# 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** |

# PROBLEM STATEMENT

### The Goals of The Programming Assignment:

- Representing two different groups of soldiers using the Stack data structure.
- Understanding and applying the basic stack operations push, pop and top.
- Simulating soldiers fighting in a sequential manner.
- Defining the health and strength characteristics of each soldier.
- Determining the battle algorithm according to the strength characteristics of soldiers.
- To determine which side wins at the end of the game.
- Allowing the input file to dynamically add soldiers.
- Handling special situations such as critical shots.
- Making the game more dynamic by creating random reinforcement soldiers.

### What are the normal inputs to the program?

In the standard input file **(input.txt);**

- Health and strength values of each soldier.
- Which side will add troops.
- Fight command.
- Critical shot command.
- Information on which side the soldiers will be added to.

### What output should the program create?

| | **Input** | **Output** |
|---|---|---|
| Add Soldier | A 1 22,987;75,647 | Add soldiers to side 1<br>S- H:22 S:987<br>S- H:75 S:647 |
| Fight | F | 2 hit 34 damage |
| Call Reinforcement | R 1 | Called reinforcements to side 1<br>S- H:100 S:59 |
| Critical Shot | C | Critical shot<br>1 has a casualty |
| Turn Info | (Based on game sequence) | Turn: Side 1 **or**<br>Turn: Side 2 |
| Soldier Death | (Based on game sequence) | Soldier from side 1 has fallen **or**<br>Soldier from side 2 has fallen |
| End Game Summary | (End of Game) | Side 1 wins with 3 soldiers remaining **or** Side 2 wins with 2 soldiers remaining |

**What error handling was required?**

If this was a user-input based game, there could be error handling such as

- Trying to draw soldiers from an empty stack.
- Invalid command input.
- Adding excessive numbers of soldiers.
- Invalid soldier values (e.g. negative strength or health).
- How the war will end when there are no soldiers left in either stack.
- No value should be created outside the specified limits when creating random soldiers.

However, since this program takes its input from a file, it simply displays an error message and restarts.

## DESIGN

**The Design Decisions:**

- Encapsulating the characteristics of soldiers within a class.
- Creating a master class that will manage the general logic and flow of the game.
- Defining functions such as adding troops, war, reinforcements and critical shots as methods.
- Creating classes and methods to validate and interpret commands received from the input file.
- Managing all situations in the game with a state machine.
- Having separate a functions for special moves like critical shots.
- Using mathematical functions to optimize military-to-military damage calculations.

**What data structures are used:**

- Stack(Using arrays, pointer, struct)
- Lists(Using arrays, pointer, struct)
- Classes(Using structures)
- Basic data types (int, char)
- Enums (for rank information, troop status, etc.)
- Two-dimensional arrays (for storing soldier properties)
- Queues (to direct soldiers moving in turn)(Using arrays, pointer)
- Arrays (for input/output operations)
- Linked lists (for dynamic data storage) (Using arrays, pointer, struct)

**What algorithms are used:**

- Damage calculation

  **Damage = (Strength1-Strength2) x 0.05 + 50**
- Stack addition and removal functions.
- Summoning random reinforcements.
- Order switching algorithm.
- Critical shot mechanics.
- Algorithm for randomizing soldier attributes.

  **rand() function**
- Comparing the soldiers of the two sides.
- Selecting new soldiers after the death of a soldier.
- Algorithm for determining the winner at the end of the game.

