READ ME

Shamik Sinha 2022468
Vaibhav Singh 2022555

Group 102

OVERVIEW & PURPOSE

The primary goal of the project is to develop a game where players control a character, Stick Hero, who moves between platforms by extending a stick. The game should involve collecting rewards (cherries) while traversing the platforms and implementing a reviving feature using the collected cherries/points.

CLASSES MADE

MainPage:

MainPage class sets up the game's main window, loads necessary resources (like music), manages serialization/deserialization of the game's database, and serves as the entry point for the application.

```
public static void serialize() throws IOException {
   ObjectOutputStream out=null;
   try {
       out = new ObjectOutputStream (new FileOutputStream("database.txt"));
       out.writeObject(currentd);
   }
   finally {
       out.close();
```

```
        //System.out.println("Saved!");

        System.exit(0);

    }

}
```

MainPageController:

This class primarily focuses on controlling the interactions between the game's main menu UI (MainPage.fxml) and the gameplay UI (GamePlay.fxml), handling user input events and transitioning between scenes. Additionally, it manages the setup and control of background music through the MediaPlayer instance.

GamePlayController:

Orchestrates the core gameplay mechanics and user interactions within the Stick Hero game. It oversees the initialization of crucial game elements like the score, pillars, hero character, and manages their interactions. This controller handles user inputs, including spacebar presses for stick growth and character movement commands. Additionally, it orchestrates transitions between game screens, such as shifting elements for the next pillar, updating the score, managing cherries, and triggering end-game events. With methods to handle sound effects, animations, and game state management, this class serves as the central control hub for the gameplay experience in the Stick Hero clone.

GamePlayScreen:

Central coordinator in the Stick Hero game, orchestrating crucial game elements and interactions. Primarily responsible for spawning and managing game components, it initializes and oversees key elements like the score, hero character, and pillars within the gameplay. Through its methods, it dynamically generates new pillars and cherries during gameplay, employing animations for their appearance and movement on the screen. Furthermore, it handles score updates and calculates distances between pillars, critical for determining the hero's actions. This class essentially acts as a control center, facilitating the dynamic generation, animation, and coordination of essential game elements for a seamless gaming experience within the Stick Hero clone.

Database:

The Database class is a data storage entity in the Stick Hero game, implementing serialization via the Serializable interface. It encapsulates the last recorded score and cherry count, offering methods to access (getLastScore, getLastCherryCount) and modify (setLastScore, setLastCherryCount) these values. Additionally, it includes a function (deleteAllProgress) to reset all stored progress, initializing both the score and cherry count to zero. The class potentially follows a singleton pattern with a static reference (d) to manage a singular instance for game progress data throughout the application.

EndGameController:

The EndGameController class manages the end-game screen functionalities within the Stick Hero game. This controller facilitates interactions and UI elements like the restart, revive, and home buttons. It handles setting and displaying the final score, cherry count, and best score achieved during gameplay. Key methods such as goToHome, revivePlayer, and restartGame handle transitions between screens upon button clicks, managing the game's restart, revive, and return to the main screen functionalities. Additionally, it uses serialization to store game progress data. Overall, the class orchestrates user interactions, screen transitions, and data display associated with the end-game phase in the Stick Hero clone.

Hero:

The Hero class manages the behavior and movements of the game's main character within the Stick Hero game. It handles the creation and control of the hero's avatar represented by an ImageView, allowing it to move forward or to the death state. The moveHero method orchestrates the hero's movement towards the next pillar, incrementing the score and triggering the transition to the next level. If the hero falls, the moveHeroToDeath method executes a falling animation, adjusts the stick's angle, plays a death sound effect, and triggers the game's end. Additionally, this class interacts with other game components, such as the Stick and GamePlayController, to coordinate gameplay and handle user interactions.

Stick:

The Stick class in the game controls the behavior and visuals of the stick that the hero uses to bridge the gap between pillars. It constructs a Rectangle object, representing the stick, and manages its growth and fall. The growStick method incrementally increases the stick's length visually, while the stopGrowStick method stops the stick's growth and initiates its fall by rotating it 90 degrees. The fallStick method executes the falling animation of the stick, rotating it and triggering the hero's movement to the next pillar upon completion. Additionally, it works in coordination with the Hero class to facilitate the hero's advancement and gameplay flow.

