Software Design Specification for Toy Robot Simulation

Author: Merco C’zar Salise

1. Project Overview

This document outlines the design of a C++ console application that simulates the movement of a toy robot on a 5x5 tabletop grid. The robot can receive commands to place itself on the table, move, rotate, and report its position and orientation. The application will ensure that the robot does not fall off the table, disregarding any commands that would cause it to do so.

Key Features:

- The robot will respond to the following commands:
  ```
  PLACE X,Y,F
  MOVE
  LEFT
  RIGHT
  REPORT
  ```
- The first command must be a valid PLACE command before any other commands are accepted.
- The robot’s movement is constrained to the boundaries of the table (5x5 units).
- Input will be handled from either a file or standard input.
- The program will be designed with modern C++ principles and focus on maintainability, error handling, and code robustness.

2. Functional Requirements

Commands:

1. **PLACE X,Y,F**: Places the robot on the tabletop at coordinates (X, Y) facing direction F (NORTH, SOUTH, EAST, or WEST). This must be the first command for any valid sequence.
2. **MOVE**: Moves the robot one unit forward in the direction it is currently facing.
3. **LEFT**: Rotates the robot 90 degrees counterclockwise.
4. **RIGHT**: Rotates the robot 90 degrees clockwise.
5. **REPORT**: Outputs the current X, Y, and F values, showing the robot's position and orientation.

Constraints:

- The robot must not move off the edge of the table.
- Commands should be ignored if they would cause the robot to fall off the table or if no valid PLACE command has been issued.

Error Handling:

- Input validation will ensure that commands follow the correct format.
- Any invalid commands or attempts to move the robot off the table will be ignored, and the program will continue to process subsequent valid commands.

3. Non-Functional Requirements

Performance:

- The program will process commands in real-time and should handle typical input sizes efficiently.

Usability:

- The application will be a command-line program with simple text-based input and output. It will be easy to use and handle errors gracefully, providing clear feedback to the user.

Testing:

- Unit tests will be provided to verify the correctness of the robot's movements, command processing, and error handling.

Modern C++ Features:

- **Smart Pointers**: For memory management, avoiding manual allocation and deallocation.
- **constexpr**: To define compile-time constants and type aliases.
- **Polymorphism and Interfaces**: To ensure flexibility and testability through the separation of responsibilities.

4. System Architecture

4.1 High-Level Overview

The system is designed following object-oriented principles, with distinct classes handling different responsibilities, such as command parsing, robot movement, and user input/output. The architecture adheres to the **Dependency Inversion Principle** to ensure loose coupling and maintainability.

Key Classes and Components:

- **iRobot Interface**: Defines basic movement and reporting functions that any robot implementation must have.
- **stdRobot** and **advRobot(not implemented)**: Concrete implementations of the `iRobot` interface with additional features as required.
- **TableTop**: Manages the 5x5 grid, checks boundaries, and tracks the robot's position.
- **MovementInterface**: Provides the execution interface for robot actions based on user commands.
- **MovementParser**: Parses input commands (either from the user or from a file) and delegates execution to the appropriate components.
- **UserInputListener**: Listens for user commands and feeds them into the system.
- **UserInputExecutor**: Executes commands parsed by the `MovementParser`.
- **Logger**: Keeps track of all actions executed and can optionally log them to a file.

- **CommandLineDisplay**: Provides output functionality to display menus and graphical representation

4.2 Detailed Class Descriptions

iRobot Interface

- **Responsibility**: Defines the core robot functionality, such as movement, rotation, and reporting.
- **Methods**:
  - `MOVE()`: Moves the robot one step forward.
  - `LEFT()`, `RIGHT()`: Rotates the robot 90 degrees in the respective direction.
  - `REPORT()`: Outputs the current position and orientation.

stdRobot and advRobot

- **Responsibility**: Implements the `iRobot` interface. `stdRobot` provides the base functionality, while `advRobot` might include additional features like extended movement capabilities or advanced reporting, but this will be for future implementation.
- **Attributes**:
  - `Face`: Current orientation (NORTH, SOUTH, EAST, WEST).
  - `X pos, Y pos`: Current position on the table.

NOTE:
AdvRobot is not implemented yet

TableTop

- **Responsibility**: Manages the 5x5 grid and ensures that robots do not fall off.
- **Methods**:
  - `CheckBounds()`: Ensures the robot's movements are within the table boundaries.
  - `PLACE()`: Places the robot on the table at the given coordinates.
  - REMOVE(): Removes the robot on the table.

MovementParser

- **Responsibility**: Parses input commands from either a file or user input and translates them into executable actions.
- **Methods**:
  - `parseFileInput()`: Parses commands from a file.
  - `parseUserInput()`: Parses commands from standard input.

UserInputListener

- **Responsibility**: Listens for user input and provides it to the system for processing.
- **Methods**:
  - `ListensToUserInputs()`: Continuously listens for and validates user inputs.

Logger

- **Responsibility**: Logs actions taken by the robot, saving them to a file upon end of program or in a set of time intervals.
- **Singleton:** logger has been implemented as a singleton to avoid resource timing access issues.
- **Methods**:
  - `logListToFile()`: Writes the log of actions to a file.

CommandLineDisplay

- **Responsibility**: Handles majority of the output to the console.

5. Command Line Arguments

1. -h: Help

The -h flag provides a help menu that displays usage information, including the available command-line options and how to execute the program. This is useful for users unfamiliar with the program or its command-line arguments.

Usage:

```
.\ToyRobotApp.exe -h
```

Output:

```
Usage: ToyRobotApp.exe [options]

Options:
  -h                Show help menu and exit
  -m                Run the program in manual input mode
  -f <file-path>    Run the program with commands from a specified file

If no option is set, simple UI is selected
```

2. `-m`: Manual Input Mode

The `-m` flag allows the user to input robot commands manually via the command line. In this mode, the program prompts the user to enter a series of robot movement commands (e.g., `PLACE`, `MOVE`, `LEFT`, `RIGHT`, `REPORT`) one by one.

Usage:

```
.\ToyRobotApp.exe -m
```

Features:

- Commands are case-insensitive.
- Command input can be terminated with a `;` or `.`.
- Default values are provided for incomplete PLACE commands (`0,0,`NORTH).
- The program handles user errors with friendly prompts.

3. `-f`: File Input Mode

The `-f` flag allows the program to execute robot movement commands from a file. This mode is useful for batch processing of commands or testing predefined command sequences. The specified file should contain robot commands in the same format as manual input.

Usage:

```
.\ToyRobotApp.exe -f <path-to-file>
```

Example:

```
.\ToyRobotApp.exe -f ..\..\sample_moveset.txt
```

The file content might look like this:

```
PLACE 1,2,EAST
MOVE
MOVE
LEFT
MOVE
REPORT
```

The program processes each command and displays the robot's final state after executing all commands in the file.

6. Testing

Unit Tests:

- Each class will have associated unit tests, focusing on the following areas:
    - Correct handling of boundaries (e.g., robot cannot move off the table).
    - Validation of input commands.
    - Rotation and movement logic.
    - Error handling for invalid input or movement attempts.

Integration Tests:

- Not implemented