Functional Tests Plan

The following high-level functional tests exercise the gameplay loop, `controller`/`runtime` logic, `board` mechanics, `collectible`s, `enemy` behavior, `spawn`ing subsystems, `board` generation helpers, `render`ing/`UI` layers, and bootstrap wiring. Each test lists the target features plus acceptance criteria.

I. `Controller` and `Runtime` Flow

1. `Player` movement gating and resolution
   Setup: Launch a level with accessible floor tiles, a start position, one `enemy`, and one `collectible`.

   Steps: Attempt `player` input while the game is paused (expect rejection). Resume, move into an empty tile, a tile with a `collectible`, an `enemy` tile, and a wall.

   Assertions – `` `GameController.tryPlayerMove` ``:
   - ignores input unless status is `RUNNING`,
   - updates the `board` via `` `Board.step` ``,
   - applies `collectible`s to the `scoreboard`,
   - reports collisions as loss, and
   - keeps position unchanged for blocked moves.

   Win/lose transitions trigger the correct status and `scoreboard` stop.

2. Tick loop sequencing
   Setup: Configure a deterministic `board` and mocked `` `Spawner` `` to observe tick calls.

   Steps: Advance several ticks via `` `GameController.onTick` ``.

   Assertions – Each tick:
   - requests a `board` advance,
   - invokes `` `Spawner.onTick` ``,
   - evaluates `` `TickSummary` `` (`player` caught),
   - toggles the timer, and
   - requests a repaint only when state changes.

3. `GameState` lifecycle invariants
   Setup: Programmatically mutate `` `GameState` `` (status changes, position setters, `enemy` placement).

   Assertions: Positions are never null, `scoreboard` stops when status transitions to `WON`/`LOST`, and tick counters increment atomically.

II. `Board` and `Cell` Mechanics

4. `Board` construction and addressing
   Validate `` `BoardGenerator.Output` `` consumption: `` `Board.cellAt` ``, `` `isInBounds` ``, and occupant registries match expected terrain and positions after initialization.

5. Movement resolution matrix
   Exercise `` `Board.step` `` for all `` `MoveResult` `` values by moving **`player`**s/**`enemies`** into free tiles, walls, occupied tiles, and out-of-bounds positions.

   Ensure occupants transfer correctly and collisions are surfaced.

6. Tick processing and **`bonus`** expiration
   Run `` `Board.tick` `` across multiple ticks with temporary **`bonus`**es.

   Confirm `` `TickSummary.playerCaught` `` is true only on catch ticks and expired **`bonus`**es are removed when `` `BonusReward`.`onTickAndAlive` `` returns false.

7. **`Collectible`** extraction bookkeeping
   Call `` `Board.collectAt` `` on **`cell`**s containing **`required`**, **`optional`**, **`bonus`**, and **`punishment`** items.

   Ensure each removal updates internal tracking lists exactly once and returns empty when revisiting.

8. **`Cell`** occupancy enforcement
   Directly interact with `` `Cell.addOccupant` ``, `` `removeOccupant` ``, and `` `isEnterableFor` `` for **`player`**s **and** **`enemies`** at **start/exit**, verifying:
   - single-occupancy,
   - **`enemy`**-free start/exit, and
   - terrain walkability rules.

III. **`Collectible`**s and Scoring

9. **`Scoreboard`** accounting paths
   Simulate collecting **`required`**, **`optional`**, **`bonus`**, and **`punishment`** items plus timer start/stop.

   Confirm score deltas, required remaining counts, and elapsed time formatting.

10. **`BonusReward`** lifetime countdown
    Instantiate timed **`bonus`**es, tick until expiration, and ensure they report alive status until lifetime hits zero, then disappear.

11. `Punishment` value safeguards
    Create `punishment`s with positive, zero, and negative values. Verify constructor clamps to non-positive scores and ASCII symbol `rendering` stays consistent.

