BLUFF BAR
BLUFF BAR
BLUFF BAR

Presenting to : Sir Mughees

Game Presentation

OUR TEAM

Mohid Irfan

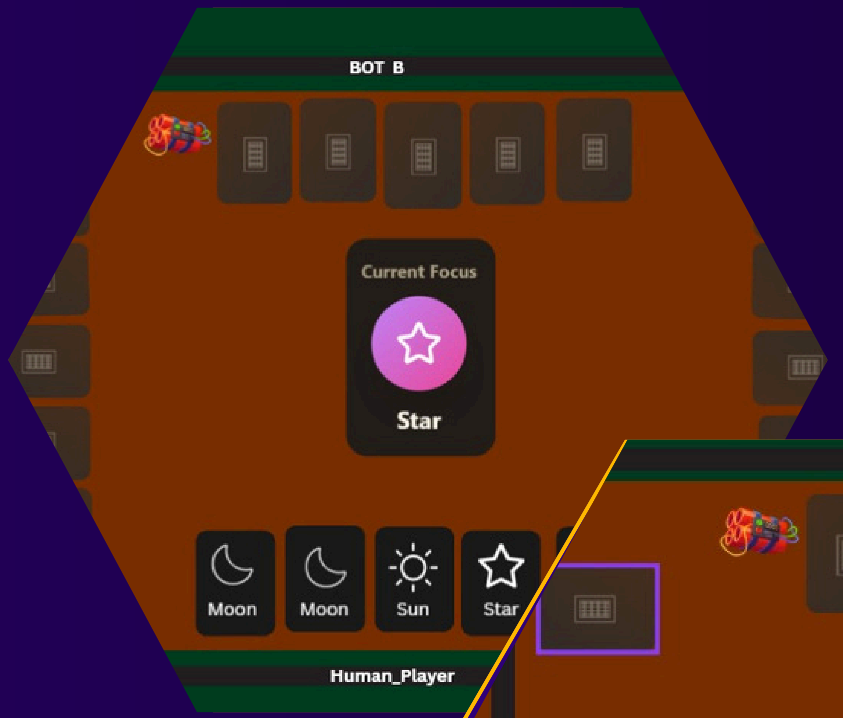
Ahmad Rumman

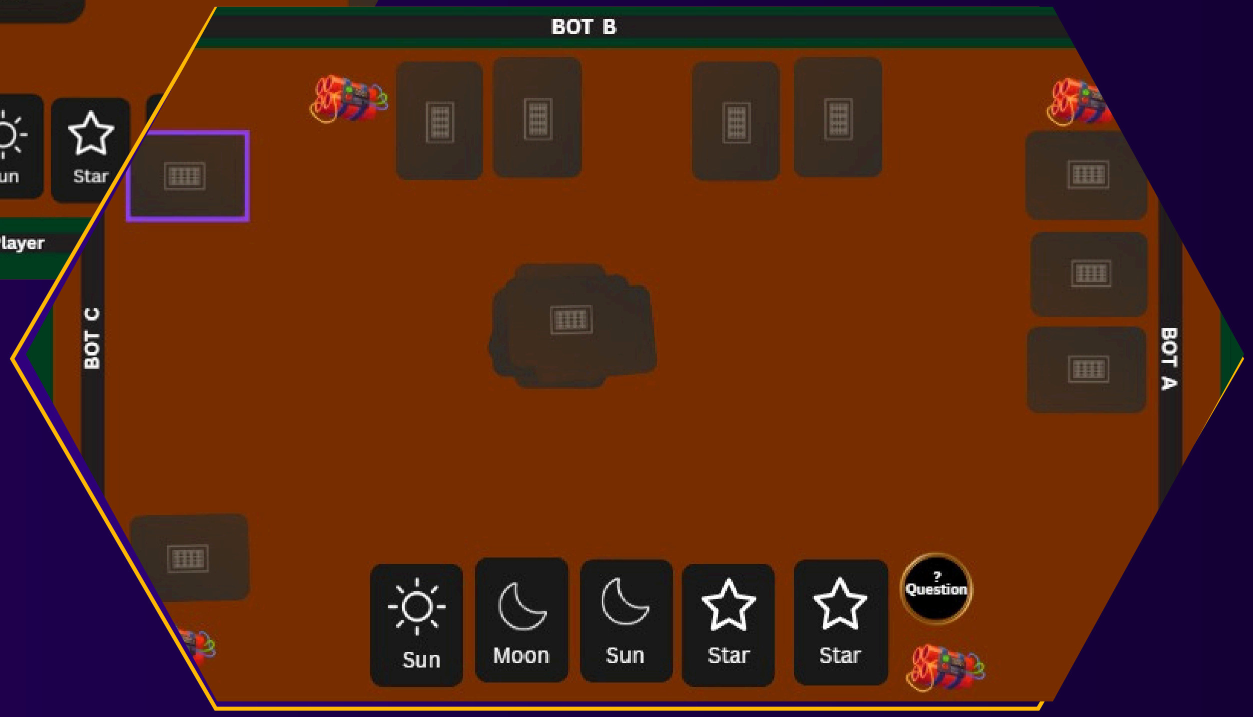
Ahad Babar

Sumiya

WELCOME TO OUR GAME

Bluff Bar is a strategy-driven card game that challenges players with bold risks, smart decisions, and unpredictable bluffs. This modern version adds opponents, multi-card moves, and dynamic bomb mechanics, keeping each round tense and thrilling. Built using C++ and OOP concepts in VS Code, it turns simple bluffing into a high-intensity mind game.





PROBLEM STATEMENT

Traditional card games like “Liar Bar” are limited in strategy and risk, allowing only single-card plays and requiring multiple human players. Bluff Bar solves this by adding multi-card moves, wildcard mechanics, and a bomb system, making gameplay more strategic, unpredictable, and thrilling.

DESCRIPTION

Focus Card Bluff Challenge is a logic-based multiplayer card game played between four players: Human, Bot1, Bot2, and Bot3.
