Randy Ortiz

Professor Kraya

CS-499 Computer Science Capstone 2025

July 25, 2025

The artifact selected for this category is the Farkle dice game I developed in C++ for IT-312: Software Development. The program is a multiplayer game that follows the standard rules of Farkle. Upon launch, the user is presented with a menu that allows them to start the game, view the game rules (which are loaded using file I/O from a .txt file), or quit. Once the game begins, the program prompts the user to enter the number of players (minimum of two) and to input the names of each player. The gameplay then proceeds in turns, beginning with Player 1. Each player must earn and bank at least 1,000 points in a single turn to officially enter the game. This requirement is enforced using scoring logic that handles both singles and triplets, based on Farkle’s rules. The game also includes logic for rolling six dice, tracking scores per player, enforcing Farkle penalties (when no scoring dice are rolled), and managing “hot dice” rerolls when all six dice result in points. The core logic was originally implemented procedurally within the Game class, with scoring handled in a single function.

The enhancement I made to this artifact was to improve the clarity and maintainability of the game’s scoring logic. Originally, the calculateScore() function handled all scoring in a single block, which made it difficult to read, debug, or scale. I refactored it into two more focused functions: scoreTriplets() and scoreSingles(), which separately handle triplets and individual 1s and 5s. I also fixed an input issue that occurred after triggering hot dice, where the user had to press Enter on a blank line before the "select scoring dice" prompt would appear. To resolve this, I created a clearInputBuffer() function that clears any leftover newline characters from the input stream. I called this function after each cin >> input to prevent skipped prompts and ensure that the “select scoring dice” message appears immediately.

These enhancements display my ability to improve algorithmic design by refactoring logic into smaller components and applying data structures. These enhancements align with course outcomes 3 and 4. For course outcome 3, the redesign of the calculateScore() function into scoreTriplets() and scoreSingles() shows my ability to design clear and efficient solutions using algorithmic thinking and control structures. This improves the readability, maintainability, and scalability of the game logic without compromising performance. For course outcome 4, breaking the scoring process into reusable components shows how I can structure a solution to solve a specific computational problem using efficient logic and clean separation of concerns.

Enhancing this artifact taught me how important modular design is when working with complex logic. By breaking down the scoring process into smaller functions I learned how to isolate individual responsibilities in code, which can make debugging and updating much easier. It also helped me better understand how to structure a program for long-term maintainability. One of the main challenges I faced was fixing the input prompt issue that happened after a hot dice reroll. The prompt to select scoring dice wouldn't show up until after the user enters an input on an empty line. This can create confusion during gameplay. After some testing, I realized it was due to leftover newline characters in the input stream. Solving this required learning how to manage input properly in C++ by implementing a custom clearInputBuffer() function. This experience helped me better understand how small issues in input/output handling can affect user experience and showed me how to apply problem-solving skills to cleanly fix them.