WHACK-A-MOLE

Harsith NV – 2410110426

Introduction:

This is a project for a simple whack-a-mole game that has been created using the Java Swing and AWT libraries to make the UI appearance. The project demonstrates core Java concepts including Object-Oriented Programming, Multithreading, and Event-Driven programming.

Working:

We have made it in a way that it runs using the Runnable interface. We created the GameEngine class which implements Runnable and has overridden the run method. This class takes care of the running of the game and makes sure that the moles and bombs are created and destroyed.

We use the concept of weighted distribution to choose the spawn. We use the Random package which can be used to create a random integer number and based on this we choose what spawns using a ratio of (6:3:1) for Moles, Bombs, and Bonus Moles respectively.

We utilize the concept of nesting multiple panels inside a single JFrame. We have created 3 panels to perform different functions:

1. **ScorePanel:** Takes care of showing the score, high score, and time left.
2. **GridPanel:** The most important panel containing the game running grid.
3. **ControlPanel:** Showcases the Start and Exit Buttons.

We use action listeners for each button that emits a signal to the method handleWhack. This makes sure that if there is a mole or any occupant, we change it to null and increase the score accordingly.

Technical Implementation:

- *Object-Oriented Design (OOP)*

Using the basic concepts of OOPs we used the

- *Concurrency & Thread Safety*

To ensure responsiveness, business logic is separated from the UI.

Game Loop: The GameEngine runs on a dedicated worker thread, using Thread.sleep(1000) for pacing.

UI Updates (invokeLater): Since Swing is not thread-safe, all UI updates from the background thread are maintained by the Event Dispatch Thread using SwingUtilities.invokeLater().

Synchronization: The handleWhack method is synchronized to act as a lock, preventing race conditions between the spawning thread and user click events.

- *Exception Handling*

InterruptedException: Handled to allow for graceful thread termination when the game is stopped or the window closes.

InvalidGameStateException: A custom unchecked exception used to enforce logic integrity (e.g., preventing access to invalid grid indices).

UML Class Diagram:

```
classDiagram
    class Game {
        -GameEngine engine
        -Thread thread
        -JLabel scoreLabel
        -JButton[] holes
        +updateScore(int)
        +refreshGrid(HoleOccupant[])
        +gameOver(int)
    }
    class GameEngine {
        -HoleOccupant[] gridState
        -boolean isRunning
        -int score
        +run()
        +handleWhack(int) : int
        -spawnRandom()
    }
    class HoleOccupant {
        <<abstract>>
        +whack() : int
        +getType() : String
        +getImage() : Icon
    }
    class Mole {
        +whack() : int
    }
    class Bomb {
        +whack() : int
    }
    class BonusMole {
        +whack() : int
    }
    class InvalidGameStateException {
        +InvalidGameStateException(String)
    }
    class Player {
        -String name
        -int score
    }
    Game ..> GameEngine : Creates & Starts
    GameEngine o-- HoleOccupant : Manages array of
    HoleOccupant <|-- Mole : Extends
    HoleOccupant <|-- Bomb : Extends
    HoleOccupant <|-- BonusMole : Extends
    GameEngine ..> InvalidGameStateException : Throws
    Game ..> Player : Uses
```