IV. `Enemy` Behavior

12. `Enemy` move cadence
    Configure `MovingEnemy` with a known move period.
    Tick repeatedly and verify movement only occurs on the correct cadence.

13. Greedy pathfinding decisions
    Place `enemy`/`player` at various offsets.

    Ensure `MovingEnemy.decide`:
    - prioritizes horizontal vs. vertical moves appropriately,
    - respects bounds, and
    - returns null when surrounded by walls.

V. `Spawner` Subsystems

14. `Bonus` quota scheduling
    Use `Spawner` configured with specific `spawnMinSec`/`spawnMaxSec` and lifetime.
    Simulate ticks to confirm `spawn` windows obey scheduling, quota decreases only on collection, and `bonus`es de`spawn` on timeout.

15. Regular `reward` placement capacity
    Attempt to `spawn` more `reward`s than available free `cell`s to trigger the exception, then run with sufficient `cell`s to ensure each `reward` is registered on the `board`.

16. `Punishment` reachability safety
    Provide `board`s where blocking certain `cell`s would isolate the exit or required `collectible`s.
    Verify `Spawner.spawnPunishments` detects and avoids invalid placements via `SpawnerHelper.canReach`.

17. `Enemy` `spawn` distancing
    Ensure `spawnEnemies` never places `enemies` within Chebyshev distance < 3 from start/exit and that `Board.registerEnemy` count matches requested `enemies`.

18. `Spawner` helper utilities
    Directly test `SpawnerHelper.freeFloorCells` filtering occupied/blocked tiles and `canReach` BFS logic with varied blocked sets.

VI. `Board` Generation and Helpers

19. Barrier-mode permutations
    Invoke `BoardGenerator.generate` for `NONE`, `PROVIDED`, `TEXT`, and `RANDOM` modes, using representative inputs.

    Confirm each mode enforces its validation rules (perimeter walls, resource parsing, random placement constraints) and produces walkable `board`s.

20. Random barrier validation helper
    Feed crafted barrier configurations into `GeneratorHelper.isBarrierConfigurationValid` to confirm it rejects isolated `cell`s, unreachable exits, or sealed floors.

21. `Generator` utility edge cases
    Exercise helper methods for perimeter wall generation, BFS counting, random start/exit placement, and malformed text parsing (ensuring descriptive exceptions).

VII. `UI` `Render`ing and `HUD`

22. GamePanel `render`-mode toggling
    Invoke `setRenderMode`/`toggleRenderMode` and ensure mode changes trigger repaints plus the correct painting pipeline (sprites vs. ASCII).

    Validate draw order covers terrain → `collectible`s → `enemies` → `player` and explosion overlay short-circuit.

23. ASCII symbol color mapping
    In symbol mode, verify each `Cell.symbol()` branch maps to the intended color (`player`, `enemies`, `reward`s, `punishment`s, walls, start, exit, explosions).

    Snapshot or mock `Graphics` to inspect the applied colors.

24. Game-over banner overlay
    Trigger `onGameOver`, verify banner visibility only when message present, and ensure the rounded rectangle and text center over the `board` regardless of size.

25. `HUD` text layout
    Inspect `paintHud` output for score (left), time (center), status (right), and remaining required count.

    Confirm values pull from `Scoreboard`/`GameState` accurately and update live.

26. Image asset loading failure path
    Force `loadImage` to reference a missing resource, expect `IllegalStateException` with informative message. Verify valid assets load successfully.

VIII. `App` Wiring/Bootstrap

27. `App` initialization pipeline
    Launch the app with each barrier mode configuration. Assert `App` builds `BoardGenerator`, `Spawner`, `Scoreboard`, `GameState`, and `GameController` once, wires dependencies, kicks off the `controller`, and that `runtime` components (`spawn` counts, timers) use the configured values.

Collectively, these functional tests cover every subsystem previously identified as requiring unit/regression coverage, ensuring gameplay, `spawn`ing, `render`ing, and wiring remain stable.