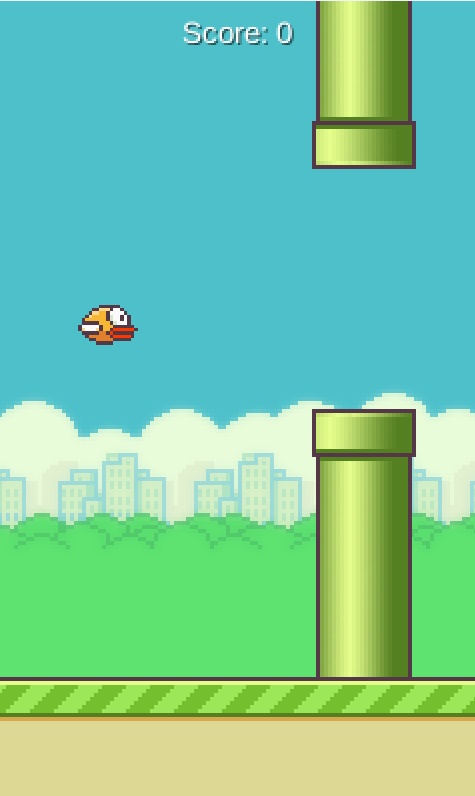
## Flappy Bird - LibGDX (Java) Android Studio Project



Submitted by:

Kezang Choden(12220037)

Chimi Kawang Selden (12220029)

Chime Gyeltshen Dorji (12220028)

### Overview

This is a Flappy Bird clone developed in Java using the LibGDX framework within Android Studio. It integrates multiple object-oriented design patterns to create a clean, extensible, and maintainable architecture. The project is structured into multiple states (menu, gameplay, credits, leaderboard) and supports power-ups, score tracking, and user input abstraction.

### Introduction

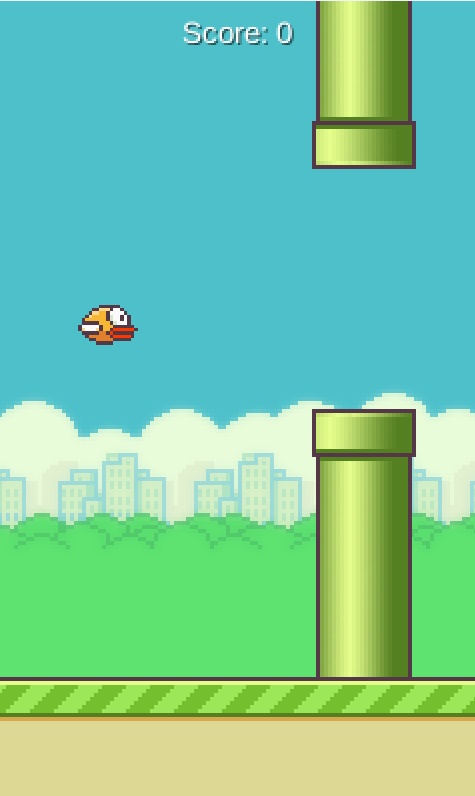
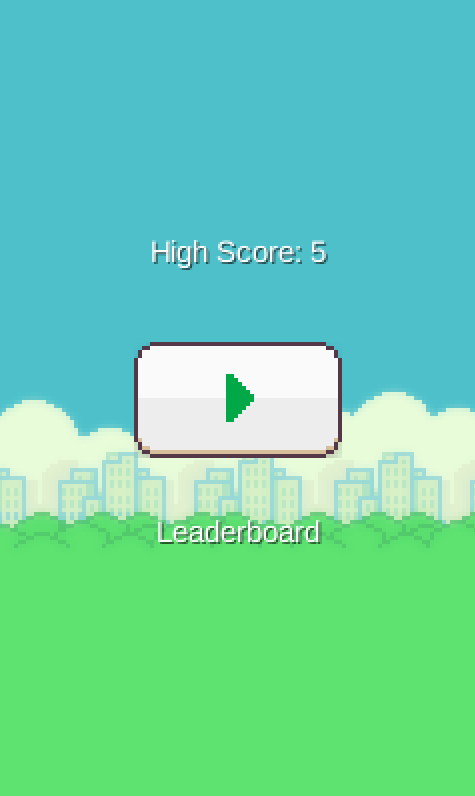
The project is about a Flappy Bird clone, a simple yet highly addictive single-player game where the primary objective is to navigate a bird through a series of pipes without hitting them. The player controls the bird by tapping the screen, causing it to flap its wings and gain altitude, while gravity constantly pulls it downwards. The game emphasizes precise timing, quick reflexes, and patience, making it both entertaining and challenging.

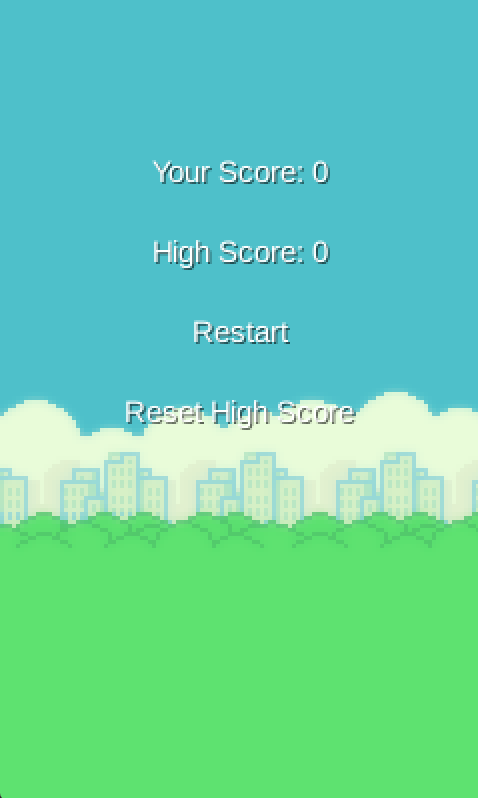
Inspired by the viral mobile game, this Flappy Bird clone incorporates modern elements to enhance engagement and playability. Features such as intuitive tap controls, a clear scoring system, and power-ups make the game suitable for a broad audience, ranging from casual players looking for a quick diversion to competitive players aiming for high scores.