**Pros/Cons of choices above:**

**Pros:**

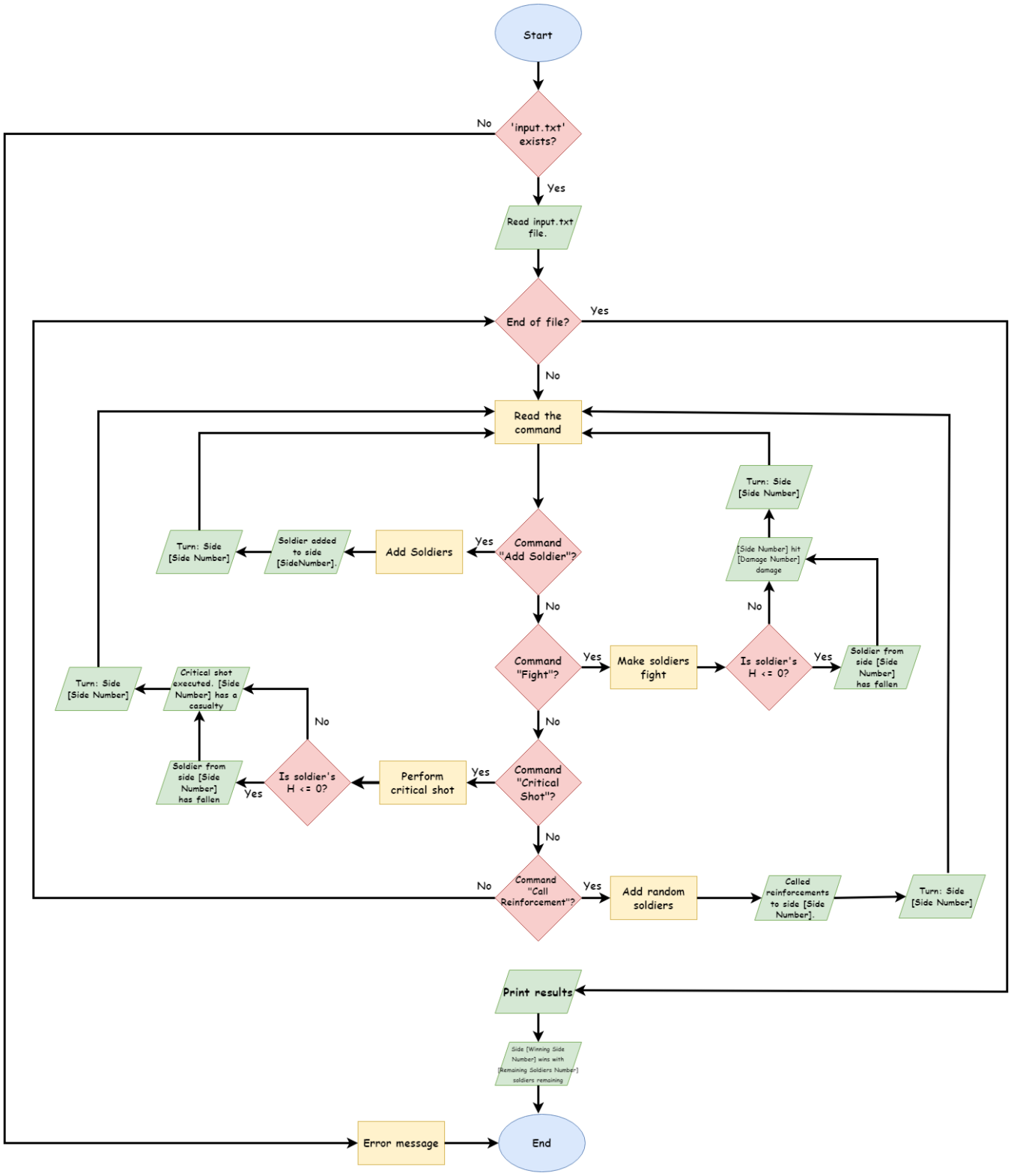
- The stack structure naturally ensures that the last soldier added will be the first to fight.
- Using classes keeps the code modular and readable.
- Separation of algorithms allows each function to be tested independently.
- Random distribution provides a different experience every time the game is played.

**Cons:**

- The stack structure is not ideal for accessing a specific rank of soldiers.
- The complexity of some algorithms can lead to performance issues.
- Random attribute distribution can sometimes lead to unbalanced games.

