Refactor 1 - Large/God class: Breaking Down MapGenerator Into Two Classes
Commit hash: 30aa1b2e3810976b15de1dac17a5cdda1f15af32

Originally, MapGenerator was responsible for both generating the game map and managing the game state, such as time progression, bonus management, and collision consequences. This violated the Single Responsibility Principle and led to a God Class code smell.
To solve this, we split responsibilities across two classes:

- MapGenerator now strictly handles initial game setup, like loading the map and positioning entities.
- MapHandler now handles gameplay updates, such as timing events, object interactions, and win/loss conditions.

This refactor improves modularity, makes future testing and changes easier, and reduces coupling between unrelated functionalities.

Refactor 2 - Feature Envy: Collapsing Player Collision Calls Into a Single Method
Commit hash: 6c983631f6f18e9c3799ac2b7428bffdde4f7552

In the original Player class, the update() method explicitly called three separate methods on CollisionChecker: checkTile(), checkObjectCollision(), and playerCollisionWithEnemy(). This created Feature Envy, since much of the collision logic resided in CollisionChecker, not Player.

To resolve this, we created a new method in CollisionChecker called checkPlayerCollisions(). This method bundles all relevant collision checks for the player into a single, internally consistent operation.

This reduces code duplication, improves encapsulation, and makes the collision logic more readable and maintainable.

Refactor 3 -High coupling: Refactoring Object Collision Handling
Commit hash: 7dd17cbc773ceb39609679954d95b82d150c526c

In the old implementation, checkObjectCollision() returned a MainObject, which the Player then passed to handleObject() for processing. This created an unnecessary dependency chain.

We improved this by letting CollisionChecker.checkObjectCollision() directly call MapHandler.handleObject() when a collision is found, eliminating the need to return the object and decouple the collision response from the player.

This improves cohesion, reduces method responsibility chaining, and simplifies the overall control flow.

Refactor 4 - Class that try to do too much: Moving handleObject() From Player to MapHandler
Commit hash: 7dd17cbc773ceb39609679954d95b82d150c526c

Previously, the Player handled game logic updates (like key collection, trap hits) through a handleObject() method. This made the Player class unnecessarily responsible for managing global game consequences.

We moved handleObject() to MapHandler, which now centralizes all logic related to game progression. This aligns better with domain responsibility and reduces feature envy, since most of the logic in handleObject() was already dependent on MapHandler or GamePanel.

This change improves class cohesion, reduces duplication, and follows object-oriented best practices.

Refactor 5 – Feature envy: changing Player coordinate calculations to its own method
Commit hash: 02292ad56194f6d7cb9be727549f9f3764d5735e

Originally, there was a feature envy in the RegularGoblin class since to calculate the player's tile coordinates, it manually used many statements to get each data in the Player needed for the calculation.

We refactored this logic by creating a method Player.getCenterTileCoordinates() inside Player to return the same tile coordinate but calculated from the Player. This encapsulation reduces the RegularGoblin dependency on the Player's data and lets the Player handle its own data for the center tile calculation.

This change helps improve encapsulation, cleans code within the RegularGoblin class and maintainability if Player coordinates are calculated differently in the future

Refactor 6 – Duplicate code in sibling classes: Extracting movement logic to parent Entity class
Commit hash: 906b126808fcf196ffffffd20d4b621d9ae80a66

Previously, both Player and RegularGoblin had almost identical codes for moving based on their direction and speed.Since the logic for the movement was the same and both were in sibling classes this was acting as duplicate code.

To fix this, we extracted the movement logic into a new method moveEntityTowardDirection() in the Entity parent class. Now, instead of each subclass hardcoding directional movement, they call this method.

This change improves maintainability, increases cohesion in the parent class by handling movement there, makes the code cleaner in the subclasses and also takes advantage of code reuse in inheritance.

Refactor 7 – Duplicate code in sibling classes: Extracting menu options to parent DefaultUI class
Commit hash: 884161c6b6066f2a5ab49eb19d5173a3bac9e4b0

Previously, each UI subclass that had menu options (EndUI, MenuUI, PauseUI, InstructionsUI) manually drew its menu options by repeating nearly identical code. Some of the repeated code included rendering text strings with borders, calculating centered X-positions, and drawing the cursor indicator. This resulted in duplicate code spread across multiple subclasses and made the UI harder to maintain or update consistently.

We refactored this by extracting the shared logic into a new drawCursorOptionsCentered() method in the parent DefaultUI class. Now all UIs that want to use a menu can call one method with their menu options in an array to display it on the screen.

This change improves modularity, consistency and removes duplication across the UI classes by utilizing the parent class.

Refactor 8 – Low cohesion/Feature Envy: Centralizing Enemy Collision Handling in CollisionChecker class
Commit hash: 0e7d4f7b9a730c9e3589f7978010b4883df9ea62

Previously, the RegularGoblin class managed its own collision logic by calling and handling multiple methods in CollisionChecker, such as checkTileCollision, checkEnemyCollision, and checkPlayer. This made RegularGoblin responsible for behaviour it didn’t own such as calling for the game to end when a collision is detected, which highlights its low cohesion while also being a feature envy.

We refactored the code by creating a new method called handleEnemyCollisions() in CollisionChecker. This method encapsulates all logic for handling enemy collisions in one place. Now RegularGoblin calls just this method instead of handling collision checks directly.

This change helps with the cohesion in the RegularGoblin class as it now better follows the single responsibility principle while also creating better readability within its class.