Each round starts with a Focus Card (Sun, Moon, or Star), and players must play 1–3 matching cards or use a Magic wildcard.
Players can bluff, and the next alive player can either Question or Not Question.
If questioned, the game checks the truth: the wrong player becomes guilty.
A guilty player faces a 1-in-3 Bomb chance—if it explodes, the player dies and is removed.
Turn order cycles Human → Bot1 → Bot2 → Bot3->Human, skipping dead players.
After any questioning event, the round ends and new cards are dealt to alive players.
The next round always starts from the player who questioned.
Gameplay continues with rounds of bluffing, checking, and bomb risks.
The game ends when only one player survives—that player is the winner.

OOP CONCEPTS

Encapsulation :

Protected data and functions within classes, e.g., Player class keeps cards and score private, accessed only via getter/setter functions.

Polymorphism :

Same function name, different behavior, e.g., show() in Card and Bomb classes outputs differently (function overriding).

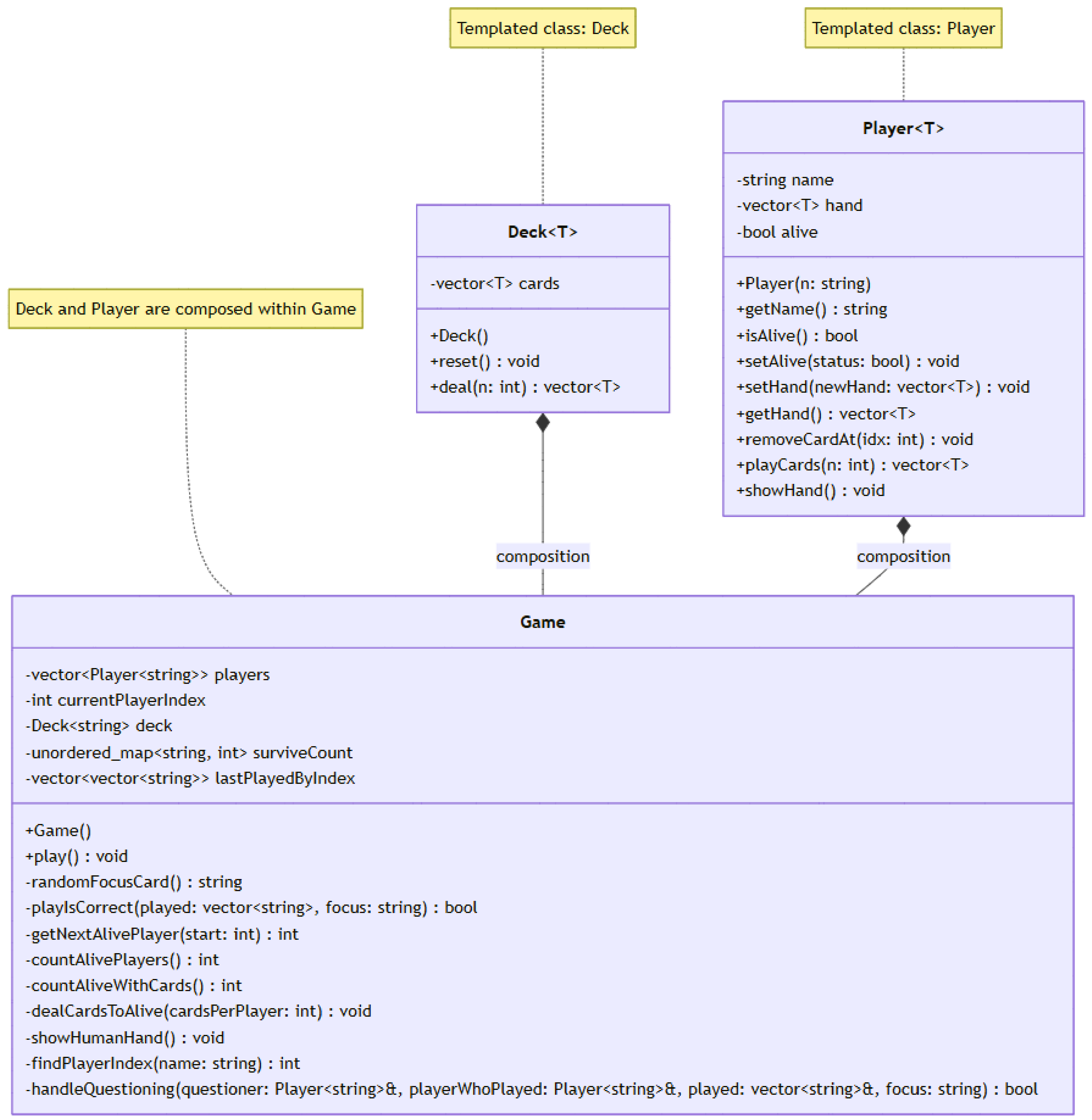
Composition :

