

PacRun: A Java-Based Exploration of Object-Oriented Game Architecture with Advanced State Management

Kishan Ravikumar

Registration Number : 204751

BSc Computer Science

Supervisor's name : Renato Amorim

Second Assessor's name : Kunyang

Acknowledgements

I would like to express my sincere gratitude to my project supervisor, Dr. Renato Amorim, for his guidance and valuable feedback throughout the development of this project. His expertise has been instrumental in shaping the technical direction of PacRun.

I extend my thanks to the Computer Science and Electronic Engineering Department at the University of Essex for providing the resources necessary for this project's completion.

Abstract

This project presents PacRun, a Java-based reimagining of the classic arcade game Pac-Man, developed to explore object-oriented programming principles within game architecture. Utilising

Java Swing for graphics rendering, PacRun implements both single-player and competitive multiplayer modes where players can control Pac-Man and ghosts in a "Pellet Race" format. The game features four progressive levels with distinct mechanics: basic maze navigation, cherry power-ups providing speed boosts, apples introducing control reversal challenges, and oranges enabling ghost-hunting capabilities. A bonus round with time constraints and combo scoring systems provides additional gameplay depth.

The development process focused on creating responsive controls, intelligent ghost AI with personality-specific behaviours, and a robust state management system handling complex power-up interactions. Technical achievements include an optimised collision detection framework, dynamic difficulty scaling, animated visual feedback systems, and position-swapping mechanics in multiplayer mode. This project successfully demonstrates practical applications of design patterns in game development while providing insights into balancing nostalgic gameplay elements with modern enhancements. PacRun serves as a comprehensive case study in Java-based game implementation using object-oriented principles.

Table of Contents

1. Introduction	5
1.1 Project Overview	5
1.2 Research Objective and Aims	5
1.3 Scope and Limitations	6
1.4 Background and Theoretical Foundations.....	7
1.5 Significance of the Study	8
2. Literature Review	9
2.1 Historical Context: The Evolution of Arcade Games and Pac-Man.....	9
2.2 Contemporary Game Development Frameworks and Technologies	9
2.3 Principles of Game Design and Mechanics	10
2.4 Prior Research on Game Modifications and Enhancements.....	10
2.5 Technical Considerations in Java-Based Game Development.....	11
3. Game Design	11
3.2 Core Gameplay Features and Mechanics.....	12
3.3 Maze Structure, Level Design, and Environment	13
3.4 User Interface, Visual Aesthetics, and Accessibility	14
3.5 Ethical and Inclusive Game Design Considerations.....	15
4. Technical Implementation	16
4.1 Selection of Programming Languages, Tools, and Frameworks	16
4.2 Software Architecture and Structural Design	17
4.3 Player Control Systems and Movement Mechanics	18
4.4 Artificial Intelligence: Enemy Pathfinding and Behavior.....	19
4.5 Collision Detection and Optimisation Strategies	21
4.6 Sound Design, Visual Effects, and Enhancing Immersion	22
5. Modifications and Enhancements from the Original Pac-Man.....	24
5.1 Visual and Technical Enhancements	24
5.2 Gameplay Innovations and Player Experience	25
6. Project Management and Reflection	25
6.1 Project Planning and Methodology	26
6.2 Git Version Control	26

6.3 Jira.....	27
6.4 Risk Management.....	28
6.5 Project Context	30
6.6 Overall Achievements and Learning Outcomes	31
7. Challenges, Solutions, and Technical Optimisations.....	32
7.1 Technical Challenges, Debugging, and Performance Optimisation.....	32
7.3 Future Recommendations	33
8. Testing and Evaluation	33
8.1 Testing, User Evaluation, and Gameplay Refinement	33
8.3 Performance Evaluation and Ethical Considerations	35
8.4 Testing Results	35
9. Conclusion and Future Directions	36
9.1 Summary of Research Contributions and Findings	36
9.2 Future Directions and Broader Implications.....	37
9.3 Comparison of Intended and Implemented Features	38
9.4 Final Reflections and Critical Assessment	39
10. References and Acknowledgments	40

1. Introduction

1.1 Project Overview

This project presents *PacRun*, a Java-based reimagination of the classic 2D *Pac-Man* game. It offers a modern gameplay experience while serving as a practical exploration of object-oriented programming (OOP) and real-time interactive system design. The project bridges theoretical computer science principles, such as encapsulation, inheritance, and polymorphism, with hands-on software engineering, highlighting how modular architecture can support responsive gameplay mechanics.

At the core of the implementation is a unified *Block* class that abstracts all game entities, from walls and pellets to player characters and AI ghosts. This design enables composable behaviour without overreliance on inheritance hierarchies, reflecting best practices in software maintainability and scalability. The game leverages the Java Swing framework for graphical rendering and event-driven input handling, avoiding external engines while demonstrating the capabilities of standard libraries.

Gameplay enhancements include power-ups that introduce reversible controls, temporary speed boosts, and ghost-hunting abilities, layered across multiple levels and a bonus mode with combo scoring. A unique multiplayer “Pellet Race” mode allows one player to control a ghost, integrating position-swapping mechanics and camping detection to ensure fair competition.

PacRun demonstrates how theoretical OOP concepts, when combined with clear architectural planning and interactive design principles, can produce extensible, polished gameplay systems within a pure Java context.

1.2 Research Objective and Aims

The overarching aim of this capstone project is to conceptualise, design, and implement *PacRun*, an enhanced iteration of *Pac-Man* that utilizes modern software development paradigms while preserving the foundational gameplay elements. This study contributes to game development advancement by integrating Java-based programming, optimised mechanics, and AI-driven behaviour to expand the classical arcade experience.

The project bridges nostalgic gameplay with contemporary computational efficiency and player engagement through structured programming principles, object-oriented design, and AI-based movement strategies, creating an immersive experience while maintaining *Pac-Man*'s core simplicity.

The key research objectives follow SMART principles (Specific, Measurable, Achievable, Relevant, Time-bound):

First, develop a structured and scalable 2D game environment using Java and object-oriented programming methodologies by April 2025. Success will be measured by implementing an architecture that clearly separates game logic, rendering systems, and input handling, with a specific evaluation of the Block class design's effectiveness in managing entity behaviour.

Second, to enhance gameplay mechanics by implementing three distinct power-ups (speed-boosting cherries, control-reversing apples, and ghost-hunting oranges) by March 2025. The measurable outcome will be successful state transitions during power-up activations and player-reported engagement with these risk-reward mechanics.

Third, to optimise collision detection and movement algorithms by February 2025, achieving responsive controls with grid-aligned precision. Performance will be measured by maintaining a consistent 30fps minimum across testing environments with no collision detection anomalies.

Fourth, to assess user experience through structured playtesting with at least eight participants by March 2025, gathering quantifiable feedback on difficulty progression and engagement. Success criteria include implementing at least 80% of the critical usability improvements identified during testing.

Fifth, to apply software engineering methodologies including MVC architecture and modular development strategies, demonstrating measurable improvements in code maintainability and system cohesion by the project completion date.

Finally, to conduct a comparative analysis between PacRun and the original Pac-Man by April 2025, documenting specific mechanical, AI, and user experience innovations while acknowledging the original design's strengths.

Through these objectives, the project demonstrates how classic game formats can be reimaged through contemporary software engineering principles, contributing to both educational understanding and practical development knowledge.

1.3 Scope and Limitations

This capstone project encompasses the development of PacRun, a Java-based reimagination of the classic Pac-Man arcade game that maintains nostalgic elements while introducing modern enhancements through object-oriented programming principles.

The core scope includes implementing a complete gameplay system with four progressive levels and a bonus round, each featuring distinct mechanics: basic maze navigation (Level 1), cherry power-ups providing speed boosts (Level 2), apple power-ups creating control reversals (Level 3), and orange power-ups enabling ghost hunting (Level 4). The bonus round features time constraints and a combo-based scoring system rewarding skilled play with multipliers.

A significant extension is the competitive multiplayer "Pellet Race" mode where one player controls Pac-Man while another controls a ghost, featuring balanced mechanics including position-swapping, camping detection, and separate scoring systems that transform the traditional single-player experience.

The project includes an enhanced user interface with animated menus and comprehensive visual feedback implemented through Java Swing, alongside distinct ghost AI personalities with procedural decision-making that adapts during different game states.

Technically, the implementation creates a modular, object-oriented architecture separating game logic, rendering, and application management, demonstrating practical application of software engineering principles.

Despite these features, limitations include static maze layouts rather than procedural generation, similar maze structures across levels with variations primarily in power-up placement, local-only multiplayer functionality, minimal audio implementation, and restriction to desktop environments running Java.

1.4 Background and Theoretical Foundations

The development of PacRun is underpinned by several theoretical frameworks and historical contexts that have shaped its conceptual and technical execution.

Pac-Man, created by Toru Iwatani and released by Namco in 1980, revolutionised arcade gaming through accessible gameplay, distinctive characters, and non-violent mechanics. Its innovative approach to navigational challenge and pursuit-evasion dynamics established a template influencing countless subsequent games [1]. Pac-Man's cultural impact extended beyond gaming, creating one of the first recognisable video game icons and demonstrating the medium's potential for broad appeal.

One of Pac-Man's key technical innovations was its ghost AI system, assigning unique personality traits and movement patterns to each ghost. This differentiation created varied challenges while maintaining algorithmic efficiency within limited hardware parameters. PacRun's implementation draws direct inspiration from this approach, recreating and extending these personality-driven behaviours with enhanced procedural decision-making.

PacRun's theoretical foundation draws from Salen and Zimmerman's concept of "meaningful play" where player actions produce outcomes both discernible and integrated into the larger game context [2]. The core loop ensures player movement decisions provide immediate feedback while contributing to broader progression goals.

The project incorporates Csikszentmihalyi's "flow" theory the optimal experience state characterised by immersion and engagement [3]. Flow occurs when challenge level matches player skill, balancing between boredom and anxiety. PacRun's progressive difficulty scaling and balanced power-ups are calibrated to maintain this state.

From a software engineering perspective, PacRun implements object-oriented design principles, particularly SOLID principles, guiding the creation of modular, maintainable code. The Model-View-Controller architectural pattern, described by Gamma et al. [4], informed the separation of game logic, visual representation, and user input processing.

The game loop implementation follows Nystrom's update method pattern [5], providing a structured approach to the continuous cycle of processing input, updating state, and rendering output. This ensures consistent physics timing while allowing variable frame rates for smooth gameplay across different hardware.

For user interface design, Nielsen's heuristics guided the creation of menu systems and HUD elements [6], emphasizing system status visibility, user control, and aesthetic minimalism.

1.5 Significance of the Study

This capstone project bridges theoretical foundations and practical application in game development. By reimagining a classic arcade title through modern software engineering, *PacRun* offers relevance to educators, developers, and researchers alike.

As a case study in object-oriented design using Java, the project demonstrates how principles like encapsulation, inheritance, and polymorphism can be applied in real-time interactive systems. The complete development cycle from design to testing serves as an educational model for structured, modular software architecture.

While preserving the essence of *Pac-Man*, *PacRun* introduces enhancements such as modular power-ups and updated visuals that meet contemporary gameplay standards. These additions also support the digital preservation of classic games through reinterpretation.

Though Java is not standard in commercial game development, its pedagogical value, particularly for teaching cross-platform, object-oriented concepts, is reaffirmed here. The project contributes to discussions around language choice in academic game curricula.

Ultimately, *PacRun* advances both game preservation and computer science education. It stands as a hands-on, documented example of how foundational techniques can be used to revitalise legacy game design for modern audiences.

2. Literature Review

2.1 Historical Context: The Evolution of Arcade Games and Pac-Man

Arcade gaming emerged as a cultural and technological milestone in the late 1970s and early 1980s, establishing game design patterns that continue to influence digital entertainment. Released by Namco in 1980, *Pac-Man* marked a significant departure from violent or abstract gameplay trends, introducing maze-based navigation centred on pellet collection and ghost evasion. Designed by Tōru Iwatani to appeal to a broader demographic, including female players, it became a global cultural icon and one of the first character-driven games [1].