Pillar:

The Pillar class manages the creation and properties of pillars in the game. It generates rectangular pillars used for the hero to hop between. The constructor creates a pillar with randomized width and horizontal position within specified ranges. The class contains methods to retrieve the dimensions and coordinates of the pillar. It utilizes a Rectangle object to represent the pillar visually, setting its width, fill color, and coordinates. The pillars serve as platforms for the hero to navigate during gameplay and are crucial elements for the game's mechanics.

ScoreBoard:

The ScoreBoard class manages the scoring system within the game. It contains elements like Rectangle (representing a score box), Label (for displaying the score), and a Pane where these elements reside. The class includes functionalities to update, increase the score, and notify attached observers about score changes. It uses a List of ScoreObserver to allow multiple observers to monitor and react to score updates. The updateScore method refreshes the displayed score label based on the current score value. The increaseScore method increments the score by a specified amount and updates the displayed score. Overall, the class plays a critical role in tracking and managing the game's scoring mechanism.

ScoreBoard:

The ScoreObserver interface defines a contract for objects that observe or listen to changes in the game's score. It consists of a single method, updateScore(int score), which any implementing class must define. Objects implementing this interface are notified whenever there is a change in the game's score, allowing them to react or perform specific actions based on the updated score value. This interface enables a flexible way to manage score-related events and actions across different components or modules in the game.

Cherry:

The Cherry class is responsible for generating cherries within the game environment. It initializes a cherry object as an ImageView with an image loaded from a file. The cherry's position is determined randomly between two given x-coordinates (currentPillarX and nextPillarX). Additionally, there's a probability mechanism that dictates whether the cherry spawns and if it should appear above or below the player's position.

The class contains methods to retrieve the cherry's ImageView, its x and y coordinates, and a boolean indicating whether it spawned. This allows other components to interact with the cherry object and use its position information for various game functionalities.

MainPage.fxml:

The root element, a Pane with the fx:id "EndGameRoot," sets the preferred dimensions for the interface. Within this pane, there's an ImageView displaying an image specified by the URL "@../../../Images/EndBackJPG.jpg." Three circular Circle elements serve as buttons positioned at different coordinates and filled with a semi-transparent color. These buttons have defined actions triggered by mouse clicks, linked to methods named "#restartButtonClicked," "#reviveButtonClicked," and "#homeButtonClicked." Additionally, the layout includes three Label elements ("myScore," "bestScore," "cherryScore") arranged with specified positions, preferred dimensions, and text content. These labels are likely utilized to display scores or textual information in a specific font style ("System Bold" with a size of 45.0) and white color. The interface layout and elements are designed to be controlled and managed by the associated EndGameController class.

GamePlay.fxml:

The XML snippet represents an FXML file defining a JavaFX game interface. It utilizes a Pane as the root container, housing various UI components. These components include buttons (Button) for user interaction, image views (ImageView) to display images, rectangles (Rectangle) as visual elements, and labels (Label) to present textual information. Positioned and styled with specific attributes, these elements likely constitute the gameplay screen layout. Managed by the GamePlayController class, they facilitate user interaction and information display within the game interface.

EndGame.fxml:
Outlines the layout for an end-game screen within a JavaFX application. Enclosed in a Pane container identified as "EndGameRoot," the interface consists of visual elements including an ImageView displaying a background image, three circular Circle buttons functioning as interactive elements linked to specific mouse-click actions, and three Label components configured to exhibit numerical scores or game-related data. Managed by the associated EndGameController class, these elements are positioned, styled, and organized to present a user-friendly and informative end-game display, offering functionalities for game restart, revival, and navigation to the home screen while presenting score-related information prominently.

OOPS CONCEPTS UTILIZED

Association & Composition:

Association: It's visible between classes like GamePlayScreen, ScoreBoard, Hero, etc., where instances of one class are associated with instances of another class through fields or method parameters.

Dependency: Classes have dependencies when one class uses another, like how GamePlayScreen depends on Pillar, Cherry, and Hero to control their behavior.

Ex: Full Association between Hero and Stick:

In the Hero class, there's a private field heroStick representing the stick associated with the hero.