Class contains objects of another class, e.g., Player class contains a Hand object to manage cards.

Exception Handling :

Handles game errors using try-catch blocks, e.g., invalid card moves or empty deck scenarios are caught and managed.

CLASS DIAGRAM



Deck<T>

-vector<T> cards

+Deck()
+reset() : void
+deal(n: int) : vector<T>

Player<T>

-string name
-vector<T> hand
-bool alive

+Player(n: string)
+getName() : string
+isAlive() : bool
+setAlive(status: bool) : void
+setHand(newHand: vector<T>) : void
+getHand() : vector<T>
+removeCardAt(idx: int) : void
+playCards(n: int) : vector<T>
+showHand() : void

Game

-vector<Player<string>> players
-int currentPlayerIndex
-Deck<string> deck
-unordered_map<string, int> surviveCount
-vector<vector<string>> lastPlayedByIndex

+Game()
+play() : void
-randomFocusCard() : string
-playIsCorrect(played: vector<string>, focus: string) : bool
-getNextAlivePlayer(start: int) : int
-countAlivePlayers() : int
-countAliveWithCards() : int
-dealCardsToAlive(cardsPerPlayer: int) : void
-showHumanHand() : void
-findPlayerIndex(name: string) : int
-handleQuestioning(questioner: Player<string>&, playerWhoPlayed: Player<string>&, played: vector<string>&, focus: string) : bool

SYSTEM ARCHITECTURE

Presentation Layer (Top Layer)

- This is where the human player interacts with the game.
- It uses cin to take inputs like how many cards to play and which cards.
- It uses cout to show the human's hand and other messages.
- It never controls the game rules — it just talks to the middle layer to ask or show things.

SYSTEM ARCHITECTURE

Application Layer (Middle Layer)