Among its technical contributions was the introduction of AI-based ghost behaviour, with each ghost assigned a unique movement pattern and pursuit style. These differentiated behaviours, while implemented under strict hardware constraints, introduced variety and tension, contributing to a dynamic and rewarding experience. Similarly, the maze layout's integration of chokepoints, alternate paths, and risk-reward placements like power pellets laid the groundwork for spatial design still seen in contemporary games.

This historical foundation justifies *PacRun*'s design direction, which seeks to preserve the accessibility and core loop of *Pac-Man* while exploring modern gameplay extensions and software engineering principles.

.

2.2 Contemporary Game Development Frameworks and Technologies

The landscape of game development has evolved significantly since *Pac-Man*'s release, with developers now having access to a wide range of programming paradigms and tools. This section focuses specifically on Java-based game development and its relevance to the *PacRun* project.

Java, while not the industry standard for commercial-grade gaming, offers a combination of platform independence, object-oriented architecture, and mature standard libraries that make it well-suited for educational and exploratory projects. Its automatic memory management simplifies development by reducing common issues related to crashes and memory leaks, though care must be taken to minimise performance costs during garbage collection.

For *PacRun*, Java's built-in Swing framework was selected due to its accessibility and native integration with the Java language. Although not originally designed for high-performance graphics, Swing supports event-driven interaction and custom rendering, which provides sufficient functionality for developing a real-time, grid-based game without the need for external engines.

Architecturally, *PacRun* follows game programming patterns that promote clean, maintainable code. The game loop pattern manages frame updates and rendering cycles, while a component-based approach supports modular entity behaviours. These design principles allowed for effective separation of concerns and scalable gameplay logic.

Performance was addressed by minimising object creation during runtime and considering spatial optimisation strategies for collision detection. Swing’s event-driven model also facilitated efficient input handling and dynamic UI feedback.

From an educational perspective, the project demonstrates how theoretical principles in computer science and software engineering can be applied in practice. The use of Java and Swing allowed for transparent implementation of architecture, control logic, and game state transitions, reinforcing object-oriented design in an applied context.

2.3 Principles of Game Design and Mechanics

Effective gameplay systems rely on principles that structure player engagement, balance, and interaction. In *PacRun*, these concepts were central to gameplay tuning, power-up design, and user feedback mechanisms.

Salen and Zimmerman’s concept of “meaningful play” emphasizes that player actions should yield immediate and integrated outcomes [2]. This idea shaped the impact of power-ups, such as speed boosts and ghost-chasing states, which produce immediate audiovisual responses while altering the gameplay state.

Csikszentmihalyi’s flow theory [3] informed *PacRun*’s progression curve. Each level gradually increases in complexity through mechanical layering and spatial constraints, maintaining a balance between challenge and skill development to sustain immersion.

Power-ups were designed to alter the state temporarily, introducing strategic trade-offs. Cherries grant speed boosts, apples reverse controls while slowing ghost movement, and oranges enable ghost-chasing. These mechanics encourage players to evaluate short-term risks versus long-term rewards in real time.

Level design combines narrow corridors, junctions, and open zones to offer both precision-based and reactive navigation. Visual feedback and minimal HUD elements communicate score, lives, and active effects clearly, maintaining interface clarity without overwhelming the player.

2.4 Prior Research on Game Modifications and Enhancements

Game modifications and reimaginations typically fall into categories such as technical enhancements and mechanical additions. *Pac Run* adopts both modernising visuals while introducing modular power-ups and a new multiplayer mode.

In place of direct remakes, the project preserves the recognisable core loop of *Pac-Man* while introducing gameplay complexity through layered mechanics and AI variation. Visual enhancements retain the original aesthetic while integrating modern effects such as animated feedback and sprite transitions.

The menu system and HUD were designed to echo the clarity of the arcade original while incorporating modern usability practices. These include responsive buttons, animated transitions, and real-time HUD elements such as timers and combo indicators.

As a student-led adaptation, *Pac Run* also reflects educational design intent. The process of reinterpreting a legacy game within a structured, object-oriented framework serves as a valuable pedagogical exercise in applying software engineering theory to interactive systems.

Ethically, the project maintains originality by building all code from scratch and acknowledging the inspiration drawn from the original *Pac-Man*, without reusing assets or proprietary algorithms.

2.5 Technical Considerations in Java-Based Game Development

PacRun was developed entirely in Java, chosen for its portability, structured design principles, and educational relevance. Java's object-oriented features and platform independence make it a suitable language for structured game projects, especially within academic contexts.

To manage memory within Java's garbage-collected environment, the project used disciplined object reuse and pooling in performance-critical sections, especially for particle effects and collision logic. Rendering was handled using Java Swing, which, while not purpose-built for gaming, provided sufficient control over custom drawing via Graphics2D.

Ghost AI is implemented using a hybrid targeting strategy instead of computationally expensive algorithms like A*. Each ghost applies heuristic movement patterns aligned with its "personality," balancing behavioural variety with computational efficiency.

For input, *PacRun* combines Java's Key Listener with a continuous state tracker. This hybrid system maintains responsiveness and allows features like control reversal to be layered without modifying the core input handling logic. Input states are decoupled from raw key events to ensure clean behavioural switching and deterministic control under time-based mechanics.

3. Game Design

3.1 Conceptual Framework and Design Rationale

PacRun reimagines the 1980 arcade classic *Pac-Man* through modern software engineering, building on its foundational mechanics while introducing modular enhancements. Pellet collection and ghost evasion remain central, but gameplay depth is expanded through structured state management and object-oriented design.

Game design decisions are guided by the principle that player actions should yield immediate, visible outcomes, reinforcing the player's sense of agency [2]. Power-ups were designed to progressively alter mechanics: Cherries introduce speed boosts, Apples invert directional input

while slowing enemies, and Oranges temporarily enable ghost-hunting. These abilities are implemented as discrete state transitions layered within the object-oriented system.

Difficulty and skill acquisition were paced across four levels and a score-focused Bonus Round. Each level introduces a new mechanic, increasing cognitive and motor demands while building on prior mechanics. Interface feedback is used to reinforce state changes and help players anticipate and react appropriately [6].

Multiplayer is extended through the “Pellet Race” mode, where one player assumes the role of a ghost. Features like position-swapping and anti-camping are designed to encourage dynamic competition while maintaining the game’s accessible, non-violent tone.

3.2 Core Gameplay Features and Mechanics

PacRun retains the core arcade gameplay loop: navigate the maze, collect pellets, and avoid ghosts. Players progress by clearing all pellets per level, with points awarded for each action and bonuses for power-up use or successful ghost interactions. Lives are limited, with the game ending when all are lost.

The game introduces level-specific power-ups:

- Cherry (Level 2): Temporarily boosts movement speed for enhanced navigation and escape.
- Apple (Level 3): Inverts directional controls while reducing ghost speed, requiring adaptability.
- Orange (Level 4): Allows Pac-Man to chase ghosts temporarily, shifting roles and increasing reward potential.

The Bonus Round (Level 5) shifts focus to high-score optimisation. Pellets respawn and consecutive collection builds a multiplier, while ghost collisions deduct score and temporarily disable the player.

In Multiplayer “Pellet Race”, Pac-Man and a ghost are controlled by two players, competing to collect pellets or outscore each other. Position-swapping introduces unpredictability, while anti-camping logic discourages passive play and fosters active engagement.

3.3 Maze Structure, Level Design, and Environment

PacRun operates within a structured, two-dimensional, grid-based environment rendered through Java Swing. Each maze is statically defined and stored as an array of String objects (e.g., tile Map, level2Map), where characters represent tile types: 'X' for impassable walls, '.' for navigable corridors containing pellets, 'P' for Pac-Man's start, and 'r', 'p', 'b', 'o' for the four ghost spawns. Level-specific tiles ('C' for Cherry, 'A' for Apple, and 'G' for Orange) mark power-up placements, interpreted by the loadMap() method during initialisation. Overall game area dimensions (boardWidth, boardHeight) are calculated from columnCount, rowCount, and tileSize, ensuring tile consistency across levels.

The environment is centred using calculated offsetX and offsetY values to ensure consistent on-screen presentation regardless of resolution. This guarantees that gameplay is always framed around the maze centre without screen distortion.

Level design follows a staged competency acquisition model, enabling players to learn new mechanics incrementally through layered progression rather than radical maze changes:

Level 1 introduces the core gameplay loop: pellet collection and evading ghosts. This layout also serves the competitive multiplayer arena.

Level 2 adds Cherry power-ups, offering temporary speed boosts for advanced movement strategies.

Level 3 integrates Apple power-ups that invert control inputs, demanding cognitive flexibility, mitigated by slower ghost movement.

Level 4 introduces the orange power-up, allowing Pac-Man to chase ghosts, flipping the predator-prey relationship.

The Bonus Round shifts to a score-centric mode with respawning pellets and combo scoring. Ghost collisions deduct score rather than lives and introduce a short immunity period.

The maze architecture includes tight corridors for precise movement, junctions that encourage strategic choices, and open zones designed for reactive dodging and power-up exploitation. Static layouts provided a stable testbed for implementing and evaluating AI behaviours, pathfinding, and state transitions. However, the fixed design limits replayability across repeated sessions.

To address this, future versions could integrate procedural maze generation, dynamically creating varied environments while retaining critical structural patterns such as symmetry, choke points, and pellet density zones, features essential to maintaining gameplay integrity.

3.4 User Interface, Visual Aesthetics, and Accessibility

The user interface (UI), visual design, and accessibility features of *PacRun* are implemented using Java Swing, combining retro arcade aesthetics with modern usability standards.

The UI is composed of two core layers: the in-game HUD and the menu system. The HUD (Figure 3.4a) includes real-time score tracking, level indication, and remaining lives, visually represented by Pac-Man icons. During power-ups, animated labels and timer bars communicate status effects such as Speed Boost, Control Reversal, and Ghost Hunt. In multiplayer and Bonus Round modes, additional UI components display player-specific scores, combo multipliers, and countdown timers.



Figure 3.4a – In-Game HUD: Real-time score, lives, level info, power-up status, and combo tracking

The Main Menu and “How to Play” screen (Figure 3.4b) are designed to support intuitive navigation and onboarding. The menu features animated backgrounds, stylised Pac-Man fonts, and interactive buttons that respond to user input with glow and scaling effects. The "How to Play" screen visually presents core mechanics, power-up behaviours, multiplayer controls, scoring objectives, and difficulty progression, ensuring that both new and experienced player guided through the game’s systems.

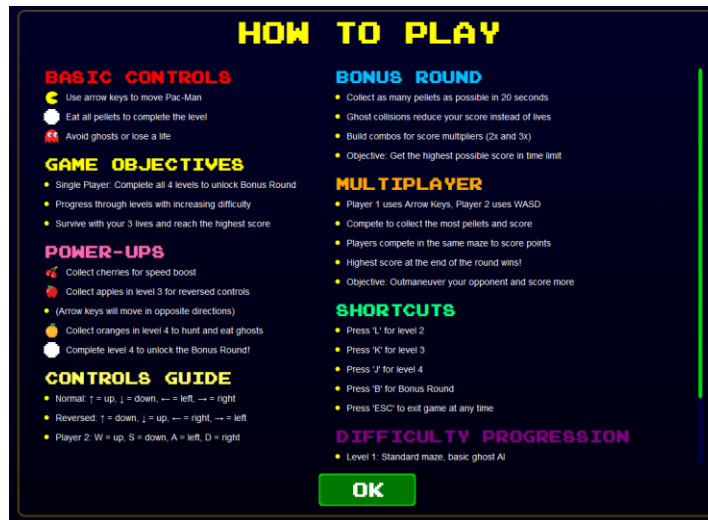


Figure 3.4b – "How to Play" Screen: Summary of controls, power-ups, multiplayer mode, and gameplay objectives.