The target audience for this game includes:

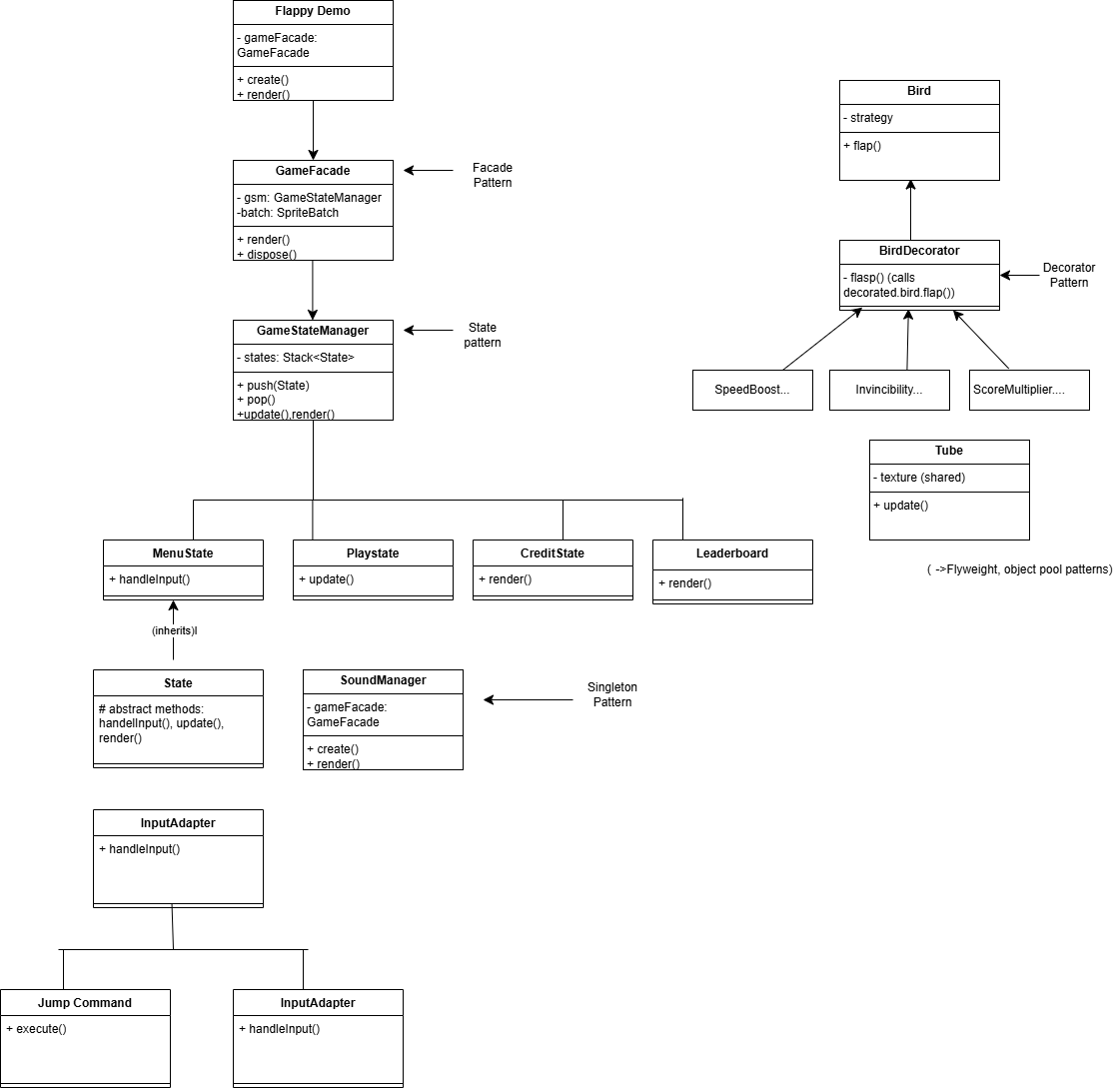
1. Casual Gamers: Players looking for a fun and quick gaming session to pass the time.
2. Competitive Players: Gamers interested in challenging themselves to achieve the highest possible score.

Game UI



Class Diagram



### **Top-Level Flow:**

* **FlappyDemo** is the entry point of the application, initializing the game via GameFacade, which encapsulates rendering and game logic — applying the **Facade Pattern**.
* **GameFacade** delegates control to **GameStateManager**, responsible for managing the stack of game states using the **State Pattern**.

### **Game States:**

* GameStateManager controls which state is currently active by pushing or popping State subclasses such as:  
  + MenuState – for the main menu
  + PlayState – the core gameplay
  + CreditState – credits screen
  + Leaderboard – high scores
* All these inherit from the abstract State class, which defines methods like handleInput(), update(), and render() — this is a **Template Method Pattern** in action.

### **Bird and Power-ups:**

* The **Bird** class contains a strategy field, allowing different movement behaviors — implementing the **Strategy Pattern**.
* The **BirdDecorator** wraps around the Bird to enhance its behavior dynamically (e.g., SpeedBoost, Invincibility, ScoreMultiplier) — this uses the **Decorator Pattern**.
* flap() in BirdDecorator calls the wrapped bird’s flap() method, layering new functionality over the base bird logic.