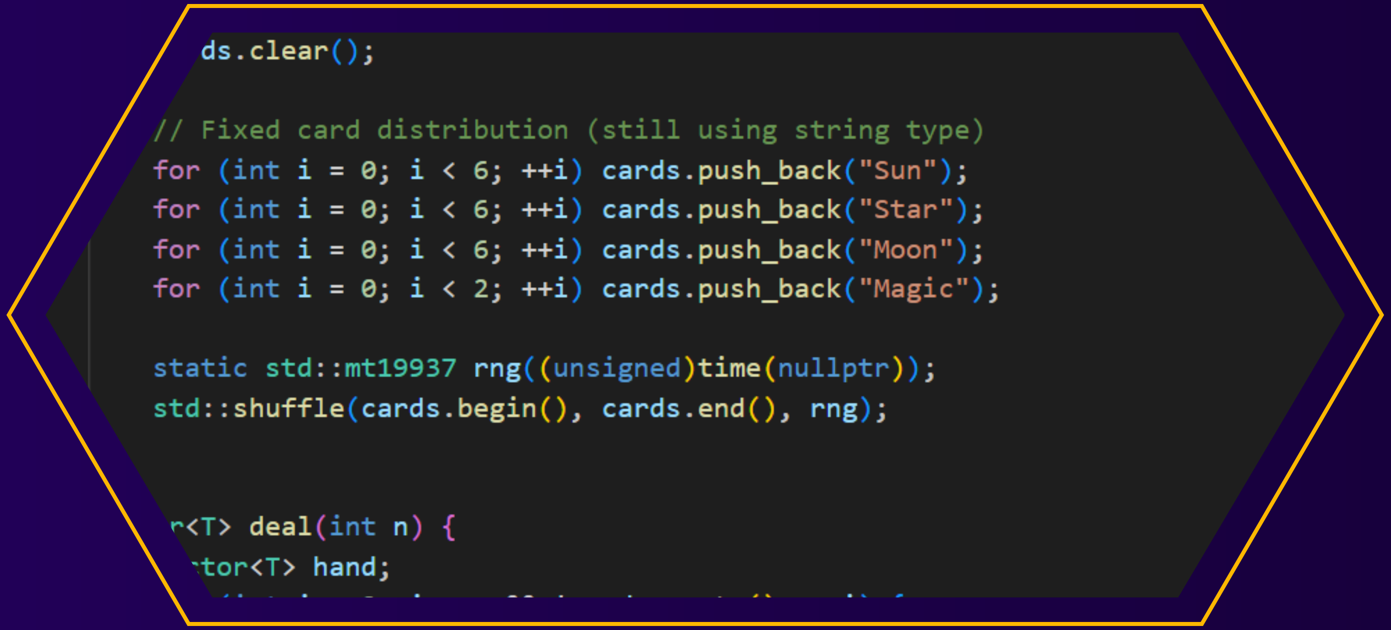
- This is your Game class, the brain of the system.
- It controls the flow of the game: whose turn it is, when the round ends, and who wins.
- It handles questioning: checks if plays are correct and decides if a player is eliminated.
- It keeps track of things like survival counts, current focus card, and last played cards.
- It uses the Data Layer classes (Deck and Player) but doesn’t store cards or players itself.

SYSTEM ARCHITECTURE

Data Layer (Bottom Layer)

- Contains your Deck<T> and Player<T> classes.
- Deck<T> stores all the cards, shuffles them, and deals them out.
- Player<T> stores each player's name, hand, and alive status.
- It also provides functions for players to play cards, remove cards, and show their hand.
- This layer only holds data and card collections, and doesn’t know about game rules or flow.