Visually, *Pac Run* uses Java’s Graphics2D API for rendering. Key assets are represented via .png sprites and fallback vector drawings, maintaining recognisable shapes and colours inspired by the original *Pac-Man*. Dynamic elements such as flashing overlays, directional sprite changes, and event-triggered animations (e.g., ghost consumption or player death) enhance responsiveness and "game feel."

Accessibility was considered, with familiar keyboard controls (Arrow keys and WASD) and clear visual cues for gameplay events. However, areas such as control remapping, colourblind-friendly design, scalable UI text, and audio feedback remain future development goals to improve support for a wider range of players.

3.5 Ethical and Inclusive Game Design Considerations

Within the game design process, ethical considerations directly informed several key mechanics and visual elements of *PacRun*. The core design retains the non-violent premise of the original *Pac-Man*, focusing on abstract interactions rather than realistic conflict representation. This approach ensures appropriateness for diverse audiences whilst maintaining the essential gameplay tension.

In the competitive multiplayer "Pellet Race" mode, ethical design principles guided the implementation of balancing mechanisms to ensure fair play. The Position Swapping feature prevents gameplay stagnation by introducing controlled disruption at timed intervals, whilst the experimental Anti-Camping system (currently disabled for balance) was designed to discourage exploitative stationary tactics. Both mechanisms reflect a deliberate effort to create competition based on skill rather than exploitation of system limitations.

Basic accessibility considerations influenced the visual and control design, employing standard input schemes (Arrow Keys/WASD) and distinct visual elements for game entities. However, the current implementation has notable limitations regarding colour-dependent identification and a lack of customisation options. These limitations, along with their broader ethical implications and potential remediation, are examined more thoroughly in Section 6.5 (Project Context).

The game design also deliberately avoids potentially problematic engagement mechanisms common in contemporary games, such as exploitative monetisation systems or designs that foster compulsive play patterns. Instead, PacRun employs session-based gameplay with clearly defined progression, prioritizing player agency and well-being within its arcade-inspired structure.

4. Technical Implementation

4.1 Selection of Programming Languages, Tools, and Frameworks

Java was selected as the primary language for *PacRun* due to its strong alignment with the project's educational and architectural goals. As a robustly object-oriented language, Java facilitated the demonstration of OOP principles such as encapsulation (e.g., Block class), inheritance, and polymorphism, aligning directly with the project's architectural exploration. Its platform independence via the Java Virtual Machine (JVM) further supported academic distribution by ensuring cross-platform compatibility without requiring code modification.

Java's standard library offered built-in support for core functionality, including data structures like ArrayList and HashSet for managing game entities, GUI components, and event handling. This reduced reliance on external libraries and streamlined development. Java's automatic garbage collection simplified memory management, though it introduced the need for performance considerations during collection cycles.

Visual rendering and user interaction were implemented using Java Swing, chosen for its native integration with the Java ecosystem. Swing enabled focus on game architecture without introducing third-party engines, supporting tasks such as rendering (JPanel, paintComponent, Graphics2D), input handling (KeyListener), and timing (javax.swing.Timer). While Swing is not optimised for high-performance game development, it proved adequate for a 2D grid-based

project like *PacRun*, with performance optimisations required in rendering and logic loops to mitigate its limitations.

Supporting tools included the Java Development Kit (JDK), an IDE (such as IntelliJ IDEA), and Git for version control, hosted on GitLab. These provided a complete, academically appropriate development stack that prioritised maintainability, architectural clarity, and portability.

4.2 Software Architecture and Structural Design

The software architecture of *PacRun* was purposefully designed around Object-Oriented Programming (OOP) principles, reflecting a primary objective of the project: to explore and demonstrate the application of these principles within the context of game development using Java [4]. The resulting structure prioritises modularity, encapsulation, and maintainability, facilitating the implementation of the game's diverse features. Whilst not adhering to a rigidly enforced architectural pattern, the design broadly follows the separation of concerns encouraged by the Model-View-Controller (MVC) approach, adapted pragmatically for use within the Java Swing framework.

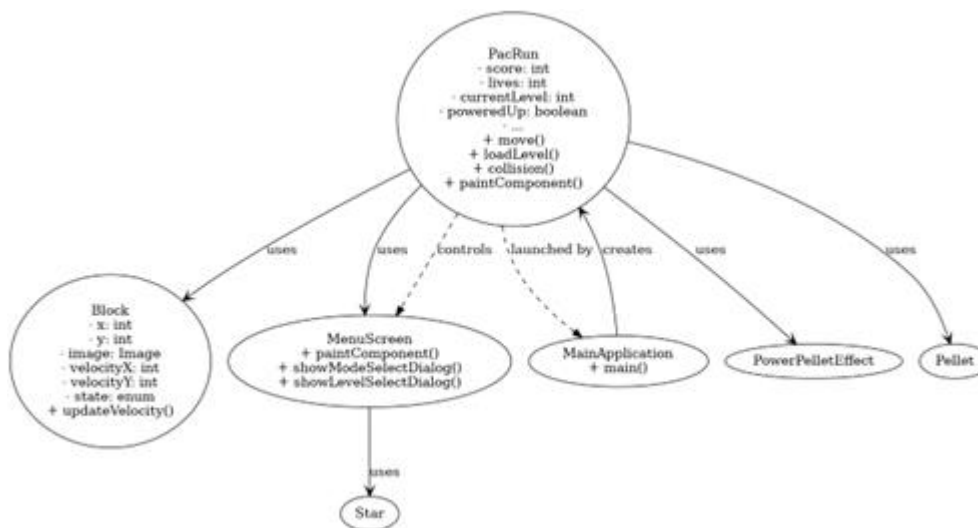


Figure 4.1 – Class structure for *PacRun* showing primary components and their relationships.

As illustrated in Figure 4.1, the architecture follows a modified MVC pattern:

Model: The core game state and logic reside primarily within the *PacRun* class. This encompasses game status variables (`score`, `lives`, `currentLevel`), state flags (`gameOver`, `poweredUp`, etc.), and collections managing game entities. The *Block* inner class is central to the design—encapsulates entity attributes (position, image, velocity, state) and behaviours, promoting code reuse across different entity types. Game rules and state transitions are implemented through methods such as `move()`, `collision()`, and `loadLevel()`.

View: Visual representation is handled primarily through the `paintComponent(Graphics g)` method within the `PacRun` class, which renders entities, HUD elements, and animations. The separate `MenuScreen` class provides the interactive main menu interface.

Controller: User input management occurs through the `KeyListener` implementation in `PacRun`, which interprets keyboard events and modifies state variables accordingly. The game loop, driven by a `javax.swing.Timer` instance, orchestrates the update-render cycle through its `actionPerformed()` method, while `MainApplication` serves as the initial application controller.

Several key structural elements enhance this architecture:

Object-Oriented Design: The `Block` class demonstrates encapsulation by combining data and behaviour, while additional classes (`MenuScreen`, `Star`, `PowerPelletEffect`, `Pellet`) show modular design through the separation of responsibilities.

Game Loop: A fixed-step implementation using `javax.swing.Timer` ensures consistent timing for physics and AI updates, decoupled from rendering [5].

State Management: Boolean flags (`poweredUp`, `controlsReversed`) and timestamp variables within the `PacRun` class manage various game states, with the `move()` method applying the correct behaviours based on current conditions.

Entity Management: The Java Collections Framework (`HashSet`, `ArrayList`) efficiently manages game entities, with `loadMap()` populating these collections and `move()` handling updates and collisions.

This architecture successfully leverages OOP to create a structured and extensible foundation, allowing for the incremental implementation of complex features while providing a clear framework for managing interactions between game entities and states.

4.3 Player Control Systems and Movement Mechanics

PacRun's character control system is designed to support responsive inputs and grid-aligned movement, critical for maze-based gameplay. User input is managed using Java's `KeyListener` interface. In single-player mode, the Arrow Keys control Pac-Man, while multiplayer assigns WASD to the player-controlled ghost. Directional inputs are stored in `requestedDirection` and `ghostRequestedDirection`, enabling concurrent movement logic for both entities.

To ensure smooth navigation, the system uses buffered input: directional changes are not applied instantly but are queued until the character reaches a valid turning point. This mechanic allows players to "pre-load" turns at junctions, improving flow and reactivity.

Turning is handled by the `tryTurn()` and `tryGhostTurn()` methods, which first verify alignment using `isAligned()` and then snap the character to the centre of the tile perpendicular to the turn. If the path is valid, direction is updated; otherwise, the character continues in the current direction. This logic ensures that turning only occurs at valid points and maintains consistent

movement across grid tiles. The structure of this logic is illustrated in Figure 4.3a, showing how directional updates depend on alignment and current input.

```
void tryTurn() {
    if (pacman == null) return;

    char originalDirection = pacman.direction;
    boolean canTurn = false;

    // First, check if Pac-Man is aligned with the grid for turning
    boolean alignedX = isAligned(pacman.x, tileSize);
    boolean alignedY = isAligned(pacman.y, tileSize);

    // Determine if we can turn based on alignment and requested direction
    if ((requestedDirection == 'U' || requestedDirection == 'D') && alignedX) {
        canTurn = true;
        // Slightly adjust X position to align with grid
        pacman.x = Math.round(pacman.x / tileSize) * tileSize;
    } else if ((requestedDirection == 'L' || requestedDirection == 'R') && alignedY) {
        canTurn = true;
        // Slightly adjust Y position to align with grid
        pacman.y = Math.round(pacman.y / tileSize) * tileSize;
    }
}
```

Figure 4.3a – Simplified Turn Logic for Pac-Man

Movement is processed within the game loop via `actionPerformed()`, calling the `move()` method to update position incrementally. Velocity is calculated in `updateVelocity()` as a proportion of `tileSize`, allowing for consistent, tile-based movement. Power-ups such as the Cherry adjust speed by modifying this velocity. Wall collisions are resolved by reversing movement updates where necessary.

The control system also accommodates gameplay state changes. For instance, reversed controls triggered by the Apple power-up remap input at runtime, while speed boost logic dynamically adjusts movement calculations. These layers integrate without disrupting the base mechanics, maintaining consistent handling throughout.

4.4 Artificial Intelligence: Enemy Pathfinding and Behaviour

The ghost antagonists within PacRun are governed by an AI system that synthesizes deterministic pathfinding with varied targeting strategies and state-based behavioural adaptations. The system draws inspiration from the original *Pac-Man*'s concept of ghosts having distinct behavioural patterns, adapted here into unique AI targeting strategies for each ghost. Creating opponents that balance predictability for strategic engagement with sufficient variability to maintain challenge.

The pathfinding capability is encapsulated within the `getGhostDirection(Block ghost, Block target)` method, which determines the optimal direction for a ghost to pursue its designated target. Using the same `isAligned()` mechanism employed for player movement, ghosts make directional decisions primarily at grid intersections. The method evaluates potential moves by:

- Checking viability against maze structures using `isWallAt()`
- Computing Euclidean distance to target via `calculateDistance()`
- Selecting the direction minimizing this distance

A critical heuristic prevents oscillatory behaviour by disallowing immediate direction reversals unless no other path exists. Limited randomness (1/20 probability) introduces non-determinism, allowing ghosts to occasionally select sub-optimal but valid directions. This slight randomness was introduced to enhance unpredictability while maintaining consistent pursuit behaviour.