### **Tube (Obstacle):**

* The Tube class represents the pipes. It uses shared textures (memory efficiency) and may reuse instances instead of creating new ones — combining the **Flyweight Pattern** and **Object Pool Pattern**.

### **Input and Commands:**

* InputAdapter handles player input and delegates actions using the **Command Pattern** (e.g., JumpCommand).
* This decouples the actual input from the logic it triggers, making input handling flexible and scalable.

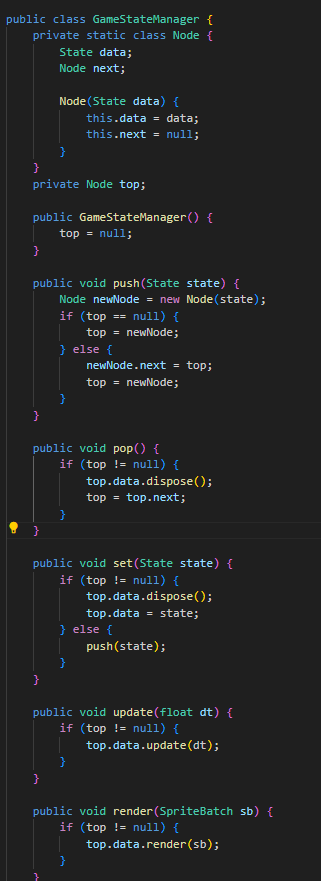
### **Sound Management:**

* SoundManager is a Singleton — only one instance controls game sound effects.
* It interacts with GameFacade to play sounds in sync with the game state.

Design Patterns Used

The project uses the following 10 design patterns, each improving specific aspects of the game's architecture:

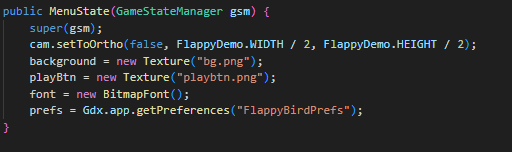
1. State Pattern
   * Used in: GameStateManager and State subclasses (MenuState,, CreditState, LeaderboardState)
   * Purpose: Manages different game screens and their transitions.



The GameStateManager class is responsible for managing the different screens (or "states") in the game, such as the main menu, gameplay, or leaderboard. It follows the **State Pattern** by maintaining a custom stack of State objects using a linked list structure, where each state represents a self-contained screen with its own logic and visuals. The push() method adds a new state on top of the stack, allowing smooth transitions to new screens. The pop() method removes the current state and properly disposes of its resources to free memory. The set() method replaces the current state with a new one, often used to reset a screen like restarting the game. The update() and render() methods delegate the game loop logic to the currently active state at the top of the stack. This class ensures clean transitions between states, prevents memory leaks through proper disposal, and keeps the game’s flow organized and modular.

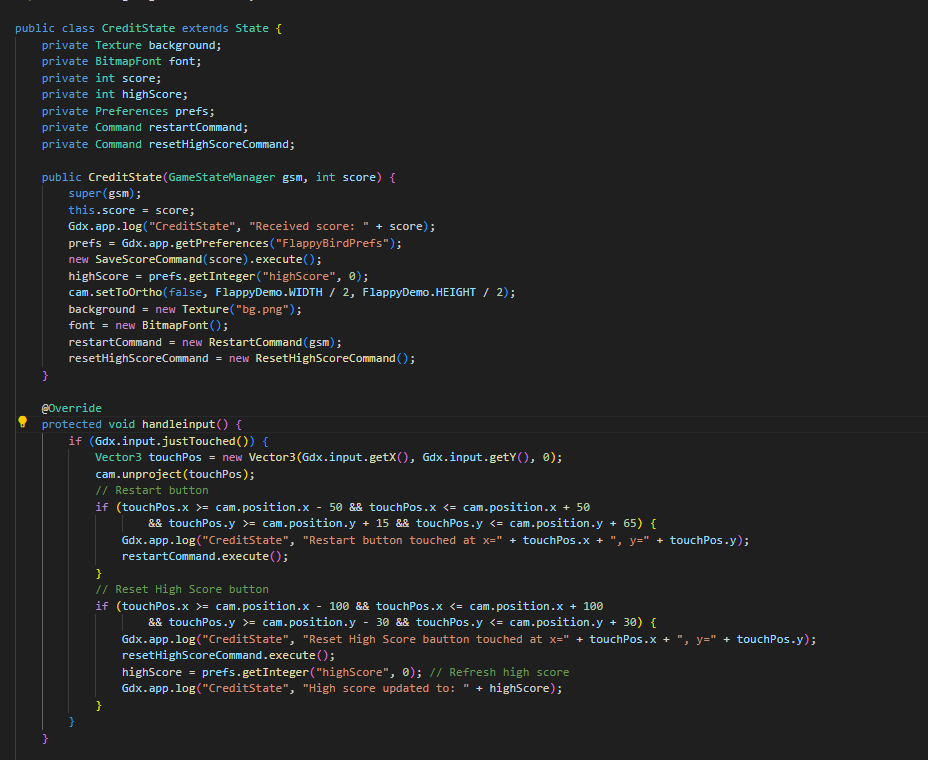
Subclass:

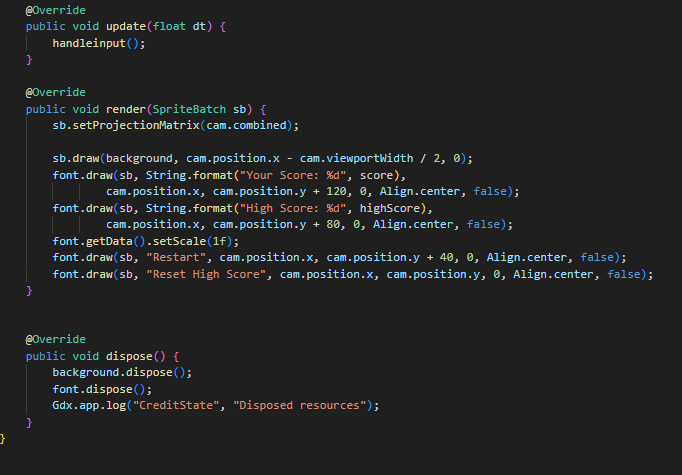
MenuState



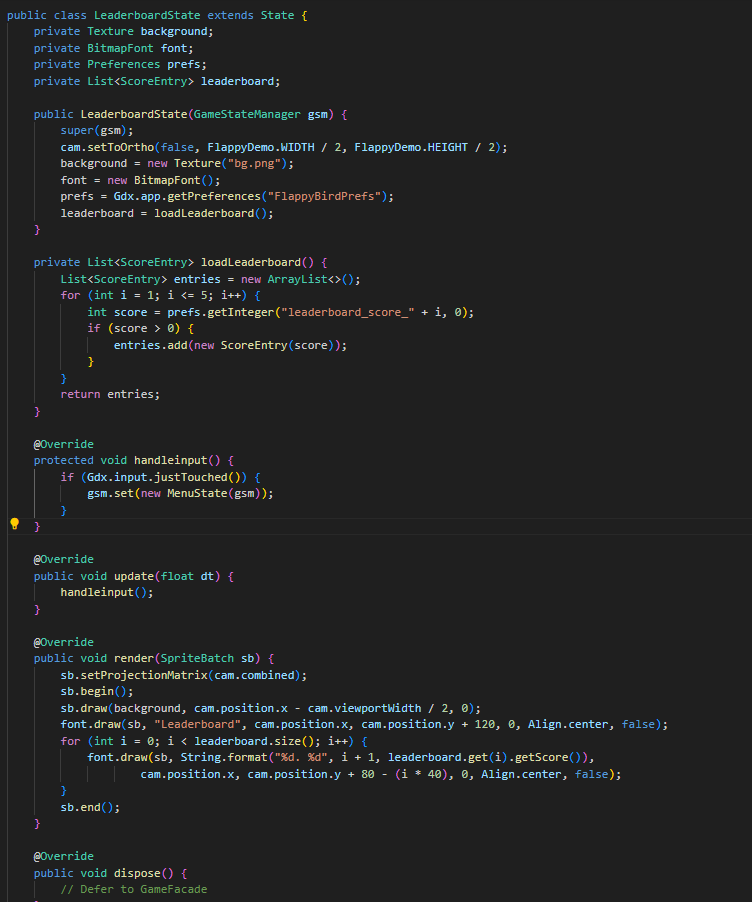
The MenuState constructor sets up the visual elements and preferences for the game's main menu. It starts by calling the superclass constructor to connect this state to the GameStateManager, which handles switching between different screens (like PlayState, MenuState, etc.). The camera is set up using setToOrtho, scaling the view to half the game’s width and height for better menu layout. It then loads the background image (bg.png) and the play button image (playbtn.png) as textures. A BitmapFont is created for drawing text, and finally, the game preferences are accessed through Gdx.app.getPreferences, which allows the menu to display or access saved data like the high score.

CreditState



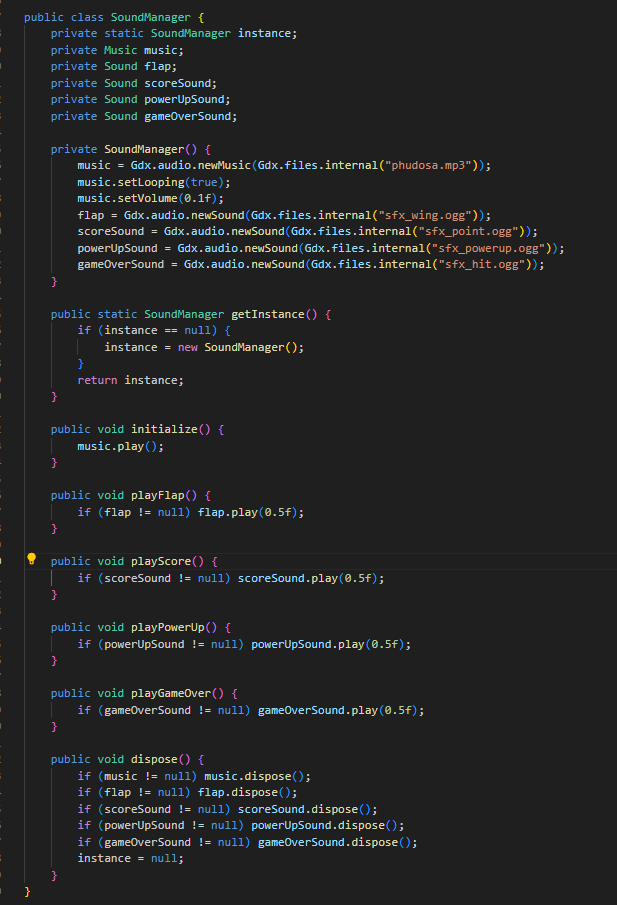


The CreditState class represents the final screen that appears after the game ends, showing the player’s current score and the all-time high score. It uses the State Pattern, extending the base State class to integrate with the GameStateManager. When this state is initialized, it saves the current score using the SaveScoreCommand and retrieves the high score from persistent preferences. It displays a background, the current score, and high score using BitmapFont, and provides two interactive buttons: one to restart the game using the RestartCommand, and another to reset the high score using the ResetHighScoreCommand. Touch input is handled manually by checking the screen coordinates and triggering the appropriate command. This class also ensures proper cleanup of resources when the state is disposed. By encapsulating UI interaction and logic for the credit screen, CreditState keeps the game modular and easy to maintain.

