** → Covers interaction #5
   After invoking `spawnEnemies`, verify no enemy was registered within Chebyshev
   distance < 3 of start or exit, ensuring board safety logic is honored.

Findings + Changes Made