The four ghost types exhibit distinct behaviours through variations in their target selection logic:

Ghost Type	Personality	Targeting Strategy
Red	'Blinky'	Direct pursuit of Pac-Man's current position
Pink	'Pinky'	Ambush by targeting position ahead of Pac-Man's movement
Blue	'Inky'	Unpredictable alternation between pursuit and random wandering
Orange	'Clyde'	Proximity-dependent behaviour: pursuit when distant, scatter when close

The AI behaviour dynamically adapts to game state:

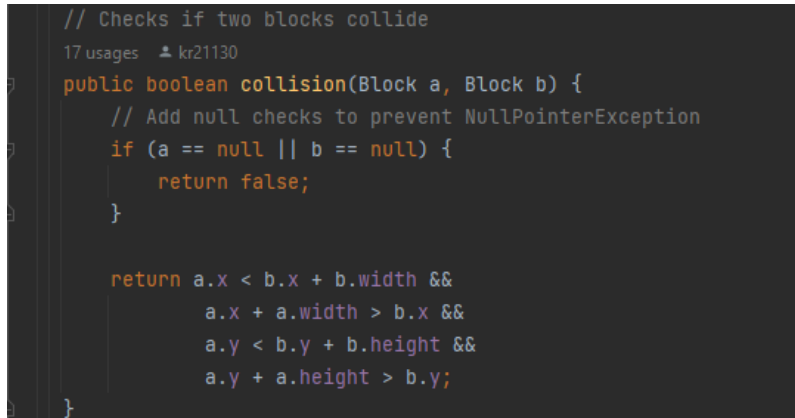
- Vulnerable State (`poweredUp = true`): Ghosts flee from Pac-Man by targeting positions opposite his location
- Slowed State (`ghostsSlowed = true`): Movement speed temporarily halves during Apple power-up effects
- Eaten State (`ghost.eaten = true`): Ghost becomes non-interactive for a defined period before respawning

This state-based adaptation integrates with the same movement mechanics used for player control, with ghosts' velocity and position updates subject to identical collision and grid alignment constraints. In multiplayer mode, the player-controlled ghost bypasses this AI system entirely, relying exclusively on the `ghostRequestedDirection` and `tryGhostTurn()` mechanisms described in Section 4.3.

The combination of consistent movement mechanics across all entities with differentiated AI behaviours for computer-controlled ghosts creates a coherent yet varied gameplay experience that rewards strategic player decision-making.

4.5 Collision Detection and Optimisation Strategies

Accurate collision detection was central to maintaining PacRun’s gameplay integrity, particularly given its grid-based maze structure and real-time interactions. The game uses an Axis-Aligned Bounding Box (AABB) collision approach, selected for its simplicity and efficiency within a 2D context. The main collision method checks for rectangular overlap by comparing axis boundaries:

A screenshot of a code editor showing a Java method named 'collision' that takes two 'Block' objects as parameters. The code is written in a dark-themed editor with syntax highlighting. It includes a comment at the top, a null check, and a series of logical AND conditions to check for axis-aligned bounding box overlap.

```
// Checks if two blocks collide
17 usages  📌 kr21130
public boolean collision(Block a, Block b) {
    // Add null checks to prevent NullPointerException
    if (a == null || b == null) {
        return false;
    }

    return a.x < b.x + b.width &&
           a.x + a.width > b.x &&
           a.y < b.y + b.height &&
           a.y + a.height > b.y;
}
```

Figure 4.5.1 – Axis-Aligned Bounding Box Collision Method

This method, used within the `move()` loop, governs all primary collision cases—Pac-Man with walls (movement blocked), pellets (collection and scoring), power-ups (state flags triggered), and ghosts (depending on powered state, either loss of life or ghost neutralisation). Ghosts also undergo separate collision checks to avoid penetrating walls or overlapping during vulnerable states.

To preserve responsiveness in Java Swing's event-driven model, several optimisation techniques were applied:

- Selective collision testing: Only relevant pairs (e.g., Pac-Man with pellets, ghosts with maze) are checked, avoiding unnecessary pairwise comparisons.
- HashSet collections: Used to store dynamic entities like pellets, enabling fast removal post-collision ($O(1)$ complexity), reducing redundant checks per frame.
- Minimal object allocation: Core functions like `move()` and `collision()` avoid creating temporary objects, helping to reduce memory churn and maintain frame stability during intensive gameplay.
- Conditional logic gating: Collision evaluations are skipped for inactive states (e.g., ghosts in eaten state are excluded unless powered-up mode is active).
- Grid alignment: All movement is snapped to discrete tile increments, reducing corner collision ambiguity and simplifying spatial reasoning.

Although more advanced spatial partitioning (e.g., quadtrees) could optimise for higher entity counts, the implemented AABB approach, paired with focused memory management and game-specific heuristics, proved sufficient to maintain consistent performance and responsive controls in the Swing framework.

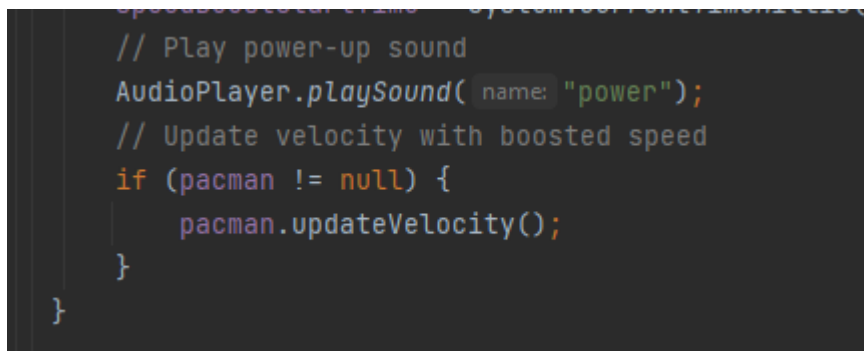
4.6 Sound Design, Visual Effects, and Enhancing Immersion

PacRun uses a combination of responsive audio cues and targeted visual effects to enhance player immersion, provide critical gameplay feedback, and reinforce the arcade aesthetic of the original Pac-Man. These systems are designed with performance constraints in mind while prioritising clarity and player responsiveness.

Audio Feedback and Sound Effects :

Sound cues immediately confirm critical player actions without distracting from fast-paced play. The `AudioPlayer` class triggers dedicated sound effects in response to gameplay events—most notably, power-up collection, ghost collisions, pellet pickups, and death sequences. This design choice allows for real-time confirmation of state changes, helping players remain aware of context without needing to glance at on-screen indicators.

As shown in Figure 4.6a, a successful power-up collection triggers the `playSound("power")` method:



```
// Play power-up sound
AudioPlayer.playSound( name: "power");
// Update velocity with boosted speed
if (pacman != null) {
    pacman.updateVelocity();
}
}
```

Figure 4.6a: Invocation of a power-up sound effect

Power-Up Duration Indicators :

To avoid ambiguity about the duration of temporary power-ups, PacRun implements a horizontal timer bar that visually depletes as time expires. This empowers players to make tactical decisions—whether to chase ghosts under the orange power-up or conserve momentum during speed boosts—based on the remaining duration of the effect.

As shown in Figure 4.6b, this logic uses elapsed time to compute and render the remaining bar width:

```
// Draw power-up timer bar
long elapsed = System.currentTimeMillis() - powerUpStartTime;
long remaining = POWER_UP_DURATION - elapsed;
int barWidth = (int) (remaining * 100 / POWER_UP_DURATION);
g.fillRect(x: offsetX + 10, y: offsetY + tileSize, barWidth, height: 8);
```

Figure 4.6b: Rendering a horizontal power-up timer bar that shrinks as the effect nears expiry

Visual Feedback for Reversed Controls :

The Apple power-up temporarily reverses directional controls. To prevent confusion or perceived mechanical error, a pulsing purple overlay is displayed along the screen borders, clearly distinguishing the reversed state without obstructing vision.

This effect is conditionally rendered based on the “controlsReversed” flag, as shown in Figure 4.6c:

```
// Display reversed controls indicator
if (controlsReversed) {
    // Add a purple border to indicate reversed controls
    g.setColor(new Color( r: 128, g: 0, b: 128, a: 128));
    g.fillRect(offsetX, offsetY, boardWidth, height: 5); // Top border
    g.fillRect(offsetX, y: offsetY + boardHeight - 5, boardWidth, height: 5); // Bottom border
    g.fillRect(offsetX, offsetY, width: 5, boardHeight); // Left border
    g.fillRect(x: offsetX + boardWidth - 5, offsetY, width: 5, boardHeight); // Right border
}
```

Figure 4.6c: Top border overlay indicating active control reversal state

Death Animation and Game Over Transitions :

To prevent abrupt resets and create emotional pacing between lives, PacRun implements a two-phase death animation. The player’s avatar collapses into an expanding circle of fading opacity, visually conveying the consequence of life loss. As illustrated in Figure 4.6d, this effect leverages alpha blending and growing radius:

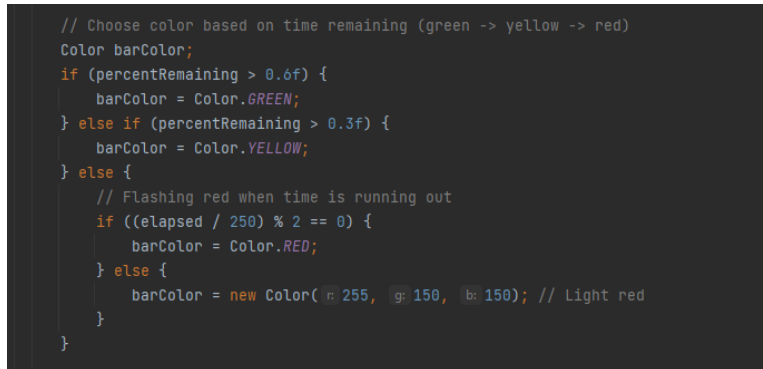
```
g2d.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER, opacity));
g2d.setColor(Color.YELLOW);
g2d.fillOval(x: centerX - size / 2, y: centerY - size / 2, size, size);
}
}
```

Figure 4.6d: Fade-based visual effect signalling Pac-Man's death.

Bonus Round: Combo System and Urgency Feedback :

The Bonus Round incentivises score maximisation within a strict time limit. To signal urgency, a timer bar changes colour from green to yellow, then flashes red in the final seconds. This mechanic creates a rising sense of pressure that enhances focus and responsiveness.

As shown in Figure 4.6e, the timer's colour logic is dynamically computed:



```
// Choose color based on time remaining (green -> yellow -> red)
Color barColor;
if (percentRemaining > 0.6f) {
    barColor = Color.GREEN;
} else if (percentRemaining > 0.3f) {
    barColor = Color.YELLOW;
} else {
    // Flashing red when time is running out
    if ((elapsed / 250) % 2 == 0) {
        barColor = Color.RED;
    } else {
        barColor = new Color(0.255f, 0.150f, 0.150f); // Light red
    }
}
```

Figure 4.6e: Bonus Round timer bar that changes colour to reflect urgency

5. Modifications and Enhancements from the Original Pac-Man

5.1 Visual and Technical Enhancements

PacRun introduces significant visual and technical advancements compared to its arcade predecessor. While the original *Pac-Man* operated on hardware-constrained, low-resolution pixel displays, *PacRun* uses Java Swing's 2D graphics capabilities to deliver higher-resolution sprites, smoother animations, and richer visual effects. Static maze layouts, character sprites, and power-up icons are rendered using modern .png assets, preserving recognisability while improving visual clarity. Effects such as screen flashes, dynamic state overlays, and animated transitions for power-ups and deaths improve feedback and reinforce the game's responsiveness.

The user interface also reflects a substantial upgrade. *PacRun*'s animated main menu and interactive level selection screen offer a more modern user experience, while the in-game HUD provides real-time status updates for lives, scores, and active power-ups. Compared to the original's minimalistic display, this interface increases clarity and situational awareness during gameplay.

Underpinning these visual improvements is a shift from hardware-specific procedural code to a modular, object-oriented architecture built in Java. The use of encapsulated classes, dynamic collections, and a loosely MVC-aligned design allows greater extensibility and maintainability. This architecture supports modular power-up logic, AI states, and multiple game modes

developments that would have been impractical under the original's fixed, assembly-level codebase.

5.2 Gameplay Innovations and Player Experience

While maintaining the iconic pellet-collection and ghost-evasion loop, *PacRun* introduces strategic depth through three level-specific power-ups (Cherry for speed, Apple for control reversal, Orange for ghost-hunting), each encouraging distinct player strategies. Level progression focuses on mechanic introduction rather than speed escalation, creating a structured learning curve. The Bonus Round transforms gameplay with time-limited score attacks and point-based penalties instead of life deductions.

The multiplayer "Pellet Race" mode shown in Figure 5.2 below represents the most significant innovation, shifting from AI evasion to player-versus-player spatial competition, balanced through position swapping and anti-camping mechanics. Enhanced ghost AI introduces context-aware behaviour, requiring dynamic tactical adaptation rather than pattern memorisation.



