Table of Contents

I. Project Abstract and Objectives

Abstract

This report provides a comprehensive technical analysis of a classic "Snake" arcade game implemented in the Java programming language. The project successfully utilizes the Java Swing graphical user interface (GUI) library to render the game environment and manage user interaction. Core functionalities include player-controlled snake navigation, consumption of food items, progressive snake growth upon consumption, real-time score tracking, and collision-detection-based "game over" conditions. This document presents a detailed overview of the implemented software architecture, core logic, and underlying algorithms. It then provides a critical analysis of the existing design, highlighting architectural limitations and algorithmic inefficiencies. Finally, the report explores advanced algorithm problem-solving, proposing optimized data structures to achieve constant-time $O(1)$ operations and discussing sophisticated AI-based solving-strategies, from A* pathfinding to Hamiltonian Cycles.

Project Objectives

The primary objectives for the development of this project were as follows [1]:

- To apply and demonstrate proficiency in core Java programming concepts within a practical, interactive application.
- To gain hands-on experience with 2D graphics programming and GUI development using the Java Swing and Abstract Window Toolkit (AWT) libraries.[3]
- To implement fundamental game development mechanics, including a real-time game loop, event-driven user input handling, and dynamic game-state management.[4]
- To accurately recreate the universally recognizable gameplay of the classic Snake game, wherein the player's objective is to navigate an ever-growing snake to consume food items while avoiding collisions with both the screen boundaries and the snake's own body.

II. System Architecture and Design Analysis

2.1 Analysis of the Implemented (Monolithic) Architecture

The project, as implemented, follows a monolithic design pattern. In this architecture, the vast majority of application logic is consolidated within a single primary class, SnakeGame. This

class serves as the central hub for the entire program, managing responsibilities that, in a more complex design, would be distributed among multiple objects.

This SnakeGame class extends javax.swing.JPanel, which allows it to be placed within a JFrame and to override the paintComponent method for custom rendering. Concurrently, the class implements multiple Java interfaces to handle discrete tasks:

1. **java.awt.event.KeyListener**: Implemented to capture keyboard events directly from the user (i.e., arrow key presses) to control the snake's direction.
2. **java.awt.event.ActionListener**: Implemented to service the javax.swing.Timer, which drives the main game loop.

Furthermore, all game-state data (the conceptual "Model") is stored as member variables within this same SnakeGame class. This data includes the snakeHead position, the snakeBody (an ArrayList), the food tile's location, the current score, and the boolean game-over flag. A small inner class, Tile, is used as a simple data transfer object (DTO) to store (x, y) coordinates, but it contains no independent logic. This consolidation of roles means a single class is responsible for rendering, input, state management, and game logic.

2.2 Architectural Critique: Lack of Separation of Concerns

The implemented monolithic design, while straightforward for a project of this small scale, suffers from a critical violation of a foundational software engineering principle: **Separation of Concerns (SoC)**.[6] SoC dictates that an application should be partitioned into distinct sections, with each section addressing a separate concern or responsibility.[8]

In the current architecture, the concerns of state, presentation, and logic are tightly coupled. For example:

- A modification to the rendering logic (e.g., changing from solid color rectangles to graphical sprites) requires editing the *same class* that handles collision detection logic.
- A change to the input system (e.g., adding mouse controls) requires modifying the *same class* that stores the snakeBody ArrayList.

This tight coupling introduces significant challenges for maintenance, debugging, and extensibility.[7] Adding a seemingly simple feature, such as a main menu or a pause screen, becomes architecturally complex because the game logic and rendering are not independent. The game state cannot be "paused" without also halting the rendering, and the rendering logic is ill-equipped to draw anything other than the game itself.

2.3 Alternative Design: The Model-View-Controller (MVC) Pattern

A more robust, professional, and extensible architecture for this application would be the **Model-View-Controller (MVC)** design pattern.[9] This pattern enforces SoC by dividing the application into three distinct, decoupled components [9]:

1. **The Model:** This component represents the data and the "business logic" of the application. In Snake, the Model would be a collection of plain Java classes (e.g., GameState.java, Snake.java, Tile.java) responsible for managing the snake's coordinates, the food's location, the score, and the game-over status.[10] Crucially, the Model would also contain all the core game logic: the move() method, collision detection algorithms, and the grow() method. The Model would be entirely independent of any GUI library (i.e., it would not import javax.swing).
2. **The View:** This component is responsible *only* for the visual presentation of the Model's data.[9] It would be a JPanel class (e.g., GamePanel.java) that holds a reference to the Model. In its paintComponent method, it would query the Model for data (e.g., model.getSnakeBodyCoordinates(), model.getScore()) and render that data on the screen. The View contains no game logic; it only draws what the Model tells it.[11]
3. **The Controller:** This component acts as the intermediary between the View and the Model.[9] It receives user input (e.g., key presses) from the View and translates those inputs into commands for the Model (e.g., "call model.setSnakeDirection(Direction.UP)").[12] It also tells the View to repaint() itself whenever the Model's state changes.

