Test Plan

General Overview

This test plan outlines the comprehensive set of test cases designed to validate the functionality and reliability of various components within the project. The tests primarily focus on ensuring correct behavior for controllers, key event handling, game mechanics, and data integrity.

Objectives:

- Validate the correctness of implemented features.
- Ensure robustness in handling edge cases and erroneous inputs.
- Maintain code quality by verifying representation invariants (repOk) where applicable.

Tools and Frameworks Used

The following tools and frameworks were utilized during the testing process:

- **JUnit 5**: Used as the primary testing framework for creating and running unit tests.
- **Mockito**: Used for mocking dependencies to isolate the units being tested and simulate different behaviors.
- **Java AWT**: Simulated key and click events for testing user interactions in key handlers and controllers.
- **GSON Library**: JSON parsing for testing scenarios involving data import/export functionality.

These tools were integrated into the development workflow to provide automated and repeatable test execution, ensuring high reliability of the codebase.

You can check "**all-tests**" branch from this link for the details:
https://github.com/enginsuhanbilgic/group-kafes-302-project.git

1. BuildModeControllerTest

Purpose

- To verify the functionality of importing hall build objects from JSON.
- To ensure that invalid or null JSON data are handled correctly.

Test Cases

1. **testImportFromJson_Valid**
   - **Purpose**: Verify that a valid JSON string correctly populates the BuildModeController internal data structures.
   - **Preconditions**:
     - BuildModeController is instantiated.
     - JSON string is valid and includes objects for EARTH and AIR.
   - **Test Steps**:
     1. Call importFromJson with the valid JSON.
     2. Fetch the list of build objects for EARTH and verify it has the expected objects.
     3. Fetch the list for AIR and verify it is empty.
   - **Expected Results**:
     - The correct objects are populated in EARTH.
     - An empty list for AIR.
     - No exceptions.
2. **testImportFromJson_Null**
   - **Purpose**: Check that passing null JSON does not break the controller and preserves empty lists.
   - **Preconditions**:
     - BuildModeController is instantiated.

- **Test Steps**:
    1. Call importFromJson with null.
    2. Verify all hall lists remain empty (but not null).
- **Expected Results**:
    - No changes to the internal data; all hall lists remain empty.
    - No exceptions thrown.

3. **testImportFromJson_InvalidFormat_ThrowsException**
    - **Purpose**: Ensure that an invalid JSON string triggers the appropriate JsonSyntaxException.
    - **Preconditions**:
        - BuildModeController is instantiated.
        - Invalid JSON string (e.g., "This is not a valid JSON string").
    - **Test Steps**:
        1. Call importFromJson with invalid JSON.
        2. Expect the method to throw a com.google.gson.JsonSyntaxException.
    - **Expected Results**:
        - The test should catch a JsonSyntaxException.
        - The map of objects remains unchanged (and presumably empty).

2. BuildObjectControllerTest

Purpose

- To verify behavior for loading, manipulating, and transferring BuildObject data, including runes.

Test Cases

1. **testLoadWorldFromJson_ValidData**
   - **Purpose**: Validate that a properly structured JSON string populates the world map correctly.
   - **Preconditions**:
     - BuildObjectController is instantiated for a specific HallType (e.g., EARTH).
   - **Test Steps**:
     1. Provide a valid JSON that includes data for EARTH.
     2. Call loadWorldFromJson.
     3. Check that the map is not empty and EARTH has the expected number of objects.
     4. Confirm that repOk() is still true.
   - **Expected Results**:
     - World objects map is populated with the correct objects.
     - Invariant is maintained.
2. **testLoadWorldFromJson_EmptyData**
   - **Purpose**: Check behavior with empty or null JSON.
   - **Preconditions**:
     - BuildObjectController is instantiated.
   - **Test Steps**:
     1. Call loadWorldFromJson with an empty string.
     2. Assert that the map is empty afterward.
     3. Call loadWorldFromJson again with null.
     4. Assert that the map remains empty.
   - **Expected Results**:
     - The map is reset or remains empty.
     - No exceptions or errors.

