# Stick Hero Game

## Overview

Stick Hero Game is a simple yet engaging platformer game where players control a character named Stick Hero. The objective is to navigate between platforms, extending a stick to bridge gaps and collecting cherries to increase the score. The game features multiple pillars of varying widths, a reviving mechanic using collected cherries, and a scoring system encouraging players to achieve higher scores.

## Innovative Ideas

Pillar Generation Algorithm/Moving Background:

- Innovation: The use of a dynamic pillar generation algorithm introduces variety in platform widths, creating a dynamically changing environment. This innovation ensures that players face new challenges with each gameplay session, preventing predictability and promoting adaptability.

Progress Saving and Persistent Storage:

- Innovation: Allowing players to save their progress with details such as the last score, highest score, and total cherries collected adds a layer of persistence to the gaming experience. This innovative feature ensures that players can resume their journey seamlessly, fostering a sense of continuity and accomplishment.

Threading and JUnit Testing Integration

- To ensure the robustness and reliability of threaded components in the Stick Hero game through automated testing.Identifying potential race conditions, deadlocks, or other threading issues.Validating the correct behavior of threaded components under different scenarios.

## Features and Logic

Stick Extension Mechanic:

- Logic:
    - Stick extension is dynamically controlled based on user input, adjusting the stick's length.
    - The game calculates the success of landing on the next platform by comparing the stick length with the gap between platforms.
    - Timing and precision are crucial, requiring players to master the art of extending the stick at the right moment.

Multiple Pillars:

- Logic:
    - Pillar generation involves an algorithm that dynamically introduces pillars of varying widths during gameplay.
    - The algorithm ensures a challenging and dynamic arrangement of platforms, adding variety and complexity.
    - Pillar placement considers Stick Hero's movement and provides a diverse gaming experience.

Reviving Feature:

- Logic:
    - Players collect cherries to activate the reviving feature.
    - Upon revival, cherries and associated scores are deducted, balancing the strategic use of this feature.
    - The logic incorporates a check for the availability of cherries and limits the use of reviving to once per gameplay session.

Timing-based Gameplay:

- Logic:
    - Core gameplay revolves around precise timing for stick extension and platform navigation.
    - Incorrect timing leads to Stick Hero falling into the abyss, triggering the end of the game.
    - The game employs timers and event listeners to capture user input and execute actions with millisecond precision.

Reward Collection:

- Logic:
    - Flipping Stick Hero upside down is initiated by user input.
    - The logic detects the flip action and ensures successful reward collection.
    - Rewards, such as cherries, contribute to the player's score, providing an incentive for strategic flips.

Scoring System:

- Logic:
    - The scoring logic calculates points based on successful platform navigation, stick extension efficiency, and reward collection.
    - Higher scores are encouraged through milestones, such as efficient stick extension and strategic cherry collection.
    - The scoring system dynamically adjusts difficulty based on the player's performance.

Progress Saving:

- Logic:
    - Players can save their progress at any point, preserving the last score, highest score, and total cherries collected.
    - The game logic stores these values persistently, allowing players to resume their progress in subsequent sessions.
    - Saving and loading mechanisms ensure a seamless and user-friendly experience.

Graphics, Sound Effects, and Animations:

- Logic:
    - JavaFX is utilized for rendering graphics, integrating sound effects, and creating animations.
    - Graphic rendering logic ensures smooth transitions and visually appealing game elements.
    - Sound effect and animation logic enhances the immersive nature of the gaming experience, providing feedback and engagement.

## Background Thread for Audio Playback

- Purpose:
    - To manage a single instance of MediaPlayer for audio playback.
- Methods:
    - getInstance(): Returns the singleton instance of MediaPlayer, creating it if it doesn't exist.
- Thread Management:
    - The background thread is initiated when the getInstance() method is called.
    - The thread plays the game audio indefinitely in a loop using setCycleCount(MediaPlayer.INDEFINITE).

## JUnit Testing Documentation

- This JUnit test class focuses on testing the ScoreLabel update functionality, especially when updated through threading in a JavaFX environment. The runOnJavaFXThread method ensures that the update is processed on the JavaFX application thread.

## Design Patterns in the Code:

Abstract Factory Pattern:

- The code includes an abstract class Obstacle that extends Rectangle, representing game obstacles. It also has an interface Collectible for collectible items.
- There's an implementation of the abstract factory pattern with the CherryFactory class, implementing the ObstacleFactory interface to create instances of the Cherry class.

Observer Pattern:

- The observer pattern is used to update different components when the score changes.
- The ScoreObserver interface is implemented by classes ScoreLabel and HighScoreLabel to update the UI components with the current score.
- The ScoreSubject class maintains a list of observers (listeners) and notifies them when the score changes.

Singleton Pattern:

- The MediaPlayerSingleton class implements the singleton pattern to ensure only one instance of the MediaPlayer is created.
- It provides a method getInstance() to retrieve the singleton instance.

Factory Method Pattern:

- While not explicitly labeled, the CherryFactory class serves as a factory method for creating instances of Cherry objects. The factory pattern encapsulates the object creation logic.

Template Method Pattern:

- The handleCollision() method in the Obstacle class is declared as an abstract method, making it a template method. Concrete subclasses must provide their own implementation of collision handling.

Strategy Pattern:

- While not explicitly labeled, there is a potential for the strategy pattern in handling different types of obstacles. The handleCollision() method in Obstacle could be considered a strategy that varies based on the type of obstacle.

Command Pattern:

- Although not explicitly implemented, the event handling for buttons (e.g., playGameButton, highScoresButton) using lambda expressions resembles the command pattern, where actions are encapsulated as objects.

State Pattern:

- While not explicitly implemented, the game has different states such as playing, game over, and replay. State transitions are managed within the showGameScreen() and restartGame() methods.

Additional Patterns:

- MVC (Model-View-Controller):
  - The structure of the code with a clear separation of UI components (start(), showGameScreen(), showHighScoreScreen()) and game logic suggests an implicit adherence to the MVC pattern.
- Strategy Pattern:
  - The code structure allows for potential use of the strategy pattern, particularly in handling different types of obstacles with varying collision strategies.
- Command Pattern:
  - The use of lambda expressions for button events reflects a simplified form of the command pattern, where actions are encapsulated and executed on user interaction.