INTERACTION OF CLASSES

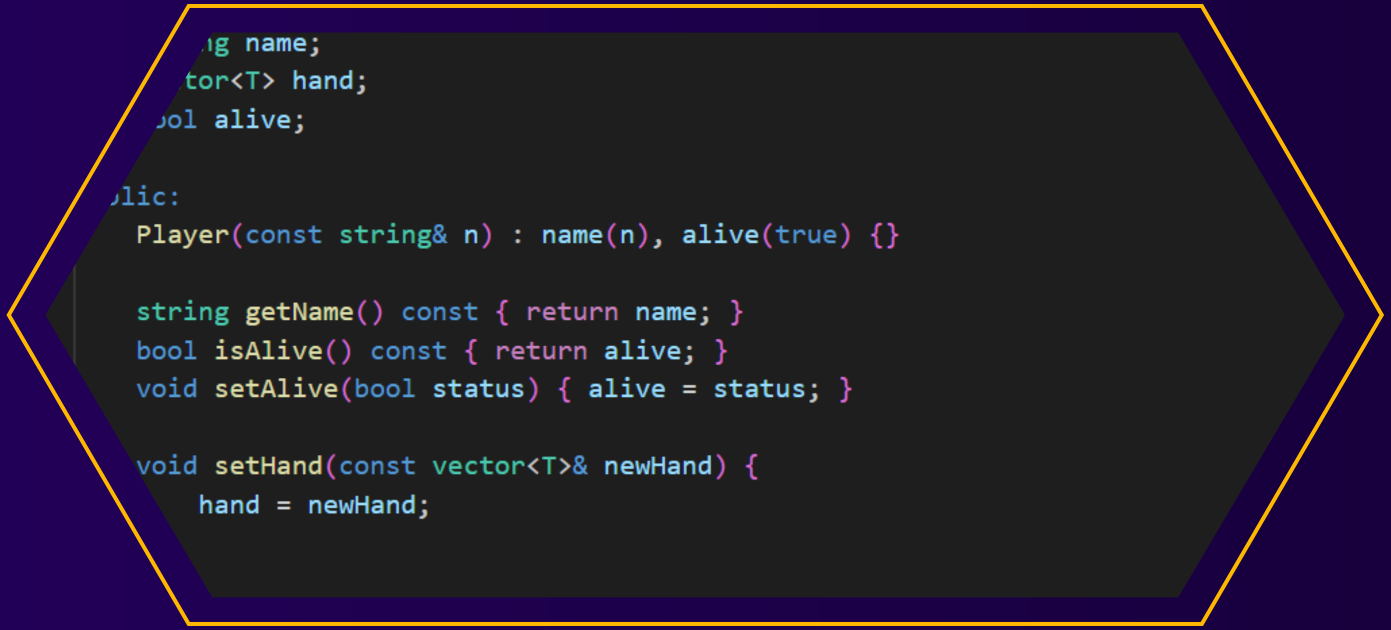


```
ds.clear();

// Fixed card distribution (still using string type)
for (int i = 0; i < 6; ++i) cards.push_back("Sun");
for (int i = 0; i < 6; ++i) cards.push_back("Star");
for (int i = 0; i < 6; ++i) cards.push_back("Moon");
for (int i = 0; i < 2; ++i) cards.push_back("Magic");

static std::mt19937 rng((unsigned)time(nullptr));
std::shuffle(cards.begin(), cards.end(), rng);


r<T> deal(int n) {
tor<T> hand;
```

Game and Deck<T>

- The Game class creates and controls the deck.
- It calls Deck’s reset() to initialize and shuffle the cards.
- It calls Deck’s deal(int n) method to distribute cards to players.

INTERACTION OF CLASSES

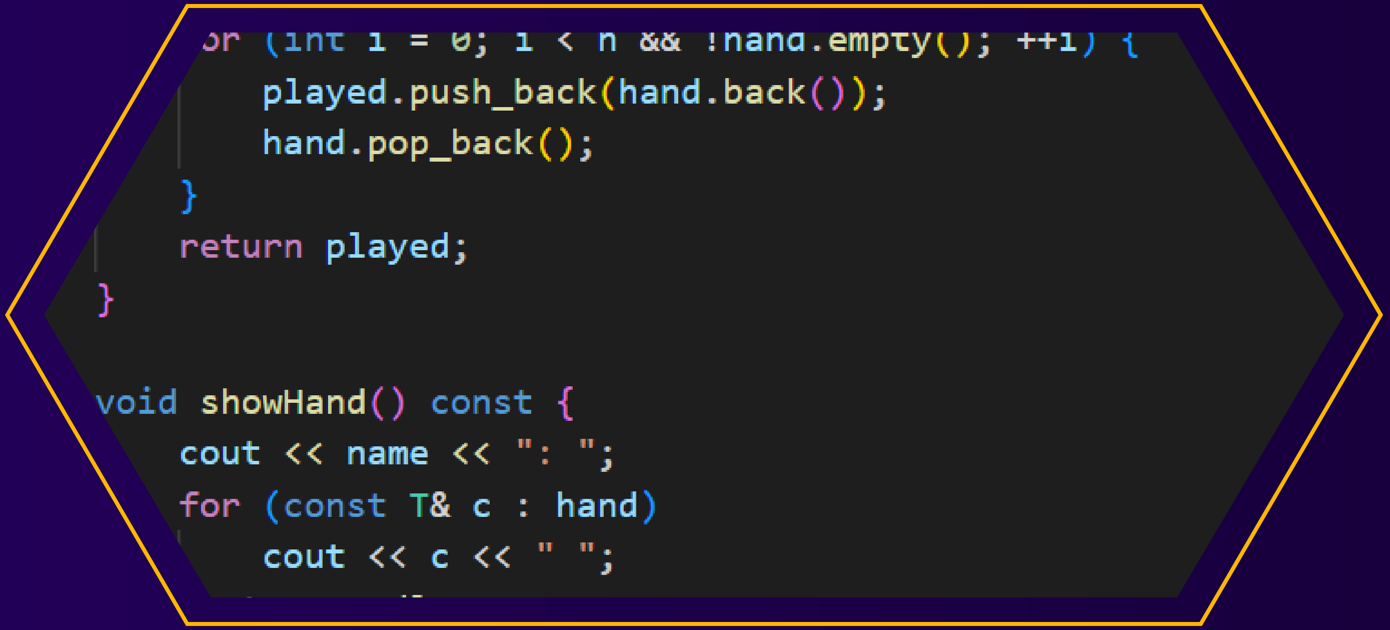


```
 ng name;
 tor<T> hand;
 ol alive;

lic:
   Player(const string& n) : name(n), alive(true) {}

   string getName() const { return name; }
   bool isAlive() const { return alive; }
   void setAlive(bool status) { alive = status; }

   void setHand(const vector<T>& newHand) {
       hand = newHand;
```

