whack-a-mole-java

Building a Java Swing GUI: A Whack-a-Mole Case Study

1.0 Introduction to Java GUI Development with Swing

Java Swing is a mature and robust GUI widget toolkit, providing a foundational framework for creating rich, platform-independent desktop applications. While newer frameworks have emerged, Swing's comprehensive component library and event-driven model remain a powerful and instructive environment for mastering the principles of graphical user interface design. This white paper uses the practical and engaging example of a "Whack-a-Mole" game to deconstruct the core principles of GUI programming, including component hierarchy, layout management, and event-driven logic.

The objective of this document is to provide developers with a clear, step-by-step guide to building a fully interactive GUI application from the ground up using standard Java libraries. By following this case study, you will gain a tangible understanding of how to orchestrate visual elements, manage application state, and respond to user input in a coherent and structured manner.

The case study is structured to mirror the logical progression of application development. We will begin by establishing a clean project environment, then proceed to architecting the main application window. From there, we will populate the window with structured panels and interactive components, handle user input to manage game state, and finally, implement timers to drive the game's dynamic behavior.

This methodical approach provides a solid foundation, starting with the necessary preliminary steps of setting up the project environment.

2.0 Project Scaffolding and Environment Setup

A well-organized project structure is the cornerstone of a maintainable application. Before writing a single line of GUI code, establishing a clear separation of concerns through class organization and proper asset management ensures that the project remains scalable and easy to navigate. This strategic setup is the first step toward building a professional-grade application.

The initial project setup process is detailed below:

1. Project Creation: Using an IDE such as Visual Studio Code, a new "no build tools" Java project is created. For this case study, the project is named WhackAMole.
2. Class Structure: Two core Java files are established within the project's source folder. App.java serves as the application's primary entry point, containing the main function. The second file, WhackAMole.java, is designed to encapsulate all the game's logic and GUI components.
3. Asset Management: Two image assets, piranha.png and monty.png, are required for the game's visuals. These files must be downloaded and placed directly into the project's source folder to be accessible to the application at runtime.

The App.java file contains minimal code, responsible only for instantiating the main game class and thereby launching the application.

public class App { public static void main(String[] args) throws Exception { WhackAMole whackAMole = new WhackAMole(); } }

The WhackAMole.java class will utilize several standard Java libraries. The following import statements are necessary to gain access to the required classes for GUI development, event handling, and randomization:

- import java.awt.*: Imports the Abstract Window Toolkit (AWT), which provides foundational graphics classes like Image, Font, and layout managers like BorderLayout.
- import java.awt.event.*: Imports the AWT event handling framework, which includes essential interfaces like ActionListener and classes like ActionEvent.
- import java.util.Random;: Imports the Random class for generating random numbers, a key component for the game's logic.
- import javax.swing.*: Imports the Swing library, which provides platform-independent, lightweight GUI components built upon AWT, such as JFrame, JPanel, and JButton.

With the project structure and dependencies in place, we can now proceed to the creation of the application's main window, the JFrame.

3.0 Architecting the Game Window: The Role of JFrame

The javax.swing.JFrame class serves as the top-level container for any Swing application. It is the foundational element upon which the entire user interface is built, functioning as the primary window complete with a title bar, borders, and standard control buttons (minimize, maximize, close). All other GUI components, such as panels, buttons, and labels, are ultimately housed within this container.

To begin, we define class-level variables to specify the window's dimensions and to hold a reference to the JFrame instance itself.

int boardWidth = 600; int boardHeight = 650; // 600px for the game board, 50px for the text panel

JFrame frame = new JFrame("Mario Whack A Mole");

The rationale for setting the height to 650 pixels, 50 pixels greater than the width, is to allocate dedicated space at the top of the window for a score display panel, separating it cleanly from the main game area.

Within the WhackAMole class constructor, the JFrame instance is configured with a series of essential properties to define its behavior and appearance.

- setVisible(true): This is a critical step that makes the window and its contents appear on the screen. It is often called after all components have been added.
- setSize(boardWidth, boardHeight): This method sets the window's initial dimensions using the predefined width and height variables.
- setLocationRelativeTo(null): Passing null to this method centers the window on the user's screen upon launch.
- setResizable(false): This prevents the user from resizing the window, ensuring the game's layout remains fixed and predictable.
- setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE): This fundamental setting ensures that the Java Virtual Machine (JVM) terminates when the user clicks the window's close button, properly ending the application.

- setLayout(new BorderLayout()): This establishes the layout manager for the JFrame. BorderLayout divides the container into five regions: NORTH, SOUTH, EAST, WEST, and CENTER, providing a simple yet powerful way to organize child components.

With the main window architected and configured, the next logical step is to populate the JFrame with JPanels to structure the user interface.

