ÇANKAYA UNIVERSITY

SOFTWARE ENGINEERING DEPARTMENT

SOFTWARE PROJECT I

Name Surname	**Dilara Çağla SARISU**
Identity Number	**202128201**
Course	**SENG-271**
Experiment	**Experiment 1**
E-mail	**c2128201@student.cankaya.edu.tr**

REQUIREMENT ANALYSIS

Expectations from the Program:

The program is expected to simulate a battle between two groups of soldiers using stack data structures, with inputs provided through an "input.txt" file rather than interactive commands. The program should:
- Process input data for the health and strength of each soldier from the file.

- Populate the two soldier stacks according to the provided specifications in the file.

- Conduct combat sequences based on the rules defined, including handling special conditions like critical hits.

- Introduce random reinforcements based on intervals specified in the input file.

- Produce an output reflecting the outcome of battles, the status of individual soldiers, and a summary at the end to declare the winning side.

How does it work?

The program functions by:

- Reading and parsing the "input.txt" file to acquire the necessary game setup and instructions.

- Performing stack operations to manage the soldiers as per the instructions in the input file.

- Simulating the battles using the soldiers' health and strength attributes to determine the outcomes of each encounter.

- Implementing special rules such as critical hits and reinforcements according to the logic specified in the file.

- Progressing through the battle sequence until one side is depleted of soldiers, then terminating the game and outputting the results.

Functional Requirements:

- File I/O: Capability to read from and write to files.

- Stack Operations: The ability to execute essential stack operations: push, pop, and top.

- Soldier Representation: Representation of soldiers with attributes such as health and strength.

- Battle Simulation: An algorithm to simulate combat based on soldier attributes.

- Special Rules Implementation: Functions to handle special rules like reinforcements and critical hits.

- Game Summary: Generate an end-of-game summary indicating the winner.

- Non-Interactive Gameplay: A system designed to operate based on file input without real-time user interaction.

Software Usage:

The software functions as a simulation tool that reads predefined battle scenarios from an "input.txt" file, processes the data, and simulates a battle between two groups of soldiers using stack data structures. Provides an opportunity to observe how different input conditions can affect the outcome of a simulated event, allowing for experimentation with variables such as soldier strength, health, and the frequency of special events like reinforcements.

Overview:

Initialization: Sets up data structures for the simulation.

File Reading: Reads and parses soldier data and rules from "input.txt".

Stack Population: Allocates soldiers to their respective sides' stacks.

Battle Simulation: Automatically calculates the results of combat rounds.

Reinforcement Logic: Integrates additional random soldiers if dictated by the input.

Result Processing: Determines and updates the status after each round.

Output Generation: Summarizes the battle outcome in an output file.

Users craft the battle scenario via the input file and review the simulation results post-execution, making it a passive but informative and entertaining experience.

Error Messages:

- The program must provide precise error messages for any issues encountered during file processing, such as format errors, missing data, or logical inconsistencies in the input.

Requirements:

- File Parsing Logic: To correctly interpret the contents of "input.txt" and translate them into game logic.

- Dynamic Stack Management: To allow stacks to grow and shrink dynamically as soldiers are added and removed.

- Combat Logic: A system to simulate battles and determine outcomes based on soldier attributes.

- Reinforcement Logic: To simulate reinforcements as indicated by the input file.

- Game State Management: To keep track of the game's progression through various states.

- End-Game Criteria: To assess when one side is out of soldiers and to present the final outcome.

DESIGN: Programmer's Perspective

How will the program perform the desired job and produce the output?

Problem:

The program is designed to simulate a battle scenario between two opposing forces, where soldiers with unique attributes engage in combat. The simulation continues until one side is completely defeated.

Solution:

The program will process the battle simulation by reading the initial setup and commands from an input.txt file instead of interactive user input. The simulation will follow the battle logic by managing the soldiers through stack data structures for each side and process actions as described in the file.

Main Data Variable:

Soldier Stack: A class or a struct that represents a soldier with attributes such as health, strength, and critical chance. Each side in the battle will have a stack consisting of Soldier instances.

Algorithm:

Initialization: Read from input.txt and initialize stacks with soldier data.

Battle Process:

- Push Operation: Add soldiers to their respective stacks as specified in the file.
- Top Operation: Simulate a fight between the top soldiers of each stack.
- Pop Operation: Remove the defeated soldier from the stack.
- Random Reinforcement: If specified, add new soldiers to the stacks at defined intervals.
- Critical Hit Calculation: Determine the outcome of fights with possible critical hits.
- Turn-based Execution: Process each action in the sequence defined by the file.
- Conclusion: When a stack is empty, the simulation ends, and the program declares the winner.

Special Design Properties:

Modularity: The program is organized into separate modules for better manageability.

Robustness: Includes error-checking for file input to ensure correct simulation processing.

Execution Flow Between Subprograms:

File Reader: Reads input.txt and parses commands and data to initialize the simulation.

Game Logic Controller: Manages the core battle logic, processes the parsed input, and updates the game state.

Stack Manager: Handles stack operations, adding and removing Soldier instances as the battle progresses.

Combat Simulator: Executes combat logic, including the assessment of critical hits and determining battle outcomes.

Reinforcement Scheduler: Triggered by conditions in the input file to add new soldiers to the battle at appropriate times.

Output Formatter: Assembles the simulation results and writes them to an output file or displays them on the screen.

End Game Analyzer: Evaluates the condition for the end of the simulation and processes the final results.

The File Reader must accurately translate file contents into commands for the Game Logic Controller. The Stack Manager must maintain the integrity of the soldier stacks based on these commands. This design ensures that each component of the program can be tested and debuged independently, facilitating maintenance and scalability.

TESTING User aspect

“How can the program break, produce wrong output, work incorrectly?”

Bugs and Software Reliability:

Parsing Errors: Misinterpretation of the input.txt file can lead to incorrect setup or progress of the battle simulation.

Data Validation: If the program doesn't validate the data from the file correctly, it may crash or behave unexpectedly with invalid or malformed input.

Logic Flaws: There might be inconsistencies in how the battle logic is applied, such as incorrect damage calculations or stack manipulations.

Edge Cases: Scenarios like a perfectly balanced battle where neither side wins or runs out of soldiers could lead to infinite loops or deadlocks.

Output Generation: The program may produce incorrect or incomplete results files, especially when handling unusual or unexpected battle outcomes.

Software Extendibility and Upgradeability:

Hard-Coded Logic: If the simulation parameters are not dynamic, it will be hard to modify or extend the battle scenarios without changing the code.

Code Maintainability: Without a clear structure or modularity, adding new features like different soldier types or battle conditions can become complex and error-prone.

Data Handling: The way the program reads and processes the input file must be adaptable to changes in the file format or the data it contains.

Performance Considerations:

Efficiency: The simulation should run efficiently, without unnecessary computation, to provide quick results, even for large-scale battles.

Resource Utilization: It should not consume excessive memory or CPU cycles, which could affect the performance on less powerful machines.

Scalability: The program should be able to handle a large number of soldiers and a high frequency of battles without a degradation in performance.

Comments:

The assignment resulted in a functional stack-based battle simulation program with modular design, critical hit mechanics, and a system for adding reinforcements. While the use of stacks and a modular design was effective and should be retained for future projects, improvements could be made in input validation, performance optimization, and enhancing user experience through better output interpretation. Additionally, incorporating automated testing could enhance software reliability. For future iterations, considering an interactive element and earlier performance profiling could further improve the project.