LeaderboardState

The LeaderboardState class displays the top scores achieved in the game, acting as a leaderboard screen. It extends the base State class and integrates with the GameStateManager to allow transitions between screens. Upon initialization, it sets up the camera, loads a background texture, and retrieves the top five scores stored in Preferences under keys like leaderboard\_score\_1 to leaderboard\_score\_5. These scores are wrapped in simple ScoreEntry objects and stored in a list. The render() method displays a title and the ranked scores using BitmapFont, centered on the screen. If the player taps the screen, the handleinput() method triggers a transition back to the main menu (MenuState). This class helps keep leaderboard logic modular and clearly separated from gameplay, and it relies on persistent data to display achievements across sessions, enhancing the game’s replay value and user engagement.

1. Singleton Pattern
   * Used in: SoundManager.java
   * Purpose: Ensures only one instance of the sound manager controls all audio.



The SoundManager class handles all audio playback in the game, including background music and sound effects. It implements the Singleton Pattern, ensuring that only one instance of the sound manager exists throughout the game. This is done through a private static instance and a public getInstance() method that lazily initializes it. Upon creation, it loads music and sound files such as the wing flap, scoring, power-up, and game-over sounds using LibGDX’s audio API. The initialize() method starts playing the background music in a loop at a low volume, while specific methods like playFlap(), playScore(), and playGameOver() are used to trigger individual sound effects during gameplay. The class also includes a dispose() method to release all audio resources when they’re no longer needed, preventing memory leaks. By centralizing all sound logic, SoundManager keeps audio handling clean, reusable, and easy to manage across various game states.

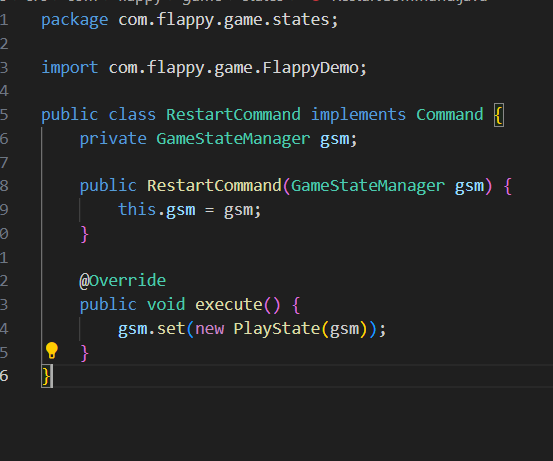
1. Command Pattern
   * Used in: JumpCommand, RestartCommand, SaveScoreCommand, ResetHighScoreCommand
   * Purpose: Decouples input handling from game logic.

JumpCommand

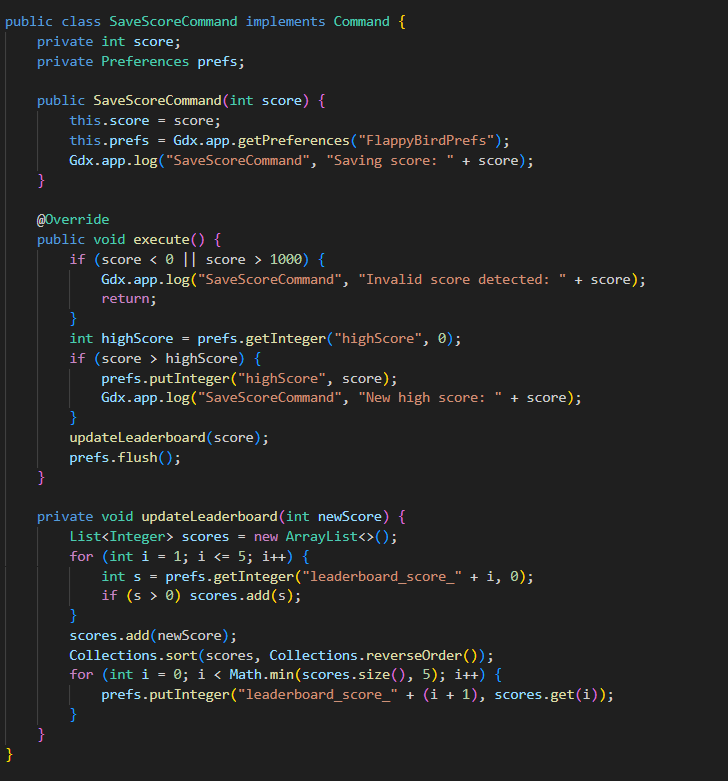


In this code, the JumpCommand class is an implementation of the Command Pattern, which is used to encapsulate a player action—in this case, making the bird jump. The class stores a reference to a Bird object and defines a single method, execute(), which calls bird.jump(). This means that the actual jumping logic is kept inside the Bird class, while the command simply triggers it. By wrapping the jump action inside a command object, the game can easily map different inputs (like a screen tap or key press) to this command without tightly coupling the input code to the bird's movement logic. This makes the game more flexible and easier to manage, especially if more commands or input types are added later. The JumpCommand would typically be used in a state like PlayState, where input is handled—when the player taps the screen, the game executes this command, causing the bird to flap upward.

RestartCommand

The RestartCommand class uses the Command Pattern to restart the game by switching to a new PlayState. It holds a reference to the GameStateManager, which manages different screens or states in the game. When the execute() method is called—usually in response to player input like pressing a restart button—it tells the GameStateManager to set the current state to a new instance of PlayState, effectively restarting the game. This design keeps the restart logic separate and easy to trigger from anywhere in the code.