4.0 Structuring the UI with JPanels and Layout Managers

JPanels are versatile, lightweight containers used to group and organize UI elements within a larger window. They are the primary tool for building complex, modular layouts. The arrangement of components within these panels is governed by layout managers—specialized objects that control the size and position of each element according to a specific algorithm. For this application, we will strategically use BorderLayout and GridLayout to achieve the desired structure.

The Score Display Panel

First, we will construct the panel at the top of the window to display the player's score. This involves creating a JLabel to hold the text and a JPanel to contain the label.

Class-level variables are declared for these components:

JLabel textLabel = new JLabel(); JPanel textPanel = new JPanel();

Inside the constructor, the textLabel is configured to meet the design requirements:

textLabel.setFont(new Font("Arial", Font.PLAIN, 50)); textLabel.setHorizontalAlignment(JLabel.CENTER); textLabel.setText("Score: 0"); textLabel.setOpaque(true);

Next, the textPanel is configured with its own BorderLayout manager and the textLabel is added to it. Finally, the textPanel itself is added to the main JFrame using the BorderLayout.NORTH constraint, instructing the frame's layout manager to place this panel in the top region.

textPanel.setLayout(new BorderLayout()); textPanel.add(textLabel); frame.add(textPanel, BorderLayout.NORTH);

The Game Board Panel

Second, we create the main game area using another JPanel. A boardPanel variable is declared at the class level:

JPanel boardPanel = new JPanel();

The critical decision for this panel is the choice of layout manager. A GridLayout is the ideal choice here because it enforces a rigid, uniform grid structure, ensuring all game tiles are of equal size without requiring manual position or size calculations, which simplifies layout management significantly.

boardPanel.setLayout(new GridLayout(3, 3));

The boardPanel is then added to the JFrame. When a component is added to a BorderLayout container without a specific region constraint, it is placed in the CENTER position by default, automatically occupying all available space not claimed by the other four regions.

frame.add(boardPanel);

With the UI meticulously structured into distinct panels, the next phase involves populating the game board with interactive button elements that will serve as the game tiles.

5.0 Implementing Interactive Tiles: JButtons and ImageIcons

In this game, JButtons serve as the primary mechanism for user interaction. Each of the nine tiles on the game board is an individual JButton that the player can click. To create a visually engaging experience, we will represent the game's characters—the mole and the plant—using ImageIcons, a standard Swing practice for displaying images on components.

First, an array of JButton is declared to hold references to the nine tile buttons.

JButton[] board = new JButton[9];

Inside the constructor, a for loop is used to efficiently create and add these nine buttons to the boardPanel.

for (int i = 0; i < 9; i++) { JButton tile = new JButton(); board[i] = tile; boardPanel.add(tile); }

A common challenge when working with images in Swing is ensuring they are scaled correctly to fit their container. Simply loading an ImageIcon directly from a file can result in the image appearing at its original, often oversized, dimensions. To address this, a multi-step process is employed to resize the images before they are applied to the buttons.

1. Declare ImageIcon Variables: Class-level variables are declared to hold the final, scaled icons for the mole and the plant.
2. Load as Image: An Image object is created by reading the image resource file (e.g., piranha.png for the plant or monty.png for the mole).
3. Scale the Image: The getScaledInstance() method is called on the Image object. This method returns a new, resized Image. We specify the target dimensions (150x150 pixels) and a scaling hint (Image.SCALE_SMOOTH) for higher-quality rendering.
4. Create Final ImageIcon: A new ImageIcon is created from the scaled Image object. This final icon is now properly sized for the buttons.

The following code block demonstrates this resizing process for the plant image:

// Original image -> scaled image -> final image icon Image plantImage = new ImageIcon(getClass().getResource("./piranha.png")).getImage(); plantIcon = new ImageIcon(plantImage.getScaledInstance(150, 150, java.awt.Image.SCALE_SMOOTH));

To enhance the visual aesthetic, it's desirable to remove the default focus rectangle that appears around a JButton's icon when it is clicked. This is achieved with a single line of code inside the button creation loop:

tile.setFocusable(false);

Finally, a crucial optimization for perceived performance is deferring the frame.setVisible(true) call to the end of the constructor. Invoking it earlier would render the frame and then populate it component by component, resulting in a noticeable flicker or a slow 'painting' effect on screen. By assembling the entire component hierarchy in memory first and then making the frame visible, the application presents a fully-formed, responsive UI to the user instantaneously.

The static game board is now complete. The next step is to bring it to life with dynamic movement and game logic.

6.0 Driving Dynamic Behavior with Timers and Game Logic

The dynamic nature of the Whack-a-Mole game—the constant movement of the mole and the plant—is driven by the javax.swing.Timer class and event handling. A Swing Timer fires an ActionEvent at regular, specified intervals. By listening for these events, we can create a recurring "tick" that allows for scheduled actions. This mechanism is central to the game's design, enabling us to update the game state and refresh the GUI automatically.