Figure 5.2 – Multiplayer Mode

6. Project Management and Reflection

6.1 Project Planning and Methodology

Pacruns's development followed a hybrid methodology combining phased milestones with iterative feature cycles, managed through Jira for task tracking and Git for version control. This approach balanced structured progress with flexibility to refine mechanics and address emergent challenges.

The project progressed through four distinct phases:

Foundation Phase (Weeks 1-4): Core architecture including Block class, rendering, and collision

Gameplay Phase (Weeks 5-8): Level 1 implementation, ghost AI, and Cherry power-up

Enhancement Phase (Weeks 9-12): Additional power-ups, Bonus Round, and multiplayer mode

Refinement Phase (Weeks 13-15): Testing, optimisation, and documentation

Tasks were organized under high-level epics corresponding to major features (ghost AI, power-ups, multiplayer, UI), then decomposed into granular, traceable subtasks in Jira. This enabled focused implementation while maintaining visibility of interdependencies. The Cumulative Flow Diagram (Figure 6.2) illustrates task throughput, showing steady foundational development followed by accelerated completion during the enhancement and refinement phases.

Time management centered on prioritizing core functionality before enhancements, with 15% buffer allocation that proved essential when integrating complex power-up states and debugging multiplayer controls. The initial architectural decisions—particularly the extensible Block class and Boolean-flag state management—facilitated modular implementation of diverse behaviours without structural rewrites, even accommodating late-stage additions like the multiplayer mode.

This hybrid approach proved effective, with phased delivery maintaining alignment to milestones while iterative cycles provided flexibility for refinement. The combination of structured planning and adaptable execution allowed for the successful navigation of technical complexity while delivering a cohesive final product.

6.2 Git Version Control

Version control using Git, hosted on the University of Essex CSEE GitLab platform, was integral to PacRun's development. In addition to structured change tracking, Git provided a reliable workspace for backing up code, ensuring that progress was never lost, especially during high-risk feature development.

The repository was logically structured, separating source code (/src), test classes (/test), and game assets. This facilitated modular development, particularly when implementing complex systems like multiplayer mechanics or power-up effects, while maintaining synchronisation between logic and visual elements.

Commit practices matured over time. Early commits were minimal, whereas later entries included detailed implementation notes and rationale. This supported debugging efforts, such as refining AI behaviour or resolving power-up interaction conflicts. The commit history aligns with the phased development timeline (Section 6.1), with foundational commits during Oct–Nov 2024 and feature completion in Mar–Apr 2025. Examples of commits, branches, and progress are included in Appendix B, documenting how Gitlab supported visual tracking and tracking regarding consistent commits and workflow.

A disciplined single-branch workflow ensured a stable codebase, with each commit tested before integration. While this approach suited the project's scope, future work might benefit from feature branching to isolate experimental changes.

A key milestone was integrating the unit testing framework into version control. These committed supported project robustness and documented functional correctness across key systems, reinforcing both code reliability and professional development practice.

6.3 Jira

Jira supported the hybrid phased-iterative methodology used during PacRun's development, offering structured task management through a three-column workflow ('To Do', 'In Progress', 'Done'). This lightweight system enabled focused individual development while allowing flexibility for iteration and refinement.

Tasks were grouped into high-level epics aligned with core features—ghost AI, power-ups, multiplayer mode, and UI—and broken into granular, traceable units. Features like the Apple power-up and ghost targeting logic evolved through multiple tickets, reflecting iterative enhancements, UI tweaks, and bug fixes informed by playtesting.

Development velocity is visualised in the Cumulative Flow Diagram (Figure 6.2), showing steady task completion through early 2025 and a surge in March–April during the multiplayer and Bonus Round phases. Examples of task lists, timeline views, and progress charts are included in Appendix A, documenting how Jira supported visual tracking and temporal planning.

Jira proved especially valuable during the polish phase, where feedback was logged as individual tasks and linked to corresponding epics. Despite requiring ongoing maintenance, the platform streamlined prioritisation, provided detailed historical insight, and directly supported structured development and post-project evaluation.

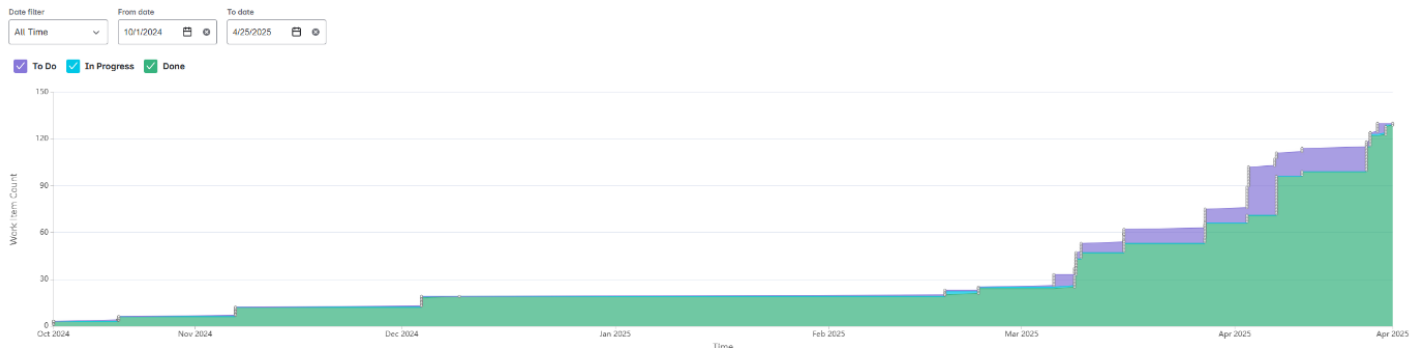


Figure 6.2(Cumulative Flow Diagram)

6.4 Risk Management

Proactive risk management was incorporated into the project planning process to identify potential threats to the project's successful completion, developer well-being, and end-user safety, allowing for the implementation of appropriate mitigation strategies. A formal Risk Assessment was conducted early in the development cycle [See Appendix X.Z for Risk Assessment Document], evaluating potential hazards, their consequences, likelihood, and necessary control measures.

Several key risk areas were identified. Firstly, developer health associated with extended screen time during intensive development and testing periods was recognized. The potential consequence was developer fatigue or strain. Initial controls included incorporating regular breaks and ensuring adequate lighting, assessed as leading to a 'Low' current risk. Additional mitigation involved consciously scheduling breaks using timer software to enforce adherence, further reducing the residual risk to 'Very Low'. This was managed effectively throughout the project via disciplined work scheduling.

Secondly, a potential risk to end-users with photosensitive conditions was identified concerning visual effects, specifically the flashing elements potentially present in animations like the death sequence. The initial control involved limiting animation speeds. However, to further mitigate this 'Low' risk, an additional control measure planned was to ensure animations complied with accessibility guidelines regarding flash rates and intensity. During development, the death animations (drawDeathAnimation) were implemented using fading and expanding effects rather than rapid, full-screen flashes, aligning with this principle and keeping the residual risk 'Low'.

Thirdly, the inherent risk of software errors and bugs causing unexpected behaviour impacting end-users was assessed. Initial controls involved basic developer testing during implementation, but the potential impact led to an initial 'Medium' risk rating. The crucial additional control identified was the implementation of a more comprehensive testing protocol, including the unit testing strategy (Chapter 8) and structured user/playtesting sessions (Section 8.2). The successful execution of unit tests and iterative bug fixing based on testing (evidenced by 'Debug' and 'Fix' tasks in Jira, Section 6.3) significantly reduced the likelihood of critical errors in the final version, lowering the residual risk to 'Low'.

Hazard (H) hazardous event (HE) consequence (C)	Who might be harmed	Current controls	Current risk LxC=R	Additional controls needed to reduce risk	Residual risk LxC=R	Target Date	Date achieved
Extended screen time during development and testing	Developer	Regular breaks, proper lighting	Low	Schedule regular breaks using timer software	Very Low	Immediate	01/02/2025
Photosensitive content in game (flashing effects during death animation)	Users with photosensitive conditions	Limited animation speeds	Low	ensure animations comply with accessibility guidelines	Low	Before final submission	09/03/2025

Ra-blank_v2 08/19 1

Consequence	Catastrophic	Medium	High	Very High	Very High	Very High
	Major	Low	Medium	High	High	High
	Moderate	Very low	Low	Medium	Medium	High
	Minor	Very low	Low	Low	Medium	Medium
	Insignificant	Very low	Very low	Low	Low	Low
R = LxC	Very unlikely	Unlikely	Each likely	Likely	Very likely	
		Likelihood of hazardous event				

Figure 6.1: Risk Assessment Matrix Used for Hazard Identification and Mitigation Planning

Finally, the risk of data loss during development due to hardware failure or accidental deletion was identified as a significant threat to project continuity, rated initially as 'Medium'. The primary control was regular manual saving. However, the essential additional control implemented was the rigorous use of Git version control hosted on GitLab (Section 6.2), providing distributed backups and detailed history. Commits were made frequently, often multiple times per development session, ensuring minimal potential data loss and providing effective mitigation, reducing the residual risk to 'Very Low'.

Reflecting on the effectiveness of this risk management process, the early identification of these potential issues allowed for the integration of mitigation strategies directly into the workflow. The planned controls, particularly the comprehensive testing protocol and the consistent use of Git, proved highly effective in managing the most significant technical and continuity risks. While the risk related to extended screen time required ongoing personal discipline, the proactive consideration of photosensitive epilepsy risks influenced the design of visual effects from the outset. Overall, the formal risk assessment provided a valuable framework for anticipating challenges and implementing robust control measures, contributing to the project's stability and successful completion.

6.5 Project Context

The development of PacRun, while primarily focused on technical objectives related to object-oriented architecture and state management, was necessarily situated within a broader context encompassing legal, ethical, health and safety, and Equality, Diversity, and Inclusivity (EDI) considerations. A conscious reflection upon these elements was maintained throughout the project lifecycle, influencing design decisions, development practices, and the evaluation of the final artefact.

From a legal standpoint, the project's direct inspiration from the seminal arcade game Pac-Man mandated careful consideration of Intellectual Property (IP) rights. To navigate this responsibly within an academic framework, PacRun was explicitly developed as an original software implementation exploring game design principles, rather than an attempt to replicate or distribute proprietary code or assets belonging to the IP holder (Namco Bandai Entertainment). All source code was authored independently, and assets such as graphics and sound were either bespoke creations or appropriately sourced, with the understanding that formal licensing would be required for any use beyond this academic demonstration. This ensured the project was conducted ethically and respectfully toward the original IP. This approach allowed for legitimate technical exploration whilst endeavouring to respect the original creators' rights.

Ethical considerations guided several key design choices. Adhering to the precedent set by the original, the gameplay was intentionally kept non-violent, focusing on abstract goals of collection and evasion suitable for a wide audience. In designing the competitive multiplayer mode, fair play was a significant concern. Mechanisms such as the Position Swapping feature were implemented to introduce dynamic balancing elements. While an anti-camping system was also developed, it was ultimately disabled in the final iteration due to balancing challenges identified during playtesting, reflecting an iterative ethical consideration for player experience within the competitive context. Furthermore, the project eschews contemporary commercial practices, often subject to ethical scrutiny, such as monetisation mechanics or user data collection, focusing purely on the gameplay experience itself.

Health and Safety (H&S) aspects were addressed for both the developer and potential end-users. Developer well-being during prolonged coding sessions was managed through awareness of ergonomic practices and the self-imposition of regular breaks, mitigating risks associated with extended screen time identified in the project's risk assessment. For end-users, the potential risk associated with photosensitive epilepsy due to flashing visual effects was acknowledged. While specific compliance testing against formal standards (e.g., WCAG Harding test) was beyond the project's scope, the design of animations, particularly the death sequences, consciously favoured fading and expansion effects over high-frequency, full-screen flashes as a mitigation strategy, maintaining the residual risk at an assessed low level.