## IMPLEMENTATION DETAILS

### Describe your implementation process. (Flowchart)



**What sample code did you start with?**

Initially, I started with a simple code based on a structure that can hold the properties of soldiers and a stack structure that can store these soldiers.

```
typedef struct soldier {
    int health;
    int strength;
} Soldier;

typedef struct stack {
    Soldier soldiers[100];
    int top;
} Stack;
```

**How did you extend or adapt this code?**

Based on this basic structure, I added functions to add soldiers to the stack, remove them from the stack, and retrieve the top soldier. I also added a number of functions and game mechanics that control how the game progresses by reading commands from the file.

```
int isEmpty(Stack* s);

void push(Stack *s, Soldier soldier) {
    if(s->top < 99) {
        s->top++;
        s->soldiers[s->top] = soldier;
    } else {
        printf("Stack is full, can't add soldier.\n");
    }
}


Soldier pop(Stack *s) {
    if(!isEmpty(s)) {
        return s->soldiers[s->top--];
    } else {
        printf("Stack is empty, can't pop soldier.\n");
        Soldier emptySoldier = {-1, -1};
        return emptySoldier;
    }
}

Soldier top(Stack *s) {
    return s->soldiers[s->top];
}



int isEmpty(Stack* s) {
    return s->top == -1;
}
```

For game mechanics, I added functions such as damage calculation, random soldier creation and critical shot.

```c
int damageCalculation(int strength1, int strength2) {
    return (strength1 - strength2) * 0.05 + 50;
}

Soldier randomSoldier() {
    Soldier s;
    s.health = (rand() % 100) + 1;
    s.strength = (rand() % 100) + 1;
    return s;
}

void criticalShot(Stack *s1, Stack *s2, int *turn) {
    if (*turn == 1) {
        printf("Critical shot by Side 1\nSide 2 is completely wiped out!\n");
        while (!isEmpty(s2)) {
            pop(s2);
        }
        *turn = 2;
    } else {
        printf("Critical shot by Side 2\nSide 1 is completely wiped out!\n");
        while (!isEmpty(s1)) {
            pop(s1);
        }
        *turn = 1;
    }
}
```

**What was your development timeline?**

For three weeks I focused intensely on implementing this code.
In the first week, I analyzed the project requirements and strategized the overall approach.
In the second week, I drafted the design by laying out the main structure of the code.
In the third week, I made significant progress:

- Created flowcharts.
- Basic structures defined: Soldier and Stack.
- Integrated critical stack operations: push, pop and top.
- Game mechanics have been unified through functions such as damage calculations and random troop selection.
- Implemented main gameplay loop involving file-based command reading.
- Multiple test cases were executed to validate the game's logic.
- Identified and rectified errors and glitches that surfaced during the tests.
- Complete with gameSummary function to determine and display game results.

**TESTING NOTES**

**Describe how you tested your program.**

Here are some specific test cases designed to assess different situations

**Basic Fight Mechanism:**

input.txt:
**A 1 80,40**
**A 2 70,50**
**F**

Expected output:

```
Add soldiers to side 1
S- H:80 S:40
Add soldiers to side 2
S- H:70 S:50
Side 1 hit Side 2 with 49 damage
End Game Summary
Side 2 wins with 1 soldiers remaining
```

**Overflow of Stack:**
Trying to push soldiers into a stack that's already full.

input.txt:
**A 1 90,90**
**(repeated 100 times)**
**A 1 90,90**

Expected output:

```
Add soldiers to side 1
S- H:90 S:90
Add soldiers to side 1
S- H:90 S:90
Add soldiers to side 1
S- H:90 S:90
Add soldiers to side 1
S- H:90 S:90
Add soldiers to side 1
S- H:90 S:90
Stack is full, can't add soldier.
```

**Underflow of Stack:**
Initiating a fight when one of the stacks is empty.

input.txt:
**A 1 80,40**
**F**

Expected output:

```
Add soldiers to side 1
S- H:80 S:40
One of the sides has no soldiers left. Can't fight.
End Game Summary
```