To manage the game's state and timing, several new class-level variables are required:

- JButton curMoleTile; and JButton curPlantTile;: These variables will keep track of which tile currently contains the mole and the plant, respectively.
- Random random = new Random();: An instance of Random is needed to select tiles randomly.
- Timer setMoleTimer; and Timer setPlantTimer;: These are the timer objects that will control the movement of the mole and the plant.

Mole Movement Timer

The setMoleTimer is initialized to fire an event every 1000 milliseconds (one second). Its constructor takes the delay and an ActionListener that defines the logic to be executed on each tick. The logic inside the actionPerformed method executes every second to clear the mole's previous position and select a new one.

With two independent timers selecting tiles randomly, a race condition is inevitable: a collision where both timers select the same tile for the same tick is guaranteed to happen. One timer's action can overwrite the other's, causing a character to 'disappear' for a cycle and creating an ambiguous game state where a single tile represents both the mole and the plant.

To resolve this, we implement a simple yet robust guard clause. After a new tile is randomly selected, we check if it is already occupied by the plant.

JButton tile = board[num];

if (curPlantTile == tile) { return; }

The if (curPlantTile == tile) { return; } check immediately exits the method if a conflict is detected. This effectively forfeits the mole's turn for that cycle, preserving the integrity of the game state and preventing one entity from visually overwriting the other. If there is no conflict, curMoleTile is updated to the new tile and the moleIcon is applied.

Plant Movement Timer

The implementation for setPlantTimer is nearly identical. It is configured with a slightly different interval of 1500 milliseconds to create more varied movement patterns. It contains the same logic for clearing the previous tile and selecting a new one, but with a reciprocal conflict check: if (curMoleTile == tile). This prevents the plant from appearing on a tile that is already occupied by the mole.

Finally, to activate the timers and start the game's automated movement, their start() methods must be called.

setMoleTimer.start(); setPlantTimer.start();

With the automated game logic in place, the final piece of the puzzle is to handle direct player input.

7.0 Closing the Loop: Player Interaction and State Management

The ActionListener interface is the bridge between user actions, such as button clicks, and the application's response logic. This final section details how player input is captured and used to update the game's state, including modifying the score and triggering the game-over condition.

An ActionListener is added to each JButton tile. This is done efficiently within the same for loop used for their initial creation, ensuring every tile is interactive.

// Inside the for loop that creates JButtons tile.addActionListener(new ActionListener() { public void actionPerformed(ActionEvent e) { // Click-handling logic goes here } });

Inside the actionPerformed method, a clear flow of control determines the outcome of a click:

1. Identify the Source: The e.getSource() method returns the specific object that triggered the event. Since this method returns a generic Object, it must be cast to a JButton to be useful. This identifies exactly which of the nine tiles was clicked.
2. Check for Mole: The clicked tile is compared to the curMoleTile. If they are the same, the player has successfully "whacked" the mole. The score variable is incremented by 10, and the textLabel is updated to display the new score.
3. Check for Plant: If the click was not on the mole's tile, an else if condition checks if the clicked tile matches curPlantTile. If it does, the player has clicked the plant, which triggers the game-over sequence.

The game-over sequence involves three distinct actions to finalize the game state:

- Update Display: The textLabel is updated to inform the player, displaying the message "Game Over:" followed by their final score.
- Stop Timers: Both the setMoleTimer and setPlantTimer are stopped using their respective .stop() methods. This immediately freezes the mole and plant in their final positions, halting all further movement.
- Disable Board: A for loop iterates through all nine buttons in the board array. The setEnabled(false) method is called on each one. This makes the buttons non-interactive and visually "grays them out," providing clear feedback to the player that the game has ended and no further input will be accepted.

This completes the core game loop, successfully connecting automated game logic with direct player interaction and state management.

8.0 Conclusion and Further Enhancements

This white paper has demonstrated the core principles of Java Swing by constructing a complete, interactive Whack-a-Mole game from scratch. Through this case study, we have explored the essential roles of JFrame as the main window, JPanel and layout managers for structuring the UI, JButton and ImageIcon for creating interactive visual elements, and the Timer/ActionListener pattern for driving dynamic behavior and responding to user input. This combination of components provides a powerful and flexible toolkit for building a wide array of desktop applications.

The foundational concepts covered here open the door to numerous potential enhancements. Developers are encouraged to build upon this project to further their skills. Potential next steps include:

- Increased Difficulty: Introduce more complex game mechanics, such as implementing multiple piranha plants that appear simultaneously.
- Replayability: Add a "Reset" button to the interface. This would allow a player to restart the game after a "Game Over" without needing to close and rerun the entire application.
- State Persistence: Implement logic to track an all-time high score. This would involve creating a variable to store the highest score achieved and displaying it on the screen, adding a long-term goal for the player.

The Swing toolkit, with its rich component set and straightforward event model, remains a highly effective and educational platform for developing sophisticated and responsive desktop applications in Java.