3. **testSetRune_AndRemoveRune**
    - **Purpose**: Verify the ability to place a rune on an object and then remove it correctly.
    - **Preconditions**:
        - Manually populate a small list of BuildObjects.
        - Insert them into the controller's map for EARTH.
    - **Test Steps**:
        1. Call setRune on a specific BuildObject.
        2. Verify that the object now has hasRune = true and runeHolder references that object.
        3. Call removeRune.
        4. Verify that the object no longer has the rune and runeHolder is null.
    - **Expected Results**:
        - Rune placement and removal updates occur correctly.
        - repOk() remains true throughout.
4. **testTransferRune**
    - **Purpose**: Test that the rune can be transferred from one object to another via transferRune().
    - **Preconditions**:
        - Populate multiple BuildObjects, one having the rune.
    - **Test Steps**:
        1. Call setRune on an object.
        2. Call transferRune.
        3. Verify that exactly one object has hasRune == true.
        4. Verify that repOk() remains true.
    - **Expected Results**:
        - Exactly one of the objects now has the rune (it should have moved from the original holder).
        - Invariant is maintained.

5. **testUpdate_ClickRune_Success**
   - **Purpose**: Confirm that clicking a rune-holding object removes the rune and adds it to the player inventory.
   - **Preconditions**:
     - At least one object in the map with hasRune = true.
     - The player is positioned close enough to click on that object.
   - **Test Steps**:
     1. Perform a click coordinate inside the object's bounding box.
     2. Call update(...).
     3. Verify that hasRune is now false, and runeHolder is null.
     4. Verify that the mockPlayer now has the rune in inventory.
   - **Expected Results**:
     - Rune is successfully removed from the object and added to the player's inventory.

3. EnchantmentControllerTest

Purpose

- To confirm correct behavior for spawning, despawning, and collecting enchantments.

Test Cases

1. **testSpawnEnchantment_AfterSpawnInterval**
   - **Purpose**: Ensure a new enchantment spawns after the configured interval has passed.
   - **Preconditions**:
     - EnchantmentController is created with a mock TilesController.
     - No enchantments initially.

- **Test Steps**:
    1. Call tick(0) for initial setup.
    2. Advance time to GameConfig.ENCHANTMENT_SPAWN_INTERVAL.
    3. Mock TilesController.getTileAt(...) to return a non-collidable tile.
    4. Call tick(...) again at spawn time.
- **Expected Results**:
    - Exactly one enchantment in the list.
    - lastSpawnTime updated to the current spawn time.

2. **testDespawnEnchantment_AfterLifetime**
    - **Purpose**: Check that enchantments are removed after their lifetime expires.
    - **Preconditions**:
        - Create an enchantment with a known lifetime and spawn time.
        - Add it to enchantmentController.
    - **Test Steps**:
        1. Advance time to exceed the enchantment's lifetime.
        2. Call tick(currentTime).
        3. Check that the enchantment is removed from the list.
    - **Expected Results**:
        - The expired enchantment is no longer in the active list.
3. **testCollectEnchantment_OnClick**
    - **Purpose**: Verify that clicking on an enchantment picks it up and adds it to the player inventory.
    - **Preconditions**:
        - Player is positioned near an enchantment.
        - Enchantment is in the list.
    - **Test Steps**:
        1. Simulate a click on the enchantment's location.
        2. Call update(mockPlayer, clickPos).
        3. Verify the enchantment is removed from the controller.

    4. Verify the player’s inventory now contains the enchantment.
  - **Expected Results**:
    - The enchantment is successfully collected.
    - Enchantment is no longer in the controller’s active list.

4. GameTimerControllerTest

Purpose

- To confirm correct timer behavior: starting, pausing, resuming, stopping, and adding time.

Test Cases

1. **testStartTimer**
   - **Purpose**: Verify that starting the timer begins the countdown and reaches zero in the expected interval.
   - **Preconditions**:
     - GameTimerController is instantiated with onTick and onTimeUp callbacks.
   - **Test Steps**:
     1. Call start(5).
     2. Wait up to 6 seconds to see if latch is released at zero.
     3. Verify timeRemaining == 0.
   - **Expected Results**:
     - Timer completes exactly at zero.
     - The onTick callback triggers at each tick.

2. **testPauseAndResumeTimer**
    - **Purpose**: Confirm that pausing holds the timer's state, and resuming continues countdown from that state.
    - **Preconditions**:
        - Timer is running.
    - **Test Steps**:
        1. Start with 5 seconds.
        2. Wait 2 seconds, then pause.
        3. Record remaining time, wait another 2 seconds paused.
        4. Resume and ensure the timer continues from the recorded remaining time.
    - **Expected Results**:
        - Time does not decrease during pause.
        - Timer finishes after resuming.
3. **testStopTimer**
    - **Purpose**: Ensure stopping the timer ceases countdown but preserves the remaining time.
    - **Preconditions**:
        - Timer is running with some time left.
    - **Test Steps**:
        1. Start timer with 5 seconds.
        2. Wait 2 seconds, then stop.
        3. Check remaining time is about 3 seconds, and confirm it does not continue counting down.
    - **Expected Results**:
        - Timer halts immediately.
        - Remaining time is retained.