SaveScoreCommand

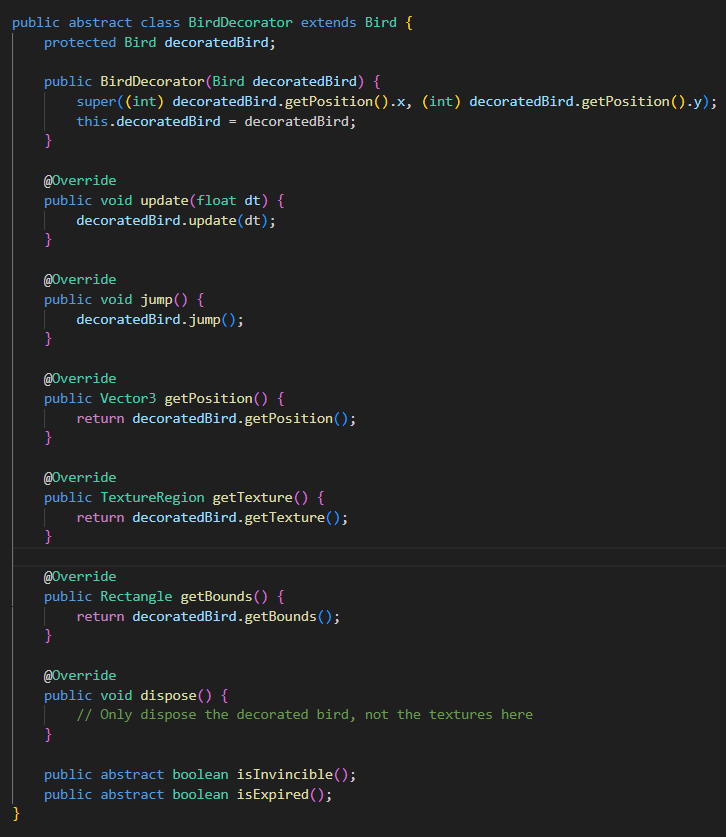
The SaveScoreCommand class uses the Command Pattern to handle saving the player's score after a game ends. It stores the score and accesses Preferences to save data. When execute() is called, it checks if the score is valid, compares it with the saved high score, and updates it if the new score is higher. It also updates the top 5 leaderboard scores by sorting and saving them. This design keeps the score-saving logic clean, reusable, and separate from the main game logic.

ResetHighScoreCommand

The ResetHighScoreCommand class is another example of the Command Pattern, designed to reset the player's high score in the game. It accesses the saved game preferences (FlappyBirdPrefs) and, when the execute() method is called, sets the high score to 0 and saves the change using flush(). It also logs the success or failure of the operation. This approach keeps the high score reset functionality encapsulated and allows it to be triggered cleanly from other parts of the game (e.g., from a settings or reset menu), without duplicating logic or tightly coupling components.

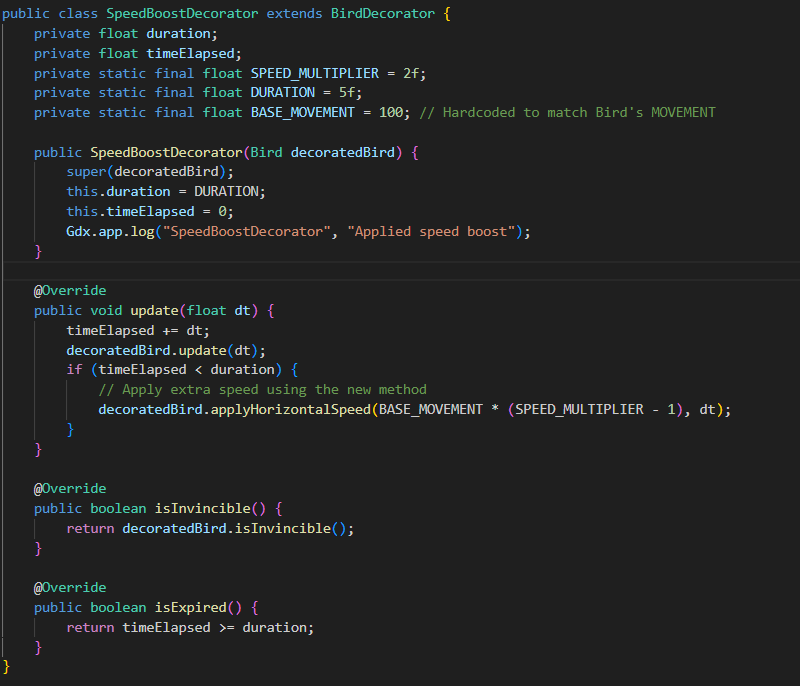
1. Decorator Pattern
   * Used in: BirdDecorator and its subclasses (SpeedBoostDecorator, InvincibilityDecorator, ScoreMultiplierDecorator)
   * Purpose: Dynamically adds power-up behaviors to the bird without modifying the base class.

BirdDecorator

The BirdDecorator is an abstract class that extends the Bird class and wraps another Bird instance (called decoratedBird). It overrides core methods like update(), jump(), getPosition(), and others by simply forwarding those calls to the wrapped bird. This setup allows you to "decorate" a bird with additional behavior—such as invincibility or power-ups—without changing the base Bird class. Subclasses of BirdDecorator can implement extra features by overriding methods or adding new ones, like isInvincible() and isExpired(), which are abstract here and meant to be defined in specific decorators. This pattern adds flexibility and keeps the design clean by separating enhancements from the core logic.