Finally, Equality, Diversity, and Inclusivity (EDI) were considered, with particular emphasis on accessibility. The game aims to appeal broadly through its culturally neutral theme and standard keyboard controls (Arrow Keys/WASD). However, a critical reflection acknowledges significant limitations in accessibility within the current implementation. Reliance on colour differentiation

to distinguish ghosts may disadvantage players with colour vision deficiencies. Additionally, the absence of remappable controls, adjustable text sizes or contrast settings, and audio alternatives restricts accessibility for individuals with visual or motor impairments. While these features were beyond the defined technical scope (Section 1.3), their omission represents a key shortcoming from an EDI perspective. Addressing these gaps remains a priority recommendation for future development (Section 7.3), ensuring the game is inclusive and accessible to a wider range of players.

6.6 Overall Achievements and Learning Outcomes

The PacRun project culminated in a fully playable, feature-rich Java-based game that met and, in several areas, exceeded the initial objectives set out in Section 1.2. Developed entirely using Java and Swing without reliance on an external game engine, it represents a significant technical and educational achievement in game architecture, systems design, and object-oriented implementation.

From a development perspective, the project delivered a complete single-player experience across four structured levels, each introducing progressively complex power-up mechanics: speed boosts, input reversal, and ghost-chasing. These mechanics were layered using a flag- and timestamp-driven state management system, enabling smooth concurrent behaviours. A Bonus Round further expanded the game's scope with respawning pellets and a score multiplier system, shifting the gameplay focus to combo chaining and high-efficiency routing.

The multiplayer "Pellet Race" mode emerged as the most ambitious and technically demanding addition. Not originally scoped, it was successfully integrated late in development, requiring separate control logic, balancing systems like position swapping, and real-time competitive mechanics. Its successful delivery demonstrated not only design adaptability but also architectural extensibility.

Key technical achievements included the creation of AI ghost personalities with variable targeting behaviour, buffered grid-aligned movement for responsive player control, animated UI elements, and real-time power-up visual feedback. Performance optimisations within the constraints of Java Swing, such as reduced object creation and selective collision checks, enabled stable gameplay even as feature complexity grew.

From a learning perspective, PacRun provided hands-on experience with real-time systems, modular object-oriented design, and state management. Complex features like dynamic input handling, AI-driven enemy behaviour, and multiplayer logic were developed and iteratively refined. Debugging AI edge cases and integrating simultaneous power-up states deepened the understanding of concurrency and timing control in game environments.

Beyond the code, the project also reinforced skills in agile planning, Jira-based task management, and structured decision-making under pressure. It provided insight into ethical considerations, IP boundaries, and accessibility challenges areas often underemphasised in technical projects.

Ultimately, building *PacRun* from foundational tools rather than higher-level engines resulted in a far more comprehensive grasp of software engineering practice and the multifaceted nature of game development.

7. Challenges, Solutions, and Technical Optimisations

7.1 Technical Challenges, Debugging, and Performance Optimisation

The development of *PacRun* presented significant technical challenges, particularly due to its implementation in Java Swing rather than a dedicated game engine, and its expanded feature scope compared to the original *Pac-Man*. These issues were addressed through iterative debugging, structural refinements, and performance optimisations.

Managing concurrent player inputs during multiplayer was a key challenge. Java Swing's single-threaded `KeyListener` model risked input crosstalk between Pac-Man and the player-controlled ghost. This was resolved by separating input handling into distinct variables and turning methods, ensuring responsive and independent control for both entities within the same game tick.

Grid-based movement and collision detection also required extensive tuning. Early versions suffered from tile-edge clipping and directional misalignment. These were corrected by refining tolerance thresholds and reordering logic to prioritise direction changes before overlap. Static map validation proved essential, as minor definition errors disrupted gameplay.

Ghost AI complexity increased substantially from early prototypes. Final implementation used personality-based behaviours, direct pursuit, prediction, randomness, and proximity-based logic managed through dynamic state transitions. Vulnerability and respawn states were integrated, and stepwise debugging supported the development of stable, responsive ghost movement.

Gameplay refinements included smoother movement, buffered turning at intersections, and clear visual cues for power-up states. Power-up balancing was achieved through repeated playtesting and adjustments to timing constants.

Performance optimisation was crucial. To reduce Swing's rendering overhead, memory usage was minimised by reusing objects and avoiding unnecessary allocations in core loops. Logical checks were streamlined, and lightweight structures like `HashSet` ensured efficient entity tracking. These measures maintained responsive gameplay even under garbage-collected JVM conditions.

Testing underpinned all improvements. Breakpoints, runtime tracing, and JUnit tests verified collision, movement, and state logic, supporting a stable final product despite increased complexity.

7.3 Future Recommendations

While PacRun met its core technical and gameplay objectives, development highlighted several areas for meaningful future improvement. These recommendations focus on enhancing scalability, replayability, presentation, and inclusivity.

A key technical recommendation is migrating the project from Java Swing to a dedicated 2D game development framework such as LibGDX or Godot. This would address limitations in rendering performance, animation fidelity, and audio integration. Hardware-accelerated graphics and built-in support for sprite sheets, shaders, and sound engines would enable smoother gameplay and richer visual and audio feedback.

To improve replayability, future iterations could implement procedural maze generation. The current use of static maps limits novelty across sessions. Algorithms like Recursive Backtracking could dynamically generate mazes per run, preserving challenge while increasing variety. Complementing this, ghost AI could be evolved with more advanced pathfinding (e.g., A*) and adaptive difficulty systems that respond to player skill.

The multiplayer “Pellet Race” mode could be expanded beyond local play through the development of networked multiplayer. Implementing a client-server architecture would introduce competitive features like online matchmaking and high score leaderboards, significantly broadening engagement.

Accessibility remains a priority for future work. Enhancements may include remappable controls, scalable text, high-contrast UI modes, and alternatives to colour-based status indicators. Migrating to a modern engine would also facilitate the integration of audio cues to support players with visual impairments.

Together, these recommendations establish a clear roadmap for transforming PacRun into a more dynamic, inclusive, and technically robust game experience.

8. Testing and Evaluation

8.1 Testing, User Evaluation, and Gameplay Refinement

Testing and quality assurance in PacRun were implemented through a combination of automated unit testing, integration checks, playtesting, and iterative refinement. These strategies ensured both functional correctness and a polished gameplay experience, despite the constraints of the Java Swing framework.

Automated unit testing was conducted using JUnit 4, focusing on validating collision logic, power-up state transitions, movement control, score updates, and AI targeting. Key methods like

`collision()`, `tryTurn()`, `activateSpeedBoost()`, and `getGhostDirection()` were rigorously tested through the `PacRunTest` class. Additional classes, including `AudioPlayerTest`, verified system states like audio toggling. To support testability, several private methods and fields within the `PacRun` class were modified to package-private access, allowing controlled interaction from the test environment. These tests formed the project's regression suite and were executed frequently during development iterations.

Integration testing occurred continuously. New features were immediately embedded into the game loop and verified through gameplay. For example, activating an orange power-up was tested to confirm correct AI state transitions (`ghost.vulnerable = true`), ghost appearance updates, and collision logic triggering score updates and respawn behaviour. Integration tests also confirmed menu-to-game transitions, level loading, and multiplayer control separation between Pac-Man and the player-controlled ghost.

System-level testing involved structured developer-led playthroughs and informal peer-based playtesting. These sessions helped uncover complex behavioural bugs and provided direct feedback on usability, visual feedback, and gameplay flow. Issues such as movement "snagging" on corners, power-up stacking conflicts, or AI indecisiveness were diagnosed and resolved during this phase.

Debugging relied on both IDE breakpoints and targeted use of `System.out.println` to trace real-time game state changes—monitoring entity positions, active power-ups, and AI targeting behaviours. This low-level tracing proved invaluable for identifying intermittent edge cases and confirming animation sequencing.

Playtesting and feedback drove substantial improvements to the player experience. Initial feedback highlighted subtle ghost vulnerability cues, prompting the addition of a flashing warning animation before expiry. Multiplayer position-swapping felt abrupt, which led to the `swapWarningActive` state and a pre-swap countdown. Feedback also influenced the adjustment of movement tolerances (`isAligned()`), improved HUD clarity (`drawLivesDisplay`, `drawHunterModeUI`), and rebalanced power-up durations and ghost difficulty parameters (`lookAheadTiles`, Clyde's distance threshold).

The control reversal mechanic (Apple power-up) was a particular focus for usability testing. Players responded differently to inverted inputs, leading to a ghost speed reduction during that phase to maintain fairness. UI clarity improvements were implemented to make active effects immediately recognisable, including labelled timer bars and colour-coded state indicators.

Throughout development, quality assurance was integrated into the workflow. Jira tracked tasks, bugs, and feature refinements. Git version control enabled safe iteration, rollback, and branch management. Performance testing and refinement, discussed further in Section 8.3, were also conducted during these cycles to ensure frame stability and consistent input responsiveness.

In summary, `PacRun`'s testing strategy moved beyond conventional correctness validation to focus on player experience, technical robustness, and continuous gameplay refinement. This

iterative approach, rooted in both automated logic verification and user-driven playtesting, was essential to delivering a playable, engaging, and reliable final product.

8.3 Performance Evaluation and Ethical Considerations

Performance evaluation for *PacRun* was primarily qualitative, as Java Swing lacks native frame-rate counters or advanced profiling tools. The aim was to assess whether the game provided a stable and responsive experience suitable for a fast-paced arcade title, despite being built on a general-purpose GUI toolkit.

Observations during development and structured playtesting confirmed that the game's animations and input responsiveness remained consistent throughout. Movement and turning felt immediate, and power-up activations, including complex, layered states, were processed without perceptible lag. Performance was tested under higher computational load, particularly during the Bonus Round, where dense pellet collection and score calculations occur. Even under these conditions, the game remained stable with no stuttering or critical slowdowns. Informal resource monitoring revealed no excessive memory usage or CPU load, suggesting that performance-conscious decisions, such as minimising object instantiation within the `move()` and `paintComponent()` loops and streamlining rendering paths, were effective. While precise metrics were not recorded, all test systems ran the game fluidly, reinforcing that *PacRun* achieved its performance targets.

In parallel, ethical considerations were strictly followed during all playtesting activities. Before participation, players were informed of the project's scope, the voluntary nature of testing, and their right to withdraw at any time. Verbal consent was obtained, and no personally identifiable data was recorded. Feedback was anonymised and used solely to guide development and support analysis.

Sessions were kept brief and informal to reduce any pressure on participants. Observations and verbal feedback were collected in a supportive environment and logged via Jira as development tasks. The abstract, non-violent nature of *PacRun* also ensured no ethical content concerns. Although the tester pool was small due to academic constraints, all interactions followed appropriate ethical guidelines, ensuring responsible engagement throughout.

8.4 Testing Results

The testing strategy for *PacRun* yielded comprehensive validation of the game's functionality. All 27-unit tests passed successfully, as evidenced in Figure 8.1, confirming the reliability of core mechanics, including movement, collision detection, power-up systems, and ghost AI behaviour.

The console output visible in the testing results provides additional verification of critical functionality. Messages such as "Blue ghost initialized with direction L," "Pink ghost ambushing," and "Orange ghost retreating" confirm that each ghost type correctly implemented

its intended personality and targeting behaviour. Similarly, level loading statements verify that the progression system functioned as designed, with appropriate power-up distribution across levels.

Integration testing revealed important interactions between game systems. Initial conflicts between simultaneous power-up states led to the implementation of timestamp comparison logic in the `move()` method. Ghost AI transitions during power-up activations were validated through observed changes in targeting behaviour, ensuring that vulnerable ghosts correctly entered fleeing states and adjusted their pathfinding accordingly.

Playtesting provided crucial feedback on the player experience that shaped several key refinements:

- Adjustment of movement tolerances to improve corner navigation based on early control feedback
- Implementation of reduced ghost speed during control reversal periods to balance Level 3 difficulty
- Addition of timer bars and visual indicators to clarify active power-up states

Refinement of the multiplayer position-swapping mechanism based on competitive play sessions

Through this iterative testing and refinement process, PacRun evolved from a technically functional implementation to a polished game with balanced difficulty progression and clear player feedback. The passing of all unit tests in the final phase demonstrates the effectiveness of combining automated verification with human-centered evaluation to produce a cohesive, engaging final product.