4. **testAddTime**
   - **Purpose**: Check that adding time while the timer is running extends the countdown.
   - **Preconditions**:
     - Timer is running.
   - **Test Steps**:
     1. Start with 3 seconds.
     2. After 1 second, add 2 seconds.
     3. Check the updated timeRemaining is 4.
     4. Wait to see if it counts down properly to zero.
   - **Expected Results**:
     - Time is incremented appropriately.
     - Timer eventually ends at zero.

5. KeyHandlerTest

Purpose

- To ensure correct handling of key events and toggles (up, down, left, right, ESC, H, B, P, R).

Test Cases

1. **testMovementKeysPressed**
   - **Purpose**: Check that pressing movement keys (UP, DOWN, LEFT, RIGHT) sets the corresponding flags to true.
   - **Preconditions**:
     - KeyHandler is instantiated.
   - **Test Steps**:

        1. Simulate keyPressed(KeyEvent.VK_UP) and check up == true.
        2. Repeat for DOWN, LEFT, RIGHT.
    - **Expected Results**:
        - Each key press sets the corresponding boolean to true.
2. **testSpecialKeysPressed**
    - **Purpose**: Verify that special keys (ESC, H, B, P, R) behave correctly, including toggles.
    - **Preconditions**:
        - KeyHandler is instantiated.
    - **Test Steps**:
        1. Press and release ESC multiple times to confirm toggle behavior.
        2. Check that pressing H sets hPressed == true.
        3. Similarly check B, P, R flags.
    - **Expected Results**:
        - ESC toggles on successive presses.
        - Others set flags to true upon press.
3. **testKeyReleased**
    - **Purpose**: Confirm that releasing any pressed key sets its flag to false.
    - **Preconditions**:
        - Relevant keys are pressed before release.
    - **Test Steps**:
        1. Press UP, then release.
        2. Assert up == false.
        3. Repeat for DOWN, LEFT, RIGHT, H, etc.
    - **Expected Results**:
        - Corresponding flags reset to false on release.
4. **testResetKeys**
    - **Purpose**: Check that resetKeys() resets all movement keys to false.
    - **Preconditions**:

        - Some movement keys are already pressed.
    - **Test Steps**:
        1. Press UP, DOWN, LEFT, RIGHT.
        2. Call resetKeys().
        3. Verify all these keys are false.
    - **Expected Results**:
        - All key flags are off (false).
5. **testMultipleKeyPresses**
    - **Purpose**: Validate that pressing multiple keys at once sets all relevant flags without interfering with others.
    - **Preconditions**:
        - KeyHandler is instantiated.
    - **Test Steps**:
        1. Press UP, RIGHT, and B simultaneously.
        2. Check that up == true, right == true, and bPressed == true; others remain false.
    - **Expected Results**:
        - Only pressed keys are active.

6. MonsterControllerTest

Purpose

- To verify monster spawning, clearing, and adjacent fighter attack logic.

Test Cases

1. **testSpawnMonsterAfterInterval**
    - **Purpose**: Ensure a monster is spawned only after the configured interval has passed.

- **Preconditions**:
  - No monsters initially.
  - inGameTime increments gradually.
  - TilesController and EnchantmentController are mocked to allow spawn location.
- **Test Steps**:
  1. Advance time to just before MONSTER_SPAWN_INTERVAL and call tick.
  2. Verify no monsters spawn.
  3. Advance time to exactly MONSTER_SPAWN_INTERVAL and call tick.
  4. Verify one monster has spawned.
- **Expected Results**:
  - Exactly one monster is in the list at the correct time.

2. **testFighterMonsterAdjacent_AttacksPlayer**
   - **Purpose**: Check that a FighterMonster adjacent to a player attacks them once the cooldown allows.
   - **Preconditions**:
     - Player at (100, 100).
     - FighterMonster placed at (playerX + TILE_SIZE, playerY).
     - lastAttackTime is sufficiently in the past.
   - **Test Steps**:
     1. Add the monster to monsterController.
     2. Set inGameTime to meet the attack cooldown requirement.
     3. Call updateAll(...).
     4. Check if the player lost 1 life.
   - **Expected Results**:
     - Player's life decreases by exactly 1.

3. **testClearMonsters**
   - **Purpose**: Verify that calling clearMonsters() empties the entire monster list.

- **Preconditions**:
  - Monster list contains a few monsters.
- **Test Steps**:
  1. Call clearMonsters().
  2. Verify the monster list is empty.
- **Expected Results**:
  - Monster list is cleared.