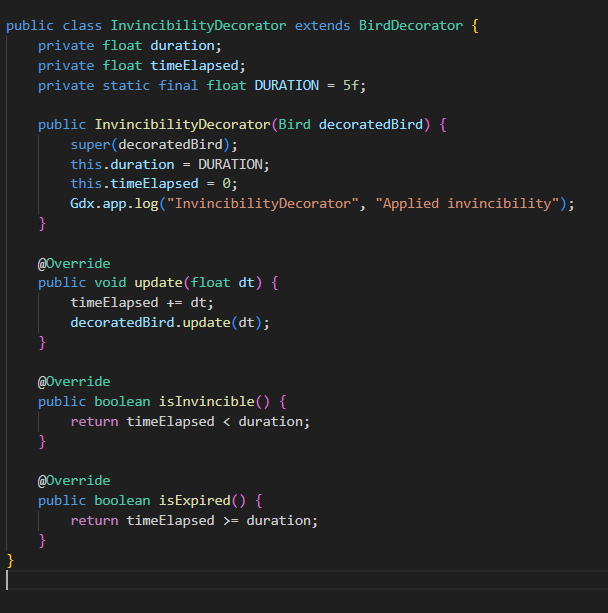
SubClass:

SpeedBoostDecorator



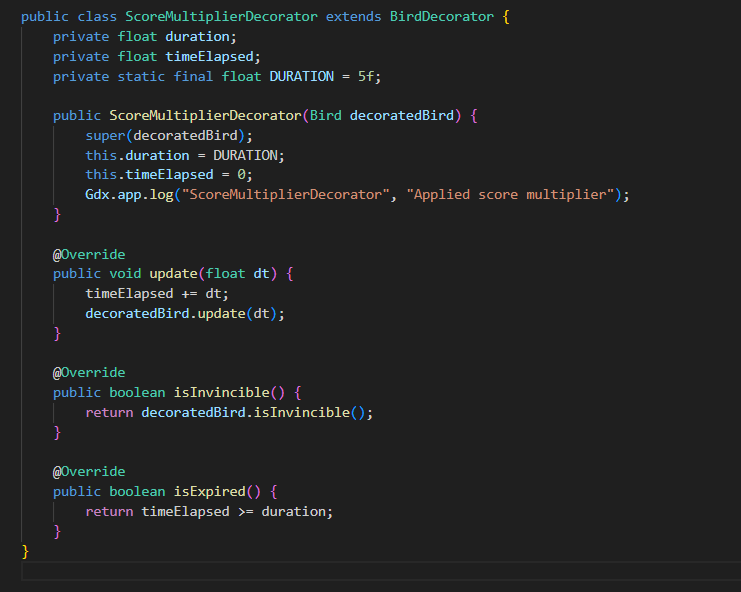
SpeedBoostDecorator extends the BirdDecorator class to add a temporary speed boost effect to the bird. It overrides the update() method to track how long the boost has been active using timeElapsed, and for a set duration (5 seconds), it increases the bird's horizontal movement speed by a multiplier (2×). It does this by calling a method like applyHorizontalSpeed() on the wrapped bird, without changing the original Bird class. The isInvincible() method passes through the base bird's value, while isExpired() returns true once the boost time is over. This design cleanly adds new behavior (speed boost) while keeping the original bird logic intact, demonstrating the flexibility of the Decorator Pattern in game feature extensions.

InvincibilityDecorator



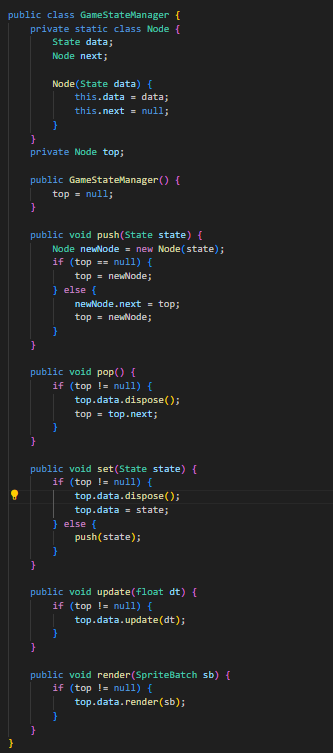
InvincibilityDecorator extends BirdDecorator to temporarily make the bird invincible. When this decorator is applied, it tracks time with timeElapsed, and for a duration of 5 seconds, it overrides the isInvincible() method to return true. This means that during this time, the bird can avoid damage or collision consequences. The update() method ensures the timer progresses while also updating the base bird’s state. Once the duration is over, isExpired() returns true, indicating that the invincibility effect has ended. This class cleanly adds new functionality to the bird without modifying the base Bird class, showcasing the power of the Decorator Pattern for adding flexible and reusable game effects.