**Reinforcements:**
Testing the randomness of the reinforcement mechanism.

input.txt:
**R 1**
**R 2**

Expected output: (This might vary due to the randomness)

```
Called reinforcements to side 1
S- H:71 S:55
Called reinforcements to side 2
S- H:40 S:37
End Game Summary
```

**Critical Shot Scenario:**
Testing the critical shot which wipes out one side.

input.txt:
**A 1 80,40**
**A 2 70,50**
**C**

Expected output:

```
Add soldiers to side 1
S- H:80 S:40
Add soldiers to side 2
S- H:70 S:50
Critical shot by Side 1
Side 2 is completely wiped out!
End Game Summary
Side 1 wins with 1 soldiers remaining
```

**Edge Health and Strength:**
Soldiers with the lowest and highest possible health and strength values.

input.txt:
**A 1 1,100**
**A 2 100,1**
**F**

Expected output:

```
Add soldiers to side 1
S- H:1 S:100
Add soldiers to side 2
S- H:100 S:1
Side 1 hit Side 2 with 54 damage
End Game Summary
```

**Mixed Commands:**
A combination of different commands to test integration.

input.txt:
**A 1 80,40**
**R 2**
**F**
**A 2 60,60**
**F**

Expected output:

```
Add soldiers to side 1
S- H:80 S:40
Called reinforcements to side 2
S- H:98 S:58
Side 1 hit Side 2 with 49 damage
Add soldiers to side 2
S- H:60 S:60
Side 2 hit Side 1 with 51 damage
End Game Summary
Side 2 wins with 2 soldiers remaining
```

**Unrecognized Commands:**
Testing the program's handling of unexpected inputs.

input.txt:
**A 1 80,40**
**X 2 70,50**

Expected output:

```
Add soldiers to side 1
S- H:80 S:40
Unrecognized command: X
Unrecognized command: 2
Unrecognized command: 70,50
End Game Summary
```

## What were the normal inputs you used?

**A**: To add a soldier to a specific side. It expects the side (1 or 2) and then the health and strength of the soldier. For example: A 1 100,50 would add a soldier with 100 health and 50 strength to side 1.
**F**: To make the soldiers fight. It doesn't need additional data after the command.
**R**: To call a random reinforcement soldier to a side. It expects the side (1 or 2) after the command. For example: R 1 would call a random reinforcement soldier to side 1.
**C**: To wipe out all soldiers from one side with a critical shot. It doesn't need additional data after the command.

Here's an example "input.txt" content:
**A 1 100,50** (A soldier is added to side 1.)
**A 2 90,45** (A soldier is added to side 2.)
**F** (The two soldiers fight.)
**R 1** (A random reinforcement is called to side 1.)
**A 2 95,48** (Another soldier is added to side 2.)
**C** (A critical shot wipes out all soldiers from one side.)

## What were the special cases you tested?

**Overflow of Stack: (available in the testing section)**
Trying to push soldiers into a stack that's already full.

input.txt:
**A 1 90,90**
**(repeated 100 times)**
**A 1 90,90**

Expected output:

```
Add soldiers to side 1
S- H:90 S:90
Add soldiers to side 1
S- H:90 S:90
Add soldiers to side 1
S- H:90 S:90
Add soldiers to side 1
S- H:90 S:90
Add soldiers to side 1
S- H:90 S:90
Stack is full, can't add soldier.
```

**Removing Soldiers from the Empty Stack:**

input.txt:
**F**

Expected Output:

```
One of the sides has no soldiers left. Can't fight.
End Game Summary
```

**Initiating a Fight When One of the Stacks is Empty:**

input.txt:
**A 1 100,100**
**F**

Expected Output:

```
Add soldiers to side 1
S- H:100 S:100
One of the sides has no soldiers left. Can't fight.
End Game Summary
```

**Unrecognized Commands: (available in the testing section)**

Testing the program's handling of unexpected inputs.

input.txt:
**A 1 80,40**
**X 2 70,50**

Expected output:

```
Add soldiers to side 1
S- H:80 S:40
Unrecognized command: X
Unrecognized command: 2
Unrecognized command: 70,50
End Game Summary
```

## Did everything work as expected?

When I first implemented the solution, there were certain discrepancies between my expected outcomes and the program's results. Some edge cases weren't behaving as anticipated, especially when the stack reached its capacity and during certain command sequences. However, after rigorous testing, debugging, and incremental refinements to the code, I was able to align the program's behavior with the expected outcomes. The iterative process of coding, testing, and refining was crucial to ensuring that the solution was both robust and accurate.

