WHITEFIELD
BUSINESS SCHOOL

PROGRAMMING IN C++

CONSOLE GAME PROGRAMMING

TUTORIAL 1
UNDERSTANDING THE PLAYER.H

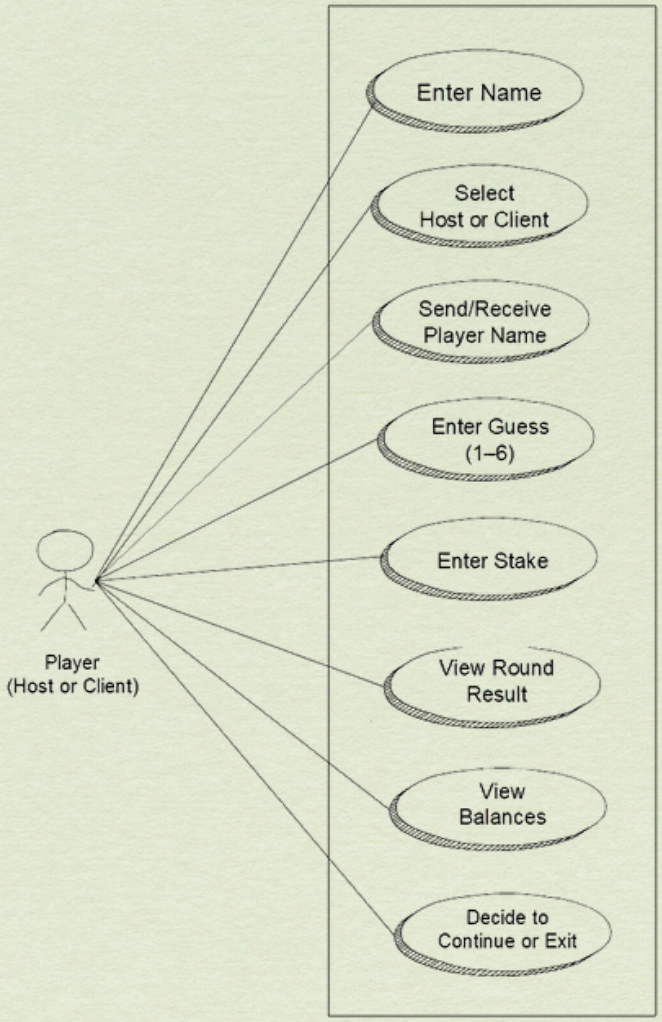
Introduction to Use-Case Diagram

- A use-case diagram is a vital tool in software development that visually represents how users (known as actors) interact with a system to achieve specific goals or functionalities.
- It captures the functional requirements of a system by illustrating the various use cases — or actions — that users can perform, along with the relationships between the users and those actions.
- Use-case diagrams help developers, designers, and stakeholders understand what the system is expected to do from the user's perspective, ensuring clarity, communication, and proper feature planning early in the development lifecycle.

Use-Case Diagram for the Player Class

- The use-case diagram for the Player class is important because it clearly illustrates the specific interactions a player (the human user) has with the game system.
- While the Player class in code holds data and behavior related to a single game participant, the use-case diagram shifts the perspective to show how a player uses the system — such as entering their name, choosing to host or join, placing a stake, making a guess, and deciding to continue or quit.
- This diagram is essential for understanding how the player fits into the overall gameplay flow. It ensures that all necessary functionalities related to player interaction are identified and accounted for during design.
- Moreover, it helps bridge the gap between technical implementation and user expectations, enabling developers to align system behavior with real-world use.

Use-Case Diagram



DESCRIPTION

- **Enter Name**
  - The player inputs their display name at the start of the game.
- **Select Host or Client**
  - The player decides whether to act as the server or connect as a client.
- **Send/Receive Player Name**
  - Names are exchanged between host and client over the network.
- **Enter Guess (1–6)**
  - The player chooses a number between 1 and 6 before the die roll.
- **Enter Stake**
  - The player bets a portion of their virtual balance.

- **View Round Result**
  - The player sees the outcome of the die roll and the round winner.
- **View Balances**
  - The system shows updated balances for both players after each round.
- **Decide to Continue or Exit**
  - After each round, the player chooses whether to play another round or quit the game.