This decoupled architecture, as seen in various open-source implementations [10], would allow, for example, the GamePanel (View) to be swapped out for a different rendering engine without affecting the game logic (Model) in any way.

2.4 Nuance: Swing's "Separable Model Architecture"

A deeper analysis of the chosen framework, Java Swing, reveals that it does not use a "pure" MVC pattern. Instead, Swing employs a variation known as the **Separable Model Architecture**.[14]

Swing's architects recognized that the View (visual representation) and the Controller (user input) are often very
tightly coupled. Therefore, Swing's design "collapses" the View and Controller roles into a single UI object, often called a "UI Delegate".14 However, this architecture still strictly enforces the separation of the Model as an independent entity.14
A classic example is JTable. The JTable component itself handles the rendering (View) and the user interaction (Controller), but all the data it displays is held in a completely separate TableModel object (Model). This allows developers to manipulate the table's data by

interacting with the TableModel directly, without ever touching the GUI component.[14]

Thus, the implemented project, by consolidating the Model, View, and Controller into a single SnakeGame class, fails to adhere to *either* the classic MVC pattern *or* Swing's own native "Separable Model" design philosophy, which would, at a minimum, mandate the extraction of all game state and logic into a separate GameState class.

III. Technical Implementation and Frameworks

3.1 Core Libraries: Java Swing and AWT

The project's graphical user interface and rendering are built upon two core Java libraries:

- **javax.swing**: This is the modern, platform-independent library used for the primary components.
  - JFrame: Used as the top-level window container for the entire application.[4]
  - JPanel: The fundamental component extended by the SnakeGame class to create a custom-drawable area for the game.
  - javax.swing.Timer: The crucial class used to implement the game's main loop and control its speed.
- **java.awt (Abstract Window Toolkit)**: This is the older, foundational library upon which Swing is built. It provides the core rendering and event-handling tools.
  - Graphics: An abstract class provided to the paintComponent method, acting as the "paintbrush" to draw shapes, text, and images onto the JPanel.[4]
  - Color: Used to define the colors for the game background, the snake, the food, and the "Game Over" text.[5]
  - java.awt.event.KeyListener and java.awt.event.ActionListener: The event-handling interfaces used to manage user input and the game loop timer, respectively.[5]

3.2 The Game Loop: javax.swing.Timer Implementation

The project correctly implements a real-time, event-driven game loop using a javax.swing.Timer. This approach is idiomatic for Swing applications and avoids critical threading issues.

The mechanism is as follows: a Timer object (gameLoop) is initialized with a 100-millisecond delay. Because the SnakeGame class implements ActionListener, its actionPerformed method is automatically invoked by the Timer every 100 milliseconds. This actionPerformed method functions as the main "tick" of the game, executing two primary commands in sequence:

1. move(): Calls the method responsible for updating the game state (moving the snake, checking for collisions, etc.).
2. repaint(): Calls the Swing method that requests a screen redraw, which in turn schedules a call to paintComponent to render the new game state.

The game loop is initiated with gameLoop.start() and is appropriately terminated via gameLoop.stop() when a game-over condition is met. This design is superior to a traditional while(true) { Thread.sleep(100); } loop, as the Timer operates on Swing's **Event Dispatch Thread (EDT)**, ensuring that all GUI updates and logic modifications are thread-safe.

3.3 Input Handling: KeyListener and its Architectural Limitations

User input is captured by having the SnakeGame class (which is a JPanel) implement the KeyListener interface. The keyPressed method is overridden to listen for KeyEvent objects, check their key codes (e.g., VK_UP), and update the snake's velocity variables accordingly.