## Include sample input/output from your program.

Mixing Commands in Different Orders

**1)**
input.txt:
**A 1 100,50**
**A 2 50,100**
**F**
**A 1 90,60**
**F**
**R 2**
**F**

Expected Output:

```
Add soldiers to side 1
S- H:100 S:50
Add soldiers to side 2
S- H:50 S:100
Side 1 hit Side 2 with 47 damage
Add soldiers to side 1
S- H:90 S:60
Side 2 hit Side 1 with 52 damage
Called reinforcements to side 2
S- H:35 S:32
Side 1 hit Side 2 with 51 damage
Soldier from side 2 has fallen
End Game Summary
Side 1 wins with 2 soldiers remaining
```

**2)**
Input.txt:
**A 1 100,50**
**A 2 90,60**
**F**
**A 1 80,55**
**A 2 85,57**
**F**
**F**
**R 1**
**R 2**
**A 1 95,62**
**A 2 88,50**
**F**
**C**
**A 2 75,48**
**A 1 98,65**
**F**
**R 2**
**F**
**A 1 70,40**
**A 2 72,42**
**F**
**F**
**R 1**
**C**

Expected Output:

```
Add soldiers to side 2
S- H:88 S:50
Side 2 hit Side 1 with 49 damage
Critical shot by Side 1
Side 2 is completely wiped out!
Add soldiers to side 2
S- H:75 S:48
Add soldiers to side 1
S- H:98 S:65
Side 2 hit Side 1 with 49 damage
Called reinforcements to side 2
S- H:39 S:25
Side 1 hit Side 2 with 52 damage
Soldier from side 2 has fallen
Add soldiers to side 1
S- H:70 S:40
Add soldiers to side 2
S- H:72 S:42
Side 2 hit Side 1 with 50 damage
Side 1 hit Side 2 with 49 damage
Called reinforcements to side 1
S- H:10 S:69
Critical shot by Side 2
Side 1 is completely wiped out!
End Game Summary
Side 2 wins with 2 soldiers remaining
```

## COMMENTS

**Describe the overall result of the assignment.**

The overall result of the assignment is a program that simulates a battle between soldiers using stacks to manage the combatants. The design includes mechanisms for adding reinforcements, simulating combat, and determining a victor when one side's soldiers are depleted.

**Was the programming project a success?**

The program was successful and worked as I wanted, the tests I made gave the results I expected, and when it did not, I corrected the errors and improved it.

**What would you do the same or differently next time?**

**Same:**
Stack Implementation: Stacks are appropriate for the type of last-in, first-out combat simulation described, and their use should be continued in similar scenarios.

Modular Design: A modular approach is crucial for maintaining and scaling the software, and this should be a standard practice.
Critical Shot Mechanism: Incorporating random elements like critical hits can make the simulation more interesting and should be kept.

**Differently:**
Input Validation: Strengthen the validation on the input.txt to prevent crashes or unexpected behavior from malformed input data.
Performance Profiling: It would be helpful to integrate performance profiling earlier in the development cycle to ensure the program runs efficiently.
User Experience: Even for non-interactive simulations, providing a way to easily understand the output and the course of the battle can improve the user experience, such as a more detailed log of the battle events.
Interactive Game: Since this game works with the input.txt file, it can be written with user input.
Automated Testing: Develop a suite of automated tests to simulate a variety of battle scenarios, which would help ensure that the program is handling all possible situations correctly.