DERIVING THE PLAYER CLASS DEFINITION

When designing a class like Player:

- Start with what real-world information needs to be tracked.

  → *Name, money, role (host or client)*

- Decide how those values will be stored.

  → *Use string, int, and bool.*

- Think about what the game needs to do with those values.

  → *Display them, modify them, or retrieve them.*

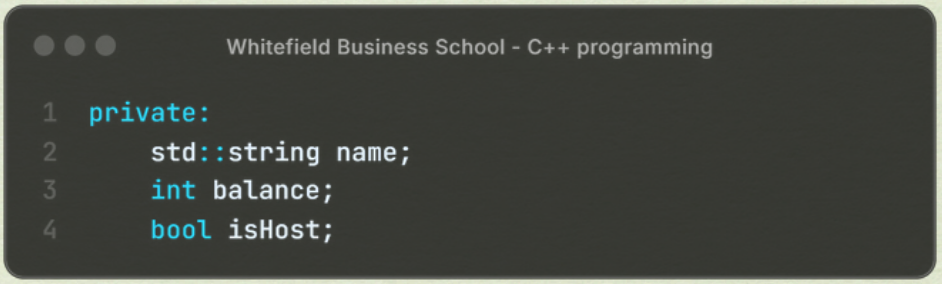
This leads you naturally to:

- Getter methods for read-only access
- Modifier methods for logic updates
- A constructor to ensure initialization happens correctly

This leads you naturally to:

- Getter methods for read-only access
- Modifier methods for logic updates
- A constructor to ensure initialization happens correctly

Instance Variables (Data Members)



Explanation:

name:

- A string to store the player's name (used for display, networking, or logs).

balance:

- An integer representing the player's current money or points (starts at 100).

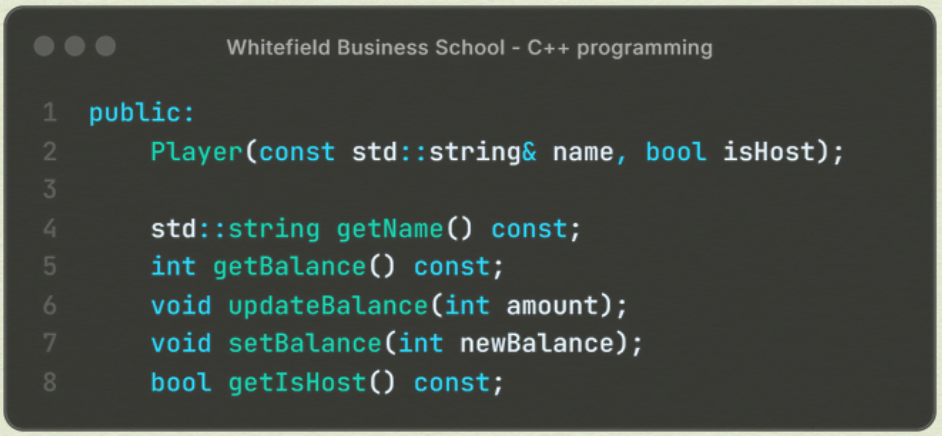
isHost:

- A boolean flag to indicate whether the player is the server/host (true) or client (false).

These are instance variables because each Player object (Alice, Bob, etc.) will hold its own copy of these values.

Methods (Member Functions)

These are declared in the public: section. Methods are actions or queries a player can perform or that the game system can perform on the player.



Constructor

```
Whitefield Business School - C++ programming
1  Player(const std::string& name, bool isHost);
```

- Initializes a player with a name and role (host/client).
- Internally sets balance to 100.

Getter Methods

- These allow reading private values without modifying them.

```
Whitefield Business School - C++ programming
1  std::string getName() const;
2  int getBalance() const;
3  bool getIsHost() const;
```

- getName() → returns the player's name.
- getBalance() → returns the current balance.
- getIsHost() → tells whether the player is acting as the host.

Why const Is Used for Methods Like getName()?

When you declare a method like this:

```
Whitefield Business School - C++ programming
1  std::string getName() const;
```

You're telling the compiler:

"This method is read-only — it guarantees not to modify any member variables of the object."

So Why Do We Do That?

