This refactoring exercise focused on improving the design and maintainability of several interconnected components of the game, including GameController, GameState, Scoreboard, Enemy, and MovingEnemy. The goal was to address code smells, clarify responsibilities, reduce coupling, and improve readability, while making sure no gameplay logic or public interfaces were changed. Because these classes interact with many other parts of the project, the refactoring was done carefully so that external behavior remained exactly the same.

The first major set of changes occurred inside the GameController class. As one of the largest classes in the project, it had collected several code smells related to readability, duplicated logic, and mixing unrelated responsibilities. One of the issues was high coupling in the handling of collectibles. The original version of applyCollectible mixed together score updates, collectible type checks, spawner notifications, and UI callbacks. This violated the Single Responsibility Principle and made the method difficult to understand. To improve this, the score-related behavior was extracted into its own helper method, applyScoreForCollectible. The separation did not change behavior but significantly improved readability and made it easier to modify score logic without touching UI or spawning concerns.

Another issue in GameController was the use of instanceof BonusReward to determine whether to notify the spawner. This is a classic type-checking smell. Ideally, collectible objects would provide their own polymorphic onCollected behavior. However, modifying the collectible class hierarchy would require changes throughout the codebase and risk breaking logic. For this assignment, the solution was to isolate the type-checking branch inside applyCollectible so that future changes could replace it with a polymorphic approach. This improves structure while fully preserving gameplay behavior.

The controller also contained an empty BLOCKED branch inside a switch statement that handled movement results. While technically correct, leaving an empty block provides no explanation and creates uncertainty about whether something is missing. Refactoring this into a named method, handlePlayerBlocked, documents the intent and helps future developers add feedback such as sound or UI effects without digging into movement code. The MOVED and COLLISION branches were also extracted into handlePlayerMoved and handlePlayerCollision, reducing clutter inside tryPlayerMove and making the sequence of movement actions easier to read.

Another minor improvement involved replacing cryptic variable names im and am with inputMap and actionMap. Although a small change, this improves readability, which is especially valuable in a class that coordinates many subsystems at once. Replacing repeated state.status() checks with a helper method called isRunning reduced duplication and made control flow clearer.

Beyond the controller, the Scoreboard class also required refactoring to maintain compatibility with other parts of the system. The original project relied on a two-argument constructor for Scoreboard, but the updated version only included a one-argument constructor. This caused compile errors in the App class and other files. To fix this without breaking logic, a second constructor was added that accepts both requiredRemaining and initialScore. This constructor internally forwards to the primary one and then sets the initial score. This ensures complete backward compatibility with the rest of the project and prevents the need for changes in unrelated classes.

The GameState class originally handled responsibilities related to setting win and loss states while also managing timing behavior. This caused a double invocation smell because GameController and GameState both attempted to stop timers and scoreboard behavior. To avoid altering any global architecture or game logic, the refactoring preserved behavior but clarified the flow of control. The running state checks were centralized, and the GameController now serves as the clearer owner of the stop operation, while GameState still maintains its original interface for compatibility. No logic was changed, but the code is now more organized and easier to navigate.

The Enemy and MovingEnemy classes were also adjusted to reflect clearer responsibilities between static enemies and moving ones. In the original code, movement and update logic were less clearly separated, which increased the mental load of understanding how enemies behaved each tick. The refactoring improved clarity by ensuring that each class handles only the behavior appropriate to it. Static enemies maintain their simple behavior, and MovingEnemy's movement logic is kept straightforward while avoiding duplicated or confusing code paths. Extracted helper methods and clearer naming help demonstrate intent without modifying any in-game behavior. These changes also ensure that changes to one type of enemy do not accidentally impact other types.

Taken together, these refactorings significantly improve the readability, maintainability, and clarity of the game codebase. No external behavior was altered, and no other classes needed to be updated to accommodate these changes. The refactoring clarified movement handling, collectible processing, spawning, scoring, and end-state transitions. It reduced coupling between unrelated components and provided clearer separation of responsibilities across the main subsystems. The improvements make it much easier for future developers to extend or debug the system without introducing unintended side effects.

In summary, the refactored codebase now exhibits better structure, cleaner separation of concerns, and more intuitive naming. Although the underlying architecture and gameplay remain unchanged, the organization and maintainability have been significantly improved.

These updates prepare the code for future enhancements and reduce the likelihood of bugs caused by unclear logic or duplicated responsibilities.