While functional, this is a common but architecturally fragile design. The primary drawback of KeyListener is the **"focus issue"**.[15] A KeyListener will *only* receive keyboard events if the component it is registered to (in this case, the JPanel) currently has the system's keyboard focus.[16]

If the user clicks on the JFrame's title bar, or if another component (like a hypothetical "Restart" JButton) were added and clicked, the JPanel would *lose focus*. Once focus is lost, the KeyListener stops receiving events, and the game appears to "freeze" from the user's perspective, as their key presses are no longer registered.[18]

The more robust and professional solution in Swing is to use **Key Bindings**.[18] Key Bindings map KeyStroke objects (a specific key, like "pressed UP") to Action objects (a unit of work, like "move snake up").[18] The crucial advantage is that bindings can be set to respond at different focus levels. By using JComponent.WHEN_IN_FOCUSED_WINDOW, the "UP" key can be bound to the "move-up" action, and this binding will fire as long as the game's *window* is active, *regardless* of which specific component (the JPanel, the JFrame, or a button) currently has focus.[16] This creates a far more reliable and user-friendly control scheme.

IV. Core Game Logic and Implemented Algorithms

4.1 Snake Body Representation: ArrayList

The core data structure used to represent the snake's body is a java.util.ArrayList<Tile> named

snakeBody. A Tile is a simple inner class used as a struct to hold the int x and int y grid coordinates of a single snake segment. The snakeHead variable is simply a reference to the Tile object at index 0 of this ArrayList. When food is consumed, a new Tile is added to the list, increasing its size by one.

4.2 Game State Algorithms

Food Placement:
When the game starts, or after a food item is eaten, a new food Tile is created. Its (x, y) grid coordinates are generated using java.util.Random, ensuring the new food item appears at a random location within the game board's boundaries.5
Snake Movement:
The snake movement algorithm is executed on each tick of the Timer.20 The logic, as implemented, follows this sequence:

1. A new Tile (the "new head") is created at the grid position immediately in front of the current snakeHead, based on the current velocity (e.g., if moving "UP," new head is at snakeHead.x, snakeHead.y - 1).
2. This new head Tile is added to the *beginning* of the snakeBody ArrayList at index 0.
3. A check is performed to see if the new head's coordinates match the food's coordinates.
4. **If no food is eaten:** The snake must maintain its length. The *last* element (the tail) is removed from the snakeBody ArrayList using snakeBody.remove(snakeBody.size() - 1).
5. **If food is eaten:** The tail-removal step is skipped. This results in the ArrayList growing by one element, effectively lengthening the snake. A new food item is then placed.
6. The snakeHead reference is updated to point to the new head at snakeBody.get(0).

4.3 Collision Detection Algorithms

The game checks for two types of "game over" collisions on every tick:

Boundary Collision (Time Complexity: O(1)):
This is a computationally "cheap" check. After the new head position is calculated, its (x, y) coordinates are compared against the four boundaries of the game board (e.g., 0, boardWidth, 0, boardHeight). If the head's x-coordinate is less than 0, or its y-coordinate is greater than or equal to boardHeight, the gameOver flag is set to true. This involves a maximum of four simple integer comparisons, making it a constant-time O(1) operation.
Self-Collision (Time Complexity: O(n)):
This is a more computationally expensive check. After the head moves, the algorithm must check if the head has collided with any other part of its own body. This is implemented with a for loop that iterates through the entire snakeBody ArrayList, starting from the second segment (index 1). Inside the loop, a simple coordinate check determines if the snakeHead

Tile shares the same (x, y) position as any other Tile in the list. If a match is found, the gameOver flag is set to true.
The algorithmic cost of this check is directly proportional to the length of the snake. This is a linear-time **O(n) operation**, where **n** is the length of the snake (and thus the size of the ArrayList). As the player's score and snake length (n) increase, the time required to perform this collision check also increases linearly.

V. Advanced Algorithmic Problem-Solving and Optimization

This section addresses the project's core algorithmic challenges by analyzing the inefficiencies of the current implementation and proposing optimized, professional-grade solutions.

5.1 Data Structure Optimization: From $O(n)$ to $O(1)$ Movement

The Problem: The implemented snake movement algorithm (Section 4.2) relies on two key ArrayList operations: snakeBody.add(0, newHead) and snakeBody.remove(snakeBody.size() - 1). While remove(size - 1) is a fast, constant-time O(1) operation, add(0, newHead) is a significant performance bottleneck.[21]

Algorithmic Analysis: A java.util.ArrayList is internally backed by a simple array.[22] When an element is added at index 0, every other element in the array must be shifted one position to the right to make room. This is a **linear-time O(n) operation**.[21] As the snake (n) grows to 100, 200, or 500 segments, this single add operation requires shifting 100, 200, or 500 elements on *every single game tick*, becoming computationally expensive.

