Michael Crevier
CS 499 Capstone
Milestone Two: Software Design & Engineering Enhancement

Artifact Description

The artifact I selected for enhancement is a Python-based AI maze-solving game originally developed in CS-370: Current and Emerging Trends in Computer Science (Oct–Dec 2024). In its original form, the project implemented a Q-learning reinforcement agent that navigated a 2D maze toward a goal. For this enhancement, I have transformed the codebase into a new project: a turn-based AI word strategy game where a human player competes against an AI opponent attempting to predict their next word.

The enhancement involved significant architectural restructuring to create a modular game engine featuring a robust game loop, comprehensive state management, input validation, word verification, progressive scoring mechanics, and letter pool generation—all operating through a command-line interface. Additional engineering improvements include centralized logging infrastructure, YAML-based configuration, comprehensive unit testing, and an object-oriented structure designed for future expandability. This rework was completed in Spring 2025 as part of the CS 499 capstone course and establishes the foundation for my planned future enhancements involving advanced AI decision-making algorithms and database persistence.

Justification for Inclusion

I selected this artifact because it demonstrates my ability to evolve a specialized, single-purpose AI system into a scalable and extendable application. Transforming the original maze-solving project into a strategic word game required thoughtful application of software design principles and showcases my engineering capabilities.

The following components highlight my software development skills:

- Modular Architecture:

  I restructured the monolithic original code into distinct components (GameState, GameLoop, InputHandler) with clear separation of concerns. This allows for isolated testing and maintenance while preparing the codebase for layered enhancements.

- Object-Oriented Design:

  The refactor leverages encapsulation and class hierarchies to maintain clarity and modularity. Each class exposes only the functionality needed by other components, enabling easier scaling.

- Progressive Scoring System:

  I designed a dynamic scoring algorithm that rewards novelty and applies fatigue penalties to repeated words. This mechanic required careful state tracking across rounds and demonstrates thoughtful algorithm design.

- Engineering Best Practices:

  The enhanced system follows professional development workflows: Git-based version control using feature branches, centralized logging, YAML-based configuration, and unit testing for all core logic.

This enhancement directly supports the program outcome to “use well-founded and innovative techniques, skills, and tools in computing practices for the purpose of implementing computer solutions that deliver value and accomplish industry-specific goals.”

Outcome Alignment

This enhancement aligns with the Software Design and Engineering outcome of the Computer Science program. The refactored architecture serves as the foundation for two additional enhancements currently in progress:

1. Enhancement Two: Integrating intelligent AI prediction strategies using Markov chains, Monte Carlo Tree Search, Naive Bayes classification, and Q-learning to simulate adaptive opponent behavior.
2. Enhancement Three: Implementing a SQLite database backend to track gameplay history, persist AI learning, and store performance metrics across sessions.

My outcome-coverage plans remain aligned with what I outlined in Module One. This enhancement has confirmed the architectural decisions needed to support advanced AI logic and long-term data persistence, ensuring a smooth development trajectory for Enhancements Two and Three.

Reflection on Enhancement Process

Enhancing this artifact was both a technical and architectural challenge. I needed to move from a rigid, maze-solving AI into a dynamic and extensible turn-based word game engine. This shift required balancing short-term goals with long-term flexibility.

Designing the GameState class was particularly challenging—it needed to track player state, manage the shared and private letter pools, handle scoring logic with diminishing returns, and expose interfaces for future AI and database integration. Through testing and iteration, I arrived at a design that met current functionality while remaining open for expansion.

This process reinforced the importance of building for evolution. Establishing robust class boundaries, maintaining a clean project structure, and following best practices like centralized logging and automated testing ensured that future changes could be implemented with minimal rework. From a workflow perspective, using feature-based Git branching made the development process clean and traceable. Each unit test gave me confidence to improve functionality without introducing regressions, and writing modular code made testing and debugging much faster. Most importantly, this enhancement helped solidify my identity as a software engineer—one who can not only build solutions, but also architect them with foresight, clarity, and adaptability.