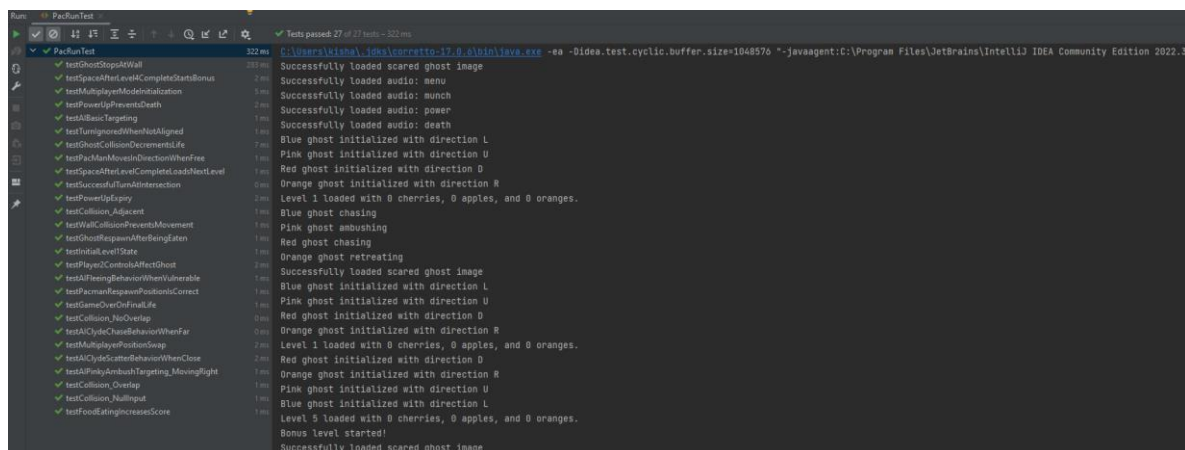


Figure 8.1: PacRun JUnit test results

9. Conclusion and Future Directions

9.1 Summary of Research Contributions and Findings

This capstone project successfully demonstrated how core object-oriented principles and real-time state management can be applied within a Java-based game architecture, using only

foundational tools like Java Swing. *PacRun* served not only as a complete reimagining of *Pac-Man*, but more critically, as a case study in constructing modular, extensible systems in a non-engine environment.

A key research contribution was validating the feasibility of implementing a robust object-oriented architecture in Java for interactive 2D games. Through encapsulated entity design and an MVC-inspired structure, the project managed layered mechanics and gameplay variations with clarity and scalability. This structure enabled the seamless integration of diverse systems such as power-up stacking, AI behaviour, and multiplayer interaction, all within a unified architecture.

The development process also yielded practical insights into Swing’s suitability for real-time game applications. While functional for graphics and input, performance constraints required conscious optimisation, such as object reuse, efficient rendering loops, and memory-conscious design. These findings contribute to the broader understanding of Swing’s limitations and potential as a prototyping platform.

Significant technical challenges, such as concurrent player input, dynamic ghost AI, and layered power-up effects, were addressed using well-structured control flow, timestamp-based state management, and modular logic. The successful implementation of these systems reinforces best practices in real-time software design and highlights the trade-offs between simplicity and control in custom-built frameworks.

Finally, the project demonstrated the role of testing in improving software reliability. The use of JUnit for unit and regression testing, combined with iterative feedback from playtesting, ensured system stability and informed continuous refinement. In this way, *PacRun* contributed both a working artefact and practical insights into academic game development using general-purpose programming tools.

9.2 Future Directions and Broader Implications

Beyond immediate technical extensions, *PacRun* presents broader opportunities for future development and meaningful contributions to game design, software engineering, and computer science education.

Enhancements such as procedural level generation, local or online leaderboard systems, and gameplay analytics could significantly improve replayability, user engagement, and balance tuning. Expanded UI features like richer menus, tutorials, and dynamic settings would further accessibility, while configurable difficulty modes and guided onboarding could ease entry for new players without compromising challenge.

More broadly, *PacRun* offers a replicable strategy for reimagining legacy games: preserving a familiar core loop while introducing modular innovations such as diverse power-ups and competitive multiplayer. This approach highlights how nostalgia and modern mechanics can be hybridised to reinterpret classic titles.

The project also affirms the strength of object-oriented design in managing interactive system complexity. Modularity, encapsulation, and state-driven logic enabled concurrent features like power-up stacking and responsive input control. Combined with agile methods and structured testing, the architecture proved scalable even without a commercial engine.

Educationally, *PacRun* illustrates Java’s continued relevance in teaching software architecture, real-time systems, and performance-aware design. The project’s development challenges provide authentic learning scenarios around managed memory, modularity, and system integration.

In summary, while *PacRun* represents a completed capstone project, its structure and findings offer a foundation for ongoing development and valuable insights into remaking classic games, applying software engineering in games, and leveraging Java for pedagogical purposes.

9.3 Comparison of Intended and Implemented Features

The project initially aimed to deliver a Java-based reimagining of *Pac-Man*, with modular architecture, three power-ups, basic AI, and a single-player mode. In practice, the implementation exceeded these goals. In addition to all planned features, *PacRun* includes a Bonus Round with combo-based scoring, a fully functional local multiplayer mode with dynamic mechanics, and refined aesthetic elements such as animated menus and enhanced HUD feedback. These additions demonstrate not only the feasibility of the original plan but also the capacity to extend beyond it within a constrained timeframe.

The *PacRun* project set out to reinterpret the 1980 arcade classic by applying contemporary software engineering methodologies to preserve its core gameplay loop while enhancing mechanical depth and architectural complexity. The initial objectives were ambitious: to construct a scalable and modular game system in Java, introduce gameplay-altering power-ups, implement personality-based AI, and deliver responsive, grid-aligned control systems. The project also sought to demonstrate effective planning, structured development, and a polished user experience without relying on a dedicated game engine.

The final implementation not only met these objectives but surpassed them. A fully modular, object-oriented codebase was realised, structured around an MVC-inspired model and supported by well-encapsulated class hierarchies. All three core power-ups, Cherry, Apple, and Orange, were implemented with distinct state transitions and gameplay consequences, reinforcing state-driven design. Each mechanic introduced new forms of player interaction and decision-making.

Beyond the original scope, several additional features were delivered. A time-limited Bonus Round introduced score multiplier mechanics and encouraged rapid pellet chaining, enhancing replay value. The aesthetic layer of the game was significantly expanded through sprite animation, themed menus, and real-time HUD effects, resulting in a visually coherent and engaging experience. Most notably, the multiplayer mode—initially outside the project's minimum scope—was successfully implemented as “Pellet Race”, allowing player-versus-player dynamics, position-swapping mechanics, and real-time score competition.

Advanced AI was integrated, with ghosts exhibiting varied pursuit strategies and context-sensitive behaviour, including transitions between scatter, frightened, and consumed states. Control responsiveness was achieved through buffered input, directional snapping, and smooth collision resolution. These systems were made resilient through the consistent application of modular logic and timestamp-based state updates.

In conclusion, *PacRun* demonstrates how a legacy arcade experience can be faithfully reimaged and expanded through disciplined architectural design, agile development, and thoughtful gameplay enhancement. It stands as a comprehensive case study in applying object-oriented principles to interactive system design and highlights Java's continued relevance in academic software engineering contexts. The project is both a technical achievement and a reflective application of computer science fundamentals to a creative, user-focused product.

9.4 Final Reflections and Critical Assessment

PacRun served as a technical culmination of my undergraduate studies, evolving from a reinterpretation of *Pac-Man* into a fully featured, Java-based game system that applied modular architecture, state-driven logic, and real-time input handling without reliance on external engines.

The project successfully delivered its research objectives, demonstrating how object-oriented design can support layered features such as power-up management, AI behaviour, and multiplayer functionality. The MVC-inspired structure and class-based encapsulation provided a maintainable foundation for expanding mechanics across multiple game modes.

Key lessons emerged from addressing complexity under evolving requirements. Multiplayer input separation, AI state transitions, and synchronised game logic required ongoing refinement. Introducing multiplayer midway significantly increased the scope but ultimately became a highlight of the final product. Persistent issues with collision detection and timing were addressed through systematic debugging and control logic adjustments.

Platform limitations became apparent. Java Swing proved effective for prototyping but lacked support for hardware acceleration, animation frameworks, and audio capabilities. Despite optimisations to maintain responsiveness, these constraints impacted immersion and limited potential for broader engagement, especially in accessibility and dynamic level generation.

From a development management perspective, tools like Git and Jira support structured progress. Milestone tracking and iterative testing guided task prioritisation, though earlier adoption of unit testing would have improved regression control during complex feature additions.

Overall, this project provided a valuable opportunity to apply theoretical concepts to a non-trivial, interactive system. It reinforced the importance of adaptable planning, systematic testing, and clean software architecture in game development, while highlighting areas for technical and professional growth.

10. References and Acknowledgments


























- [1] T. Iwatani, "Pac-Man (1980)," in *The Video Game Explosion: A History from PONG to PlayStation and Beyond*, M. J. P. Wolf, Ed. Westport, CT, USA: Greenwood Press, 2008, pp. 73-74.
- [2] K. Salen and E. Zimmerman, *Rules of Play: Game Design Fundamentals*. Cambridge, MA, USA: MIT Press, 2003.
- [3] M. Csikszentmihalyi, *Flow: The Psychology of Optimal Experience*. New York, NY, USA: Harper & Row, 1990.
- [4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA, USA: Addison-Wesley, 1994.
- [5] R. Nystrom, *Game Programming Patterns*. Geneva Benning, 2014. [Online]. Available: <https://gameprogrammingpatterns.com/>
- [6] J. Nielsen, "Enhancing the explanatory power of usability heuristics," in *Proc. SIGCHI Conf. Human Factors Comput. Syst. (CHI '94)*, Boston, MA, USA, 1994, pp. 152-158.
- [7] H. Lowood, "Videogames in Computer Space: The Complex History of Pong," *IEEE Annals Hist. Comput.*, vol. 31, no. 3, pp. 5-19, Jul.-Sep. 2009.
- [8] D. L. King, P. H. Delfabbro, and J. Billieux, "Unfair play? Video games as exploitative monetized services: An examination of game patents from a consumer protection perspective," *Comput. Human Behav.*, vol. 101, pp. 131-143, Dec. 2019.