The Solution (Deque): The snake's movement pattern (add to front, remove from back) is a classic computer science use case for a **Deque (Double-Ended Queue)**.[24] The optimal Java data structure for this task is java.util.ArrayDeque (or java.util.LinkedList, which also implements the Deque interface).[24]

An ArrayDeque provides addFirst() and removeLast() methods, both of which execute in **amortized O(1) constant time**.[21] By replacing the ArrayList with an ArrayDeque, the snake movement algorithm's time complexity per tick drops from O(n) to O(1), ensuring that the game's speed remains perfectly consistent, regardless of the snake's length.

5.2 Collision Detection Optimization: From $O(n)$ to $O(1)$ Lookup

The Problem: The self-collision check (Section 4.3) iterates the entire ArrayList, resulting in an $O(n)$ time complexity. This O(n) lookup operation *compounds* with the O(n) movement operation, making the entire game tick O(n) + O(n) = O(n), which is inefficient.

The Solution (HashSet): The problem of self-collision is one of set membership: "Does this new head coordinate *exist* in the set of occupied body coordinates?" The fastest data structure in Java for set membership lookup is a **java.util.HashSet**.

The O(1) Game Tick Implementation: The most efficient solution is to use *two* data structures concurrently [24]:

1. **Deque<Tile> snakeBody**: This ArrayDeque maintains the *order* of the snake segments, so we know which Tile is the head and which is the tail.
2. **HashSet<Tile> snakePositions**: This HashSet stores the (x, y) coordinates of every segment currently in the snake's body. Its only purpose is to provide instant $O(1)$ average-time lookups. (This requires the Tile class to properly implement the hashCode() and equals() methods).

By combining these, the entire game tick logic becomes O(1) on average.[24] The new move() algorithm would be:

```
// 1. Calculate newHead (x,y) position (O(1))
// 2. Check wall collision (O(1))

// 3. Handle snake movement and growth
if (newHead.equals(food)) {
    // EAT FOOD (All O(1) operations)
    // Do NOT remove tail from Deque or HashSet
    score++;
    placeNewFood();
} else {
    // NORMAL MOVE (All O(1) operations)
    Tile tail = snakeBody.removeLast();  // O(1)
    snakePositions.remove(tail);       // O(1)
}

// 4. Check self-collision (O(1) average time)
if (snakePositions.contains(newHead)) {
    gameOver = true;
    return;
```

```
}

// 5. Add new head (All O(1) operations)
snakeBody.addFirst(newHead);      // O(1)
snakePositions.add(newHead);      // O(1)
```

This optimized design ensures that the game's core logic loop is maximally efficient and scalable.

Table 1: Data Structure Performance Comparison (Big-O Time Complexity)

The following table justifies the algorithmic recommendations by comparing the time complexity of relevant operations across different data structures.[21]

Operation	ArrayList<T>	LinkedList<T>	ArrayDeque<T>	Use Case in Snake
get(index)	**O(1)**	O(n)	N/A	(Not needed for core logic)
add(0, E) (Add Head)	**O(n)**	**O(1)**	**O(1)**	**Frequent (Every Tick)**
remove(size-1) (Rem Tail)	**O(1)**	**O(1)**	**O(1)**	**Frequent (Every Tick)**
contains(E) (Collision)	**O(n)**	O(n)	O(n)	**Frequent (Every Tick)**

5.3 Algorithmic Solutions for an AI-Controlled Snake

Exploring the creation of an AI "solver" for Snake presents a more advanced algorithmic problem.

The Naive Approach (BFS/A* to Food):
A common first attempt is to use a standard pathfinding algorithm, such as Breadth-First Search (BFS) or A-Star (A*), to find the shortest path to the food.26 The A* algorithm is a powerful, informed search algorithm that finds the most optimal path by combining the actual

cost from the start node, $g(n)$, with an estimated "heuristic" cost to the goal, $h(n)$.28 For a grid, the standard heuristic is the Manhattan distance: h(n) = |x_1 - x_2| + |y_1 - y_2|.29
This "greedy" approach, however, is fatally flawed. The AI will successfully find the shortest path to the food, but in doing so, it will frequently trap itself (e.g., by following food into a corner), leaving no viable path to escape.[26] The AI solves the wrong problem: the goal is not just to "find the food," but to "find the food *and survive*."