When a Hero instance is created, it initializes its heroStick by creating an instance of the Stick class. This happens in the Hero constructor:

```
this.heroStick = new Stick(this, pane);
```

When instantiating a Stick object within the Hero constructor, the Hero instance (this) is passed as an argument to the Stick constructor. This allows the Stick instance to have a reference to the Hero it is associated with.
The Stick class holds a reference to the Hero instance via the heroID field, which allows it to interact with the associated hero's methods and properties.

For instance, the Stick class might use this reference to trigger actions on the hero, like moving the hero when the stick falls.

Encapsulation:

The fields and methods within classes are encapsulated, meaning their internal workings are hidden from other classes, allowing controlled access via public methods. For instance, in the Hero class, the methods to move the hero or check its status are encapsulated.

Inheritance:

Inheritance is demonstrated through various instances where classes extend other classes or interfaces. For example, the GamePlayScreen class has fields of type ScoreBoard, Hero, Pillar, etc., indicating an inheritance relationship or implementation of shared behavior.

Polymorphism:

Polymorphism is showcased in several ways, such as method overriding and method overloading, where methods in child classes (GamePlayController, EndGameController, etc.) override or overload methods defined in parent classes (ScoreBoard, GameElements, etc.

Composition:

Composition is observable in classes like GamePlayScreen, which contains instances of other classes (ScoreBoard, Hero, Pillar, etc.) as its fields, allowing them to work together.

Design Patterns Utilized

Singleton Design Pattern:

```
    int uniqueId = counter.getAndIncrement();
    String key = "Hero_" + uniqueId;

    if (!heroInstances.containsKey(key)) {
        Hero newHero = new Hero(him, pane, scoreboard, CherryCount);
        heroInstances.put(key, newHero);
    }
    return heroInstances.get(key);
}

private Hero(ImageView him, Pane pane, ScoreBoard scoreboard, int
CherryCount) {
    this.heroCharacter = him;
    this.pane = pane;
    this.scoreBoard = scoreboard;
    this.cherryCount = CherryCount;
    this.heroStick = new Stick(this, pane);
```

```
}
```

This implementation follows a Singleton-like pattern where the getHeroInstance method controls the creation and retrieval of Hero instances, ensuring that only one instance exists for each unique key generated, thus facilitating a form of Singleton behavior.

Observer Design Pattern:

The ScoreBoard class utilizes the Observer Design Pattern. It serves as an observable or subject. It maintains a list of ScoreObserver objects and offers methods to manage these observers. The attachObserver method allows new observers to register, while detachObserver removes them.

Whenever there's a change in the scoreValue, the ScoreBoard updates its state and notifies all registered observers by invoking their updateScore method. This method contains the updated scoreValue. This mechanism ensures that any class implementing the ScoreObserver interface can subscribe to the ScoreBoard to receive updates about score changes.

This pattern facilitates loose coupling between the ScoreBoard and its observers. It enables multiple objects to react to score changes independently. Observers can be added, removed, or modified without directly impacting the ScoreBoard class, promoting flexibility and easier maintenance of the codebase.

J-UNITs

1. *testGetPillar_height(): This test ensures that the getPillar_height() method returns the expected height of the pillar (116 units).*
2. *testGetPillar_Y_coordinate(): This test checks if the getPillar_Y_coordinate() method returns the expected Y coordinate of the pillar (1000 units).*
3. *testGetPillar_X_coordinate(): This test verifies whether the getPillar_X_coordinate() method returns a value between 300 and 650, ensuring that the X coordinate of the pillar falls within this range, indicating proper generation within specified limits.'*
4. *The test method named testImageEquality() attempts to load two images using ImageIO.read() from specific file paths or resources. However, it seems there is an*

unfinished line of code where the actual image is not loaded, leading to an incomplete test.

5. *The test class PillarTest contains test methods that verify the creation and modification of Pillar objects using different approaches: with specific values, random values, and modifying the X coordinate.*
6. *The test methods utilize assertions from JUnit's testing framework to validate the expected behavior of the Pillar class.*

Commands to be used

Home folder: 2022468_2022555

All the commands should be run on the terminal in the HOME_FOLDER unless otherwise specified.

0) Download the 2022468_2022555 code folder from Classroom and unzip.

1) mvn clean

2) mvn compile

3) mvn package

4) java -jar target\2022468_2022555-1.0-SNAPSHOT.jar