Appendices Appendix[A] – Jira

Pac-Run											
SummaryTimelineBacklogBoardReportsListFormsGoalsCodeArchived work itemsPagesShortcuts											
Search listFilter											
	Type	Key	Summary	Status	Comments	Assignee	Due date	Labels	Created	Updated	Reporter
		KAN-6	2D enemy used for the Maze	Done	Add comment				Oct 20, 2024	Oct 20, 2024	Kishan Ravikumar
		KAN-7	Fixes to the Maze so Pac can move and room more freely	Done	Add comment				Oct 20, 2024	Oct 20, 2024	Kishan Ravikumar
		KAN-8	Creating walls in pacman and ensuring it works accordingly	Done	1 comment				Oct 20, 2024	Oct 20, 2024	Kishan Ravikumar
		KAN-2	Challenge Week document - Includes documentation of...	Done	Add comment				Oct 11, 2024	Oct 11, 2024	Kishan Ravikumar
		KAN-3	Simple Risk assessment document for challenge week 4...	Done	Add comment				Oct 11, 2024	Oct 11, 2024	Kishan Ravikumar
		KAN-4	Code for challenge of pac being able to move up/down...	Done	Add comment				Oct 11, 2024	Oct 11, 2024	Kishan Ravikumar
		KAN-9	Wall Maze has been completed and working functionality	Done	Add comment				Nov 6, 2024	Nov 6, 2024	Kishan Ravikumar
		KAN-10	Pacman food has been evidently completed	Done	Add comment				Nov 6, 2024	Nov 6, 2024	Kishan Ravikumar
		KAN-11	Main class to run the game has been updated	Done	Add comment				Nov 6, 2024	Nov 6, 2024	Kishan Ravikumar
		KAN-13	Game Window Setup Configure window size, layout, on...	Done	Add comment				Nov 6, 2024	Nov 6, 2024	Kishan Ravikumar
		KAN-14	Initialize Game Components Load images and assets fo...	Done	Add comment				Nov 6, 2024	Nov 6, 2024	Kishan Ravikumar
		KAN-17	UI and Visuals	Done	Add comment		Nov 8, 2024		Nov 6, 2024	Dec 3, 2024	Kishan Ravikumar
		KAN-18	Study Event Handling in Java	Done	Add comment				Dec 3, 2024	Dec 3, 2024	Kishan Ravikumar
		KAN-16	Evaluate existing Java Maze Libraries	Done	Add comment				Dec 3, 2024	Dec 3, 2024	Kishan Ravikumar
		KAN-17	Research Speed Boost Implementation	Done	Add comment				Dec 3, 2024	Dec 3, 2024	Kishan Ravikumar
		KAN-15	Research Maze Generation Algorithms(background res...	Done	Add comment				Dec 3, 2024	Dec 3, 2024	Kishan Ravikumar
		KAN-21	Debug Ghost Interaction	Done	Add comment				Dec 3, 2024	Dec 9, 2024	Kishan Ravikumar
		KAN-19	Debug Maze Wall Collisions	Done	Add comment	Kishan Ravikumar	Dec 3, 2024		Dec 3, 2024	Dec 9, 2024	Kishan Ravikumar
		KAN-20	Debug Maze Rendering Issues	Done	Add comment	Kishan Ravikumar	Nov 24, 2024		Dec 3, 2024	Dec 9, 2024	Kishan Ravikumar
		KAN-23	Maze collision fixes	Done	Add comment				Feb 17, 2025	Feb 17, 2025	Kishan Ravikumar
		KAN-22	Introduction for Report - Comments	Done	Add comment				Feb 17, 2025	Feb 22, 2025	Kishan Ravikumar
		KAN-24	Code debugging	Done	Add comment				Feb 17, 2025	Feb 22, 2025	Kishan Ravikumar
		KAN-26	Created a menu page	Done	Add comment				Feb 22, 2025	Feb 22, 2025	Kishan Ravikumar
		KAN-29	Create a new power-up item (Cherry) in the map	Done	Add comment				Mar 5, 2025	Mar 8, 2025	Kishan Ravikumar

Pac-Run											
SummaryTimelineBacklogBoardReportsListFormsGoalsCodeArchived work itemsPagesShortcuts											
Search listFilter											
	Type	Key	Summary	Status	Comments	Assignee	Due date	Labels	Created	Updated	Reporter
		KAN-30	Apply a temporary effect (e.g., Speed Boost for 10s)	Done	Add comment				Mar 5, 2025	Mar 6, 2025	Kishan Ravikumar
		KAN-32	Use a timer to reset the effect after 10 seconds.	Done	Add comment				Mar 5, 2025	Mar 6, 2025	Kishan Ravikumar
		KAN-34	Detect When Pac-Man Eats the Cherry	Done	Add comment				Mar 5, 2025	Mar 6, 2025	Kishan Ravikumar
		KAN-35	Apply the Power-Up Effect (Speed Boost)	Done	Add comment				Mar 5, 2025	Mar 6, 2025	Kishan Ravikumar
		KAN-23	Super power speed boost for pacman	Done	Add comment				Feb 17, 2025	Mar 6, 2025	Kishan Ravikumar
		KAN-27	Change Game to Full Screen size accordingly	Done	Add comment	Kishan Ravikumar			Feb 22, 2025	Feb 22, 2025	Kishan Ravikumar
		KAN-28	Adding a super power - cherry item when consumed.	Done	Add comment				Mar 5, 2025	Mar 6, 2025	Kishan Ravikumar
		KAN-36	Improve Ghost Movement Patterns	Done	Add comment				Mar 6, 2025	Mar 6, 2025	Kishan Ravikumar
		KAN-37	Implement Cherry Power-ups	Done	Add comment				Mar 6, 2025	Mar 6, 2025	Kishan Ravikumar
		KAN-38	Create Level Select Functionality	Done	Add comment				Mar 6, 2025	Mar 6, 2025	Kishan Ravikumar
		KAN-39	Apply Pac-Man Font Throughout Interface	Done	Add comment				Mar 6, 2025	Mar 6, 2025	Kishan Ravikumar
		KAN-40	Improve HUD layout with clear separation between ele...	Done	Add comment				Mar 6, 2025	Mar 6, 2025	Kishan Ravikumar
		KAN-41	Implement dark blue gradient background instead of pl...	Done	Add comment				Mar 6, 2025	Mar 6, 2025	Kishan Ravikumar
		KAN-42	Fix title positioning to prevent overlap with game info	Done	Add comment				Mar 6, 2025	Mar 6, 2025	Kishan Ravikumar
		KAN-43	Add smooth fade-out death animation when losing a life	Done	Add comment				Mar 6, 2025	Mar 6, 2025	Kishan Ravikumar
		KAN-44	Replace numeric lives counter with visual Pac-Man icons	Done	Add comment				Mar 6, 2025	Mar 6, 2025	Kishan Ravikumar
		KAN-45	Add Pac-Man style lives icons and death animation	Done	Add comment				Mar 6, 2025	Apr 2, 2025	Kishan Ravikumar
		KAN-46	Enhance game UI and visual effects	Done	Add comment				Mar 6, 2025	Apr 20, 2025	Kishan Ravikumar
		KAN-47	Add Level 3 with reversed controls apple power-up	Done	Add comment				Mar 6, 2025	Apr 2, 2025	Kishan Ravikumar
		KAN-48	UI Improvements: Fix title/level text overlap	Done	Add comment				Mar 6, 2025	Apr 2, 2025	Kishan Ravikumar
		KAN-49	Moved title banner higher and made it larger to create p...	Done	Add comment				Mar 6, 2025	Mar 6, 2025	Kishan Ravikumar
		KAN-50	Add animated background effects to MenuScreen	Done	Add comment				Mar 9, 2025	Mar 9, 2025	Kishan Ravikumar
		KAN-51	Enhance menu UI with custom styled buttons	Done	Add comment				Mar 9, 2025	Mar 9, 2025	Kishan Ravikumar
		KAN-52	Implement level selection dialog	Done	Add comment				Mar 9, 2025	Mar 9, 2025	Kishan Ravikumar

Appendix [B] – Gitlab Activity Log

<div>Mar 27, 2025</div> <div><div> Implement bonus level with scoring and combo system : ...</div><div>Ravikumar, Kishan authored 4 weeks ago</div></div> <div><div> Implement intelligent ghost movement with grid alignment : ...</div><div>Ravikumar, Kishan authored 4 weeks ago</div></div> <div><div> Fix: Implement responsive turning system for Pac-Man movement ...</div><div>Ravikumar, Kishan authored 4 weeks ago</div></div>	<div>Apr 23, 2025</div> <div><div> Add Junit and GUI test coverage for PacRun game ...</div><div>Ravikumar, Kishan authored 1 day ago</div></div>
<div>Mar 15, 2025</div> <div><div> Add Level 4 with ghost-hunting mechanics ...</div><div>Ravikumar, Kishan authored 1 month ago</div></div>	<div>Apr 22, 2025</div> <div><div> Fixed audio resource loading paths in AudioPlayer class ...</div><div>Ravikumar, Kishan authored 2 days ago</div></div>
<div>Mar 09, 2025</div> <div><div> Add animated background effects to MenuScreen ...</div><div>Ravikumar, Kishan authored 1 month ago</div></div>	<div>Apr 21, 2025</div> <div><div> Add ghost player lives system in multiplayer mode ...</div><div>Ravikumar, Kishan authored 3 days ago</div></div>
<div>Mar 08, 2025</div> <div><div> Improve game UI: Reposition PAC-RUN title to avoid overlap ...</div><div>Ravikumar, Kishan authored 1 month ago</div></div> <div><div> Improve UI: Add Pac-Man style lives icons and death animation</div><div>Ravikumar, Kishan authored 1 month ago</div></div> <div><div> Adding a super power - cherry item when consumed. ...</div><div>Ravikumar, Kishan authored 1 month ago</div></div>	<div>multiple bonus level gameplay and UI issues ...</div> <div>Ravikumar, Kishan authored 4 days ago</div> <div>Apr 20, 2025</div> <div><div> Enhance game navigation with intuitive shortcuts and menu return functionality ...</div><div>Ravikumar, Kishan authored 4 days ago</div></div> <div><div> Fix multiplayer collision logic for improved gameplay experience ...</div><div>Ravikumar, Kishan authored 5 days ago</div></div>
<div>Feb 24, 2025</div> <div><div> hardcoded pixels and modified screen size</div><div>Ravikumar, Kishan authored 1 month ago</div></div>	<div>Apr 10, 2025</div> <div><div> Replace programmatically drawn logo with image : ...</div><div>Ravikumar, Kishan authored 2 weeks ago</div></div>
<div>Feb 22, 2025</div> <div><div> Add menu screen and update PacRun with fullscreen implementation</div><div>Ravikumar, Kishan authored 2 months ago</div></div> <div><div> Updated PacRun game with fullscreen implementation</div><div>Ravikumar, Kishan authored 2 months ago</div></div>	<div>Apr 06, 2025</div> <div><div> Enhance menu screen with animated ghost characters : ...</div><div>Ravikumar, Kishan authored 2 weeks ago</div></div> <div><div> title bar</div><div>Ravikumar, Kishan authored 2 weeks ago</div></div> <div><div> Add improved How to Play dialog : ...</div><div>Ravikumar, Kishan authored 2 weeks ago</div></div>
<div>Nov 09, 2024</div> <div><div> Added new files for Pac-Run project</div><div>Ravikumar, Kishan authored 5 months ago</div></div> <div><div> Merge branch 'master' of cseegit.assex.ac.uk:24-25-ce30V24-25_CE30L_ravikumar_kishan</div><div>Ravikumar, Kishan authored 5 months ago</div></div> <div><div> initial commit</div><div>Ravikumar, Kishan authored 5 months ago</div></div>	<div>Apr 02, 2025</div> <div><div> Enhance menu UI with centered components and animated buttons : ...</div><div>Ravikumar, Kishan authored 3 weeks ago</div></div> <div><div> Fix key shortcut controls in multiplayer mode : ...</div><div>Ravikumar, Kishan authored 3 weeks ago</div></div>

Fix multiplayer collision logic for improved gameplay experience

- Fixed collision detection between Pac-Man and player-controlled ghost
 - Implemented collision handling between player ghost and AI ghosts
 - Pac-Man now passes through player ghost without dying
 - Player ghost now correctly dies when colliding with AI ghost
 - Added appropriate visual effects and score adjustments for collisions
 - Fixed edge cases with eaten ghost respawning logic
- Improve player ghost spawn position in multiplayer mode
- Modified level map for multiplayer to separate player ghosts
 - Relocated player-controlled ghost to top section of maze
 - Positioned AI ghost centrally to create fair gameplay
 - Prevented immediate collision between players at game start
 - Created dedicated multiplayer map variant for better player experience
- Enhance AI ghost targeting for balanced multiplayer gameplay
- Implemented nearest-player targeting algorithm for pink ghost
 - Added intelligence to ghost movement to pursue closest player
 - Created dynamic threat presence that affects both players
 - Maintained original ghost behavior in single-player mode
 - Improved logging for AI decision making and movement patterns
 - Ensured balanced challenge for both players in multiplayer mode
- Balance scoring system in multiplayer mode
- Verified equal point rewards (10 pts) for pellet collection by both players
 - Maintained strategic depth with bonus/penalty scoring mechanics
 - Added appropriate score penalties when player ghost is eaten
 - Ensured score values properly display in multiplayer UI
 - Fixed score reset when starting new multiplayer rounds