The "Perfect Game" Solver: Hamiltonian Cycles
A Hamiltonian Cycle is a path in a graph (the game board) that visits every vertex (every tile) exactly once and returns to the starting vertex.31 By pre-calculating a Hamiltonian Cycle for the game board, the snake can be programmed to always follow this set path.31 It will never collide with itself because the path, by definition, never intersects.32
When food appears, the AI simply continues along its pre-defined cycle until it consumes it. This strategy is *unbeatable* and guarantees a "perfect game" by filling the entire board.[33]
More advanced versions can use A* to find safe "shortcuts" to the food, but only if the AI can safely return to the main cycle.[34]

The Intelligent Pathfinding Solver: A* with Tail-Chasing
A more dynamic and intelligent AI can be built by modifying the A* heuristic.35 Instead of just pathfinding to the food, the AI performs a multi-step check 27:

1. **Path to Food:** Use A* to find the shortest (and safest) path from the snake's head to the food.
2. **"What If" Analysis:** *Before* committing, the AI simulates the move. It assumes it *has* eaten the food and grown.
3. **Path to Tail:** From this *new* head position (at the food), the AI runs a *second* pathfinding check: "Is there *still* a valid path from my new head to my *new tail*?".[26]
4. **Decision:**
    - If a path to the food exists AND a path from the food to the tail *also* exists, the path is deemed "safe." The AI takes it.
    - If a path to the food exists, but it leads to a trap (no path to the tail), the AI *ignores the food*. Instead, it switches its goal to *following its own tail* (i.e., finding the *longest* possible path).[26] This bides time, allowing the tail to move and open up a safe path.

This "tail escape" heuristic creates a robust AI that prioritizes long-term survival over short-term "greedy" gains.

Table 2: AI Solver Algorithm Comparison

Algorithm	Core Concept	Optimality (Shortest	Completenes s (Guarantees	Risk of

		Path)	Win)	Self-Trapping
Greedy	Move one step closer to food.	No	No	**Very High**
A* (to Food)	Finds shortest path to food.[28]	**Yes**	No	**High** [26]
A* (with Tail Escape)	Finds path to food *only if* a path to its tail remains.[27]	No (Prioritizes safety)	Very High	**Very Low**
Hamiltonian Cycle	Follows a pre-defined path that visits every tile.[31]	No (Path is long)	**Yes** (Perfect) [32]	**Zero**

VI. Conclusion and Future Enhancements

Conclusion

This report has successfully documented the creation of a functional Snake game using the Java Swing and AWT libraries. The project met its primary objectives, demonstrating a practical application of Java programming, GUI rendering, and event-driven logic.[3]

A technical analysis of the implementation revealed a common monolithic architecture that, while simple, lacks the Separation of Concerns necessary for robust, extensible software. The analysis also identified the use of algorithmically suboptimal data structures (ArrayList) for the core game mechanics. This choice leads to an $O(n)$ time complexity for both snake movement and self-collision detection, resulting in a game loop whose performance degrades linearly as the player's score increases.

To address these findings, this report proposed concrete, professional-grade solutions. These include refactoring the project to a **Model-View-Controller (MVC)** architecture [9], replacing the fragile KeyListener with the robust **Key Bindings** API [18], and, most critically, implementing a Deque/HashSet data structure combination to achieve a maximally efficient, constant-time **O(1) game tick**.[24] Finally, an exploration of advanced AI-solving algorithms, from heuristic-based A* [27] to unbeatable Hamiltonian Cycles [31], was conducted to demonstrate

solutions to the "perfect game" problem.

Future Enhancements

Based on the analysis, the following enhancements are recommended for future development:

1. **Architectural Refactoring:** Refactor the entire project to implement the proposed MVC pattern, strictly separating the game logic (Model) from the rendering (View) and input (Controller).[4]
2. **Algorithmic Optimization:** Implement the ArrayDeque and HashSet data structures to replace the ArrayList and achieve an O(1) game tick. An empirical performance test could then be run to measure the speedup as the snake's length grows.
3. **Feature Expansion:** Add new features such as multiple difficulty levels (by adjusting the Timer delay), a persistent high-score system that saves to a file, and player profiles.[2]
4. **AI Implementation:** Implement one of the advanced AI algorithms, such as the A* solver with a "tail escape" heuristic, to create an AI-controlled "solver" mode or an AI opponent.[2]

VII. References

- Baeldung. (2023). *Time Complexity of Java Collections*.
- Design Gurus. (2024). *Design Snake Game*.
- GeeksforGeeks. (2024). *MVC Design Pattern*.
- Hogg, T. (2022). *Solving Snake with A* Algorithm*.
- Kychin. (2021). *Hamiltonian Cycle*.
- Oracle Corporation. (2008). *A Swing Architecture Overview*.
- Reddit. (2014-2023). Assorted discussions on Snake game algorithms and design. [r/algorithms, r/compsci, r/reviewmycode]
- Stack Overflow. (2010-2023). Assorted technical discussions on Java data structures, Swing, and algorithms.