Because it's a safeguard — a promise that:

- You (or any future developer) cannot accidentally change internal data inside a method that's meant to just "look at" the object.
- The compiler will raise an error if you try to modify something inside a const method.

Example — Protective Behavior

```
Whitefield Business School - C++ programming

class Player {
private:
    std::string name;

public:
    std::string getName() const {
        name = "Hacker"; // ❌ Error: cannot modify in const method
        return name;
    }
};
```

- The line name = "Hacker"; will cause a compiler error, because it's trying to modify a member variable in a method marked const.

It's Best Practice

Using const on accessors like getName(), getBalance(), or getIsHost() is considered good C++ practice, because:

- It enforces read-only behavior.
- It improves code safety and clarity.
- It allows these methods to be called on const Player objects.

What if We Didn't Use const?

Then this would compile fine:

```
Whitefield Business School - C++ programming

std::string getName() {
    name = "Oops";  // Compiles, even if we didn't want it to.
    return name;
}
```

But that violates the intent of a getter — it's supposed to give you data, not mutate it.

Getter Methods

- These change the internal state of the object.

```
Whitefield Business School - C++ programming
1  void updateBalance(int amount);
2  void setBalance(int newBalance);
```

- updateBalance() → increases or decreases balance.
- setBalance() → forcibly sets the balance to a given value (used when syncing between players in multiplayer mode).

Summary

Element	Type	Description
`name`	Variable	Player identity
`balance`	Variable	Player's money
`isHost`	Variable	Role of player in networked game
`getName()`	Method	Returns player's name
`getBalance()`	Method	Returns player's current balance
`getIsHost()`	Method	Returns if this player is the host
`updateBalance()`	Method	Adjusts balance by amount (±)
`setBalance()`	Method	Assigns new balance (clamped to ≥ 0)

Full Game Round Flow

It can be split into two perspectives:

- Host Player flow
- Client Player flow

Host Player Flow

Step-by-step Sequence:

1. **Prompt for Guess**
    - The host's console prompts: "Enter your guess (1–6)".
    - Player inputs a number, e.g., 3.
2. **Send Guess Over Network**
    - GameManager calls NetworkManager.sendMessage("GUESS|3") to inform the client.
3. **Prompt for Stake**
    - Console prompts: "Enter your stake".
    - Host enters a value (e.g., $20), validated against balance.
4. **Send Stake**
    - GameManager sends STAKE|20 to the client.
5. **Wait for Remote Input**
    - GameManager waits for client's guess and stake using receiveMessage() twice.
6. **Roll the Die**
    - As the host, the GameManager triggers rollDie() → random number (e.g., 4).
7. **Send Die Result**
    - GameManager sends ROLL|4 to the client.

8. Evaluate Round Outcome

- Using both players' guesses and stakes, the GameManager decides:
  - Who won or lost
  - How balances should be updated

9. Update Local Balance

- GameManager calls localPlayer.updateBalance(...).

10. Send Balance to Remote

- GameManager sends the new local balance.

11. Receive Remote Balance

- Remote balance is received and stored via remotePlayer.setBalance(...).

12. Display Result

- The outcome is printed in the console for the host.

Client Player Flow

Step-by-step Sequence:

1. **Wait for Remote Input**
   - GameManager receives host's guess and stake using receiveMessage() twice.
2. **Prompt for Guess**
   - Console prompts: "Enter your guess (1–6)"
   - Client inputs their number, e.g., 5.
3. **Send Guess**
   - GameManager sends GUESS|5 to host.
4. **Prompt for Stake**
   - Console prompts: "Enter your stake".
   - Client enters amount, e.g., $15.

5. Send Stake

- GameManager sends STAKE|15 to host.

6. **Wait for Die Result**

- GameManager receives ROLL|4 from the host.

7. Evaluate Round Outcome

- Based on guesses and die roll, outcome is determined.

8. Update Local Balance

- GameManager adjusts local balance via updateBalance(...).

9. Send Balance

- Client sends updated balance to host.

10. **Receive Remote Balance**

- Client gets the host's new balance and stores it.

11. **Display Result**

- Console shows result for the round and both balances.

Common Ending for Both

- If either balance becomes 0, or a player exits, the game ends.
- Otherwise, both are prompted:

  → "Play another round? (y/n)"