7. TilesControllerTest

Purpose

- To confirm correct tile grid loading, boundary/wall setup, and retrieval.

Test Cases

1. **testInitialTileGridCreation**
   - **Purpose**: Check that loadTiles() sets up the tile grid correctly and center tiles are as expected.
   - **Preconditions**:
     - TilesController is instantiated.
   - **Test Steps**:
     1. Call loadTiles().
     2. Retrieve a tile at (5, 5) (an arbitrary in-bounds location).
     3. Confirm it's non-null and not collidable.
   - **Expected Results**:

        - Tile is correctly initialized.

2. **testKafesBorderWalls**
    - **Purpose**: Verify that the border for the “kafes” area is set to collidable.
    - **Preconditions**:
        - TilesController loaded via loadTiles().
    - **Test Steps**:
        1. Check top, bottom, left, right border tiles for isCollidable == true.
    - **Expected Results**:
        - Borders are correctly set to collidable.
3. **testGetTileAtOutOfBounds**
    - **Purpose**: Confirm the method returns null when coordinates exceed the valid range.
    - **Preconditions**:
        - Tiles loaded.
    - **Test Steps**:
        1. Try getTileAt(-1, -1) and getTileAt(1000, 1000).
    - **Expected Results**:
        - Both calls return null.
4. **testSetTransparentTile**
    - **Purpose**: Ensure that setting a transparent tile marks it as collidable (e.g., a glass-like tile).
    - **Preconditions**:
        - Tiles loaded.
    - **Test Steps**:
        1. Call setTransparentTileAt(5, 5).
        2. Verify the tile at (5, 5) is collidable == true.
    - **Expected Results**:
        - The specified tile is updated as expected.

5. **testDraw**
   - **Purpose**: Validate that drawing the grid doesn't throw any exceptions.
   - **Preconditions**:
     - A valid Graphics2D context is provided.
     - Tiles loaded.
   - **Test Steps**:
     1. Call tilesController.draw(g2).
   - **Expected Results**:
     - No exceptions or errors are raised.
6. **testTileGridInitialization**
   - **Purpose**: Check if the first and last tiles in the grid are correctly initialized.
   - **Preconditions**:
     - Tiles loaded.
   - **Test Steps**:
     1. Get tile at (0, 0) and (NUM_HALL_COLS - 1, NUM_HALL_ROWS - 1).
     2. Check they are non-null.
   - **Expected Results**:
     - Both edges are valid, non-null tiles.
7. **testLoadTilesFloorTiles**
   - **Purpose**: Confirm all floor tiles are marked as not collidable.
   - **Preconditions**:
     - Tiles loaded.
   - **Test Steps**:
     1. Iterate over grid and confirm floor tiles are isCollidable == false.
   - **Expected Results**:
     - All floor tiles are non-collidable.
8. **testLoadTilesWallTiles**
   - **Purpose**: Confirm all wall tiles are marked as collidable.
   - **Preconditions**:

        - Tiles loaded.
    - **Test Steps**:
        1. Iterate over the “kafes” bounding area to check if walls are isCollidable == true.
    - **Expected Results**:
        - All walls are collidable.

8. PlayerTest

Purpose

- To validate functionality related to a Player using a LuringGemEnchantment in various scenarios.

Test Cases

1. **testUseLuringGem_ValidUsage**
    - **Purpose**: Confirm that using a gem from inventory places it in the correct spot for monsterController.
    - **Preconditions**:
        - Player has a LuringGemEnchantment.
    - **Test Steps**:
        1. Player attempts to use the gem with a valid direction (e.g., W).
        2. Verify gem is removed from inventory and placed two tiles up in monsterController.
    - **Expected Results**:
        - Player’s inventory no longer has the gem.

- monsterController.getLuringGemLocation() matches (player.x, player.y - 2*TILE_SIZE).

2. **testUseLuringGem_NoGemAvailable**
   - **Purpose**: Check no side effects occur when attempting to use a gem that isn't in inventory.
   - **Preconditions**:
     - Player inventory is empty.
   - **Test Steps**:
     1. Call useLuringGem(...).
   - **Expected Results**:
     - No gem is placed in monsterController.
     - Inventory remains empty (no errors).
3. **testUseLuringGem_InvalidDirection**
   - **Purpose**: Validate that an invalid direction does not consume the gem.
   - **Preconditions**:
     - Player has a LuringGemEnchantment.
   - **Test Steps**:
     1. Call useLuringGem('X', monsterController) where 'X' is not a recognized direction.
   - **Expected Results**:
     - The gem remains in the player's inventory.
     - monsterController has no gem location set.