ScoreMultiplierDecorator



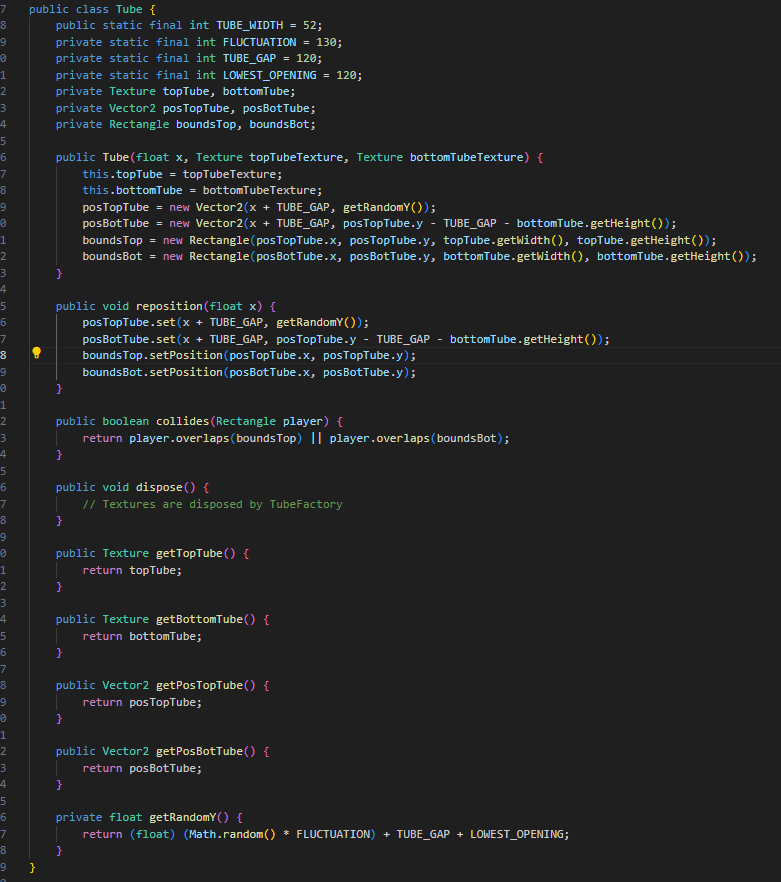
ScoreMultiplierDecorator extends BirdDecorator to apply a temporary score multiplier effect to the bird. When this decorator is active, it tracks the elapsed time and lasts for a fixed duration of 5 seconds. Although the actual score-multiplying logic is not shown in this class, it's expected to be handled elsewhere (e.g., in the score calculation logic that checks if the bird has this decorator applied). The update() method tracks time and continues to update the original bird’s state. It overrides isExpired() to signal when the effect has ended, while isInvincible() simply returns whatever the wrapped bird returns. This decorator makes it easy to temporarily boost the player's score without modifying the base Bird class, demonstrating the Decorator Pattern’s flexibility in adding game features dynamically.

1. Factory Pattern
   * Used in: GameStateFactory.java
   * Purpose: Creates game states without exposing instantiation logic.



The GameStateManager manages the different screens or "states" in the game—such as the main menu, gameplay, or leaderboard—using a custom stack structure (Node). Each Node holds a State object and a reference to the next one, allowing the game to "push" a new state onto the stack (e.g. entering gameplay), "pop" the current state off (e.g. exiting to the menu), or "set" a new state in place of the current one. The update() and render() methods delegate behavior to the active state on top of the stack. This design follows the State Pattern, helping organize screen transitions cleanly and consistently, and avoids having unrelated logic scattered throughout the code.

1. Flyweight Pattern (partial)
   * Used in: Tube.java
   * Purpose: Shares the same texture instance across multiple tubes to save memory.



The Tube class models the top and bottom pipes that the bird must fly through. It uses two textures (topTube, bottomTube) and positions them using Vector2 objects. The vertical position of the top tube is randomized using the getRandomY() method, and the bottom tube is placed below it with a fixed gap (TUBE\_GAP) in between. The reposition() method allows the tubes to be moved to a new horizontal position with a new vertical gap, which helps with recycling tubes as the bird moves forward, improving performance. Collision detection is handled by checking if the bird’s rectangle overlaps with either pipe's bounding rectangle. This class keeps the obstacle logic modular and efficient, supporting both gameplay and performance optimizations

#### **Summary Table**

|  |  |  |  |
| --- | --- | --- | --- |
| No. | Design Pattern | Applied In | Purpose |
| 1 | State | GameStateManager, State subclasses | Manage transitions between different game states |
| 2 | Singleton | SoundManager.java | Single global instance for sound management |
| 3 | Command | JumpCommand, RestartCommand, etc. | Decouple input handling from game actions |
| 4 | Decorator | BirdDecorator + Powerup wrappers | Dynamically enhance bird behavior with power-ups |
| 5 | Factory | GameStateFactory.java | Centralized, flexible creation of game states |
| 6 | Flyweight | Shared tube textures | Reduce memory use by reusing shared image data |
| 7 | Facade Pattern | GameFacade.java | Simplifies rendering and resource management, coordinating SpriteBatch and GameStateManager |

**Workflow**

The workflow of the Flappy Bird LibGDX project begins with the game launch through either the Android or desktop entry point, which initializes the GameFacade.java as the central controller for rendering and game logic. From there, the GameStateManager sets the initial game state, such as the menu or play screen, and manages transitions between various states. Each active state, like PlayState, operates through a structured game loop defined by the Template Method pattern, consisting of handleInput, update, and render methods. User input is captured by InputAdapter.java and translated into actions using the Command pattern. During gameplay, the bird’s behavior is managed using the Strategy pattern and dynamically enhanced with power-ups via the Decorator pattern. Obstacles like tubes are efficiently handled using Object Pooling to recycle objects and the Flyweight pattern to reuse shared textures. Game states are created using the Factory pattern, while sound effects are managed globally by the Singleton-based SoundManager.java. Finally, score changes and game events are reflected in the UI through a manual implementation of the Observer pattern, ensuring a responsive and modular game experience.

### **Conclusion**

The Flappy Bird LibGDX project demonstrates a well-structured and maintainable game architecture built on robust object-oriented principles and design patterns. By leveraging patterns such as State, Template Method, Command, Decorator, Strategy, Singleton, Factory, and more, the project ensures clean separation of concerns, flexibility in behavior, and ease of extending game features like power-ups, state transitions, and user interactions. This modular workflow not only enhances code readability and reusability but also optimizes performance through efficient resource management techniques like object pooling and flyweight. Overall, the project exemplifies best practices in game development with Java and LibGDX, making it both scalable and developer-friendly.