Game and Player<T>

- The Game creates players and stores them in a vector.
- It tells players when to play cards by calling their playCards() method.
- It updates player status (alive or dead) based on game events.
- It accesses players’ hands to check if they can play and to display the human player’s hand.

INTERACTION OF CLASSES



```
    or (int i = 0; i < n && !hand.empty(); ++i) {
        played.push_back(hand.back());
        hand.pop_back();
    }
    return played;
}

void showHand() const {
    cout << name << ": ";
    for (const T& c : hand)
        cout << c << " ";
```

Player<T> and Deck<T>

- Players receive cards from the deck but don’t directly control the deck.
- Players manage their own hands (cards dealt from the deck).
- Players remove cards from their hand when they play.

FLOW OF DATA CONTROL

Input

The user starts the game.
Players enter their names.
Game receives player moves (play card, bluff, challenge).

Processing

Game creates a Deck and shuffles it.
Game creates Player objects and gives them cards.
During each turn:
Player plays a card.
Game checks if the move is valid.
If bluffing: game verifies truth or lie.
Game updates player hands.
Game moves to next player.
Game keeps doing this until one player has no cards left.

Output

Game shows:
Valid / invalid move messages.
Cards played.
Updated turn information.
Final winner at the end

FEATURES

- The bomb has 3 chances to explode.In Bluff Bar bomb explodes automatically at 3rd chances while in Liars Bar bomb doesn’t explode automatically. Player has to cut wire.
- In Liar Bar, a player can throw only one card at a time while in Bluff Bar, a player can throw at least three cards at a time with specifying index.
- In Liar Bar, there is no concept of “Wild Card”. A wild card is a magic card that behaves as a round card in each round.

FEATURES

- In “Liar Bar”, no random focus card occurs at each round. Only one focus card retains till last. But in Bluff Bar, new random focus card appears at each round.

- Liar Bar requires 4 human players to play game. In Bluff Bar we generate 3 Bots therefore Bluff Bar requires only one human player to play game.

- In Liar Bar only wrong Answer player becomes guilty and has to cut bomb while in Bluff Bar both players can become guilty(challenged and challenger) so both can cut bomb. This feature is increasing risk in game.

CHALLENGES

- Player Acting Like Bot - Fixed the human player acting automatically without user input
- Bomb Probability - Made the 1/3 bomb chance work correctly (it was exploding too often)
- Magic Card Recognition - Fixed the game not recognizing special Magic cards properly
- Single Correct Card Bug - Fixed where if only one card was correct, the game thought ALL cards were correct

LEARNING OUTCOMES

- Classes - Blueprints for making game parts
- Objects - Actual game parts made from blueprints
- Encapsulation - Keeping game data safe and private
- Abstraction - Making complex things simple to use
- Composition - Building big things from smaller parts
- Templates - Making code work with different types of data

SKILLS IMPROVED

- Modular Coding - Making separate, reusable pieces
- Logic Building - Turning game rules into working code
- Debugging - Finding and fixing errors in code
- System Design - Planning how all pieces fit together
- Problem Solving - Figuring out solutions to coding problems

FUTURE IMPROVEMENTS

- Graphical Interface - Adding pictures and buttons instead of just text
- Online Multiplayer - Letting people play together over internet
- More Features - Adding new cards, rules, and game mode

SUMMARY

This project is a console-based card game developed using Object-Oriented Programming concepts. The system is divided into three modules—Game, Player, and Deck—each handling a specific responsibility. The Game module controls the overall flow, distributes cards, manages turns, and checks valid or invalid moves. The Deck module generates and shuffles cards, while the Player module stores player data and actions. The program follows a simple input → processing → output flow to run the game until a winner is decided. Overall, the project demonstrates modular design and strengthens understanding of class interaction and OOP structure.

THANK YOU

THANK YOU