**Intergalactic Warfare: A Turn-Based Strategy Game**

**Object-Oriented Programming Course Project**

**Team: Syeda Fatima Tu Zahra(2024625)**

**Syed Rayyan Hassan( )**

**Abdullah Fawad Khan(2024038)**

**1. Introduction**

**This project implements a console-based strategy game simulating conflict between human and alien factions using core OOP principles. Designed in C++, the system features:**

* **Asymmetric factions with unique units and economies**
* **Dynamic combat with damage distribution algorithms**
* **Resource management tied to recruitment systems**
* **Mission-based progression with victory conditions**

**The architecture demonstrates how polymorphism and inheritance can model complex game entities while maintaining clean code separation.**

**2. Motivation & Background**

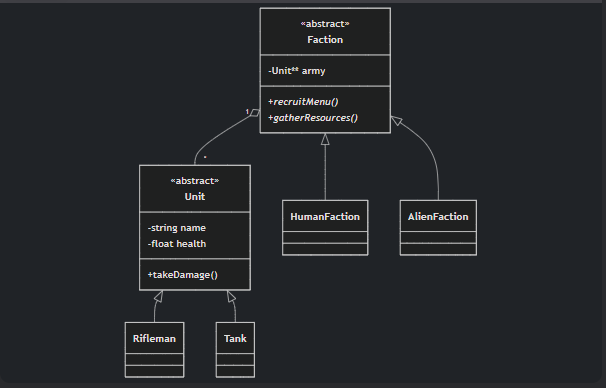
**Traditional strategy games often share core mechanics between factions, but implementing true asymmetry presents unique OOP challenges. This project was conceived to:**

1. **Solve the "dual hierarchy problem" (maintaining parallel human/alien systems without code duplication)**
2. **Explore polymorphic damage calculation where unit types affect damage distribution**
3. **Implement dynamic army management using pointer arrays and bitwise expansion**

**Inspired by games like *Starcraft* and *Advance Wars*, the system proves how abstract base classes can enable faction diversity.**

**3. System Architecture**

**3.1 Class Structure**

****

**Key Components:**

* **Unit Hierarchy: Base class defines shared combat attributes, while derived classes (Rifleman, Zorg, etc.) specify faction-unique stats**
* **Faction System: Abstract Faction handles common army management, with derived classes implementing faction-specific:**
  + **gatherResources() (Human: random 1-40, Alien: random 1-40 + 10% bonus)**
  + **recruitMenu() (Different unit types and costs)**

**3.2 Core Algorithms**

**The combat system implements a polymorphic damage distribution algorithm through three key phases:**

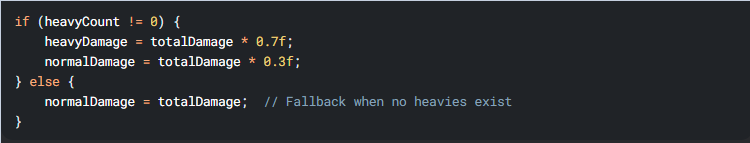
**1. Unit Classification**

**The system first categorizes units using the base Unit class's getIsHeavy() method:**

****

**2. Asymmetric Damage Distribution**

**Damage follows a 70/30 split favoring heavy units, implemented exactly as:**

****

**3. Memory-Safe Resolution**

**Dead units are removed via your existing removeUnit() method, which handles:**

* **Memory deallocation (delete army[index])**
* **Array compaction (index-shifting loop)**
* **Size tracking (armySize--)**

**Key Design Advantages:**

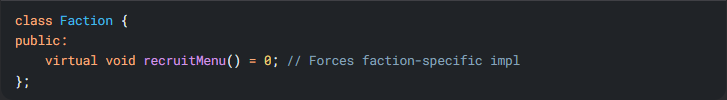
1. **RAII Compliance: Automatic cleanup via Faction destructor**
2. **Polymorphism: Works for all Unit subclasses (Rifleman, Tank, etc.)**
3. **Balance: The 70/30 split creates tactical unit prioritization**

***Implementation Note: All combat math uses floating-point division to prevent integer truncation artifacts.***

**4. OOP Principles Applied**

**4.1 Inheritance & Polymorphism**

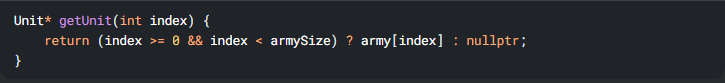
* **Virtual Methods:**

****

* **Human/Aliens implement different recruitment logic while sharing the same interface**
* **Abstract Base Classes:  
  Unit and Faction define contracts that derived classes must fulfill**

**4.2 Encapsulation**

* **All unit stats (health, attackPower) are private**
* **Army array access controlled via methods:**

****

**4.3 Resource Management**

**RAII (Resource Acquisition Is Initialization) Principle:**

**RAII is a fundamental C++ programming concept that ties resource management to object lifetimes. The core idea:  
"Resources should be acquired during object initialization and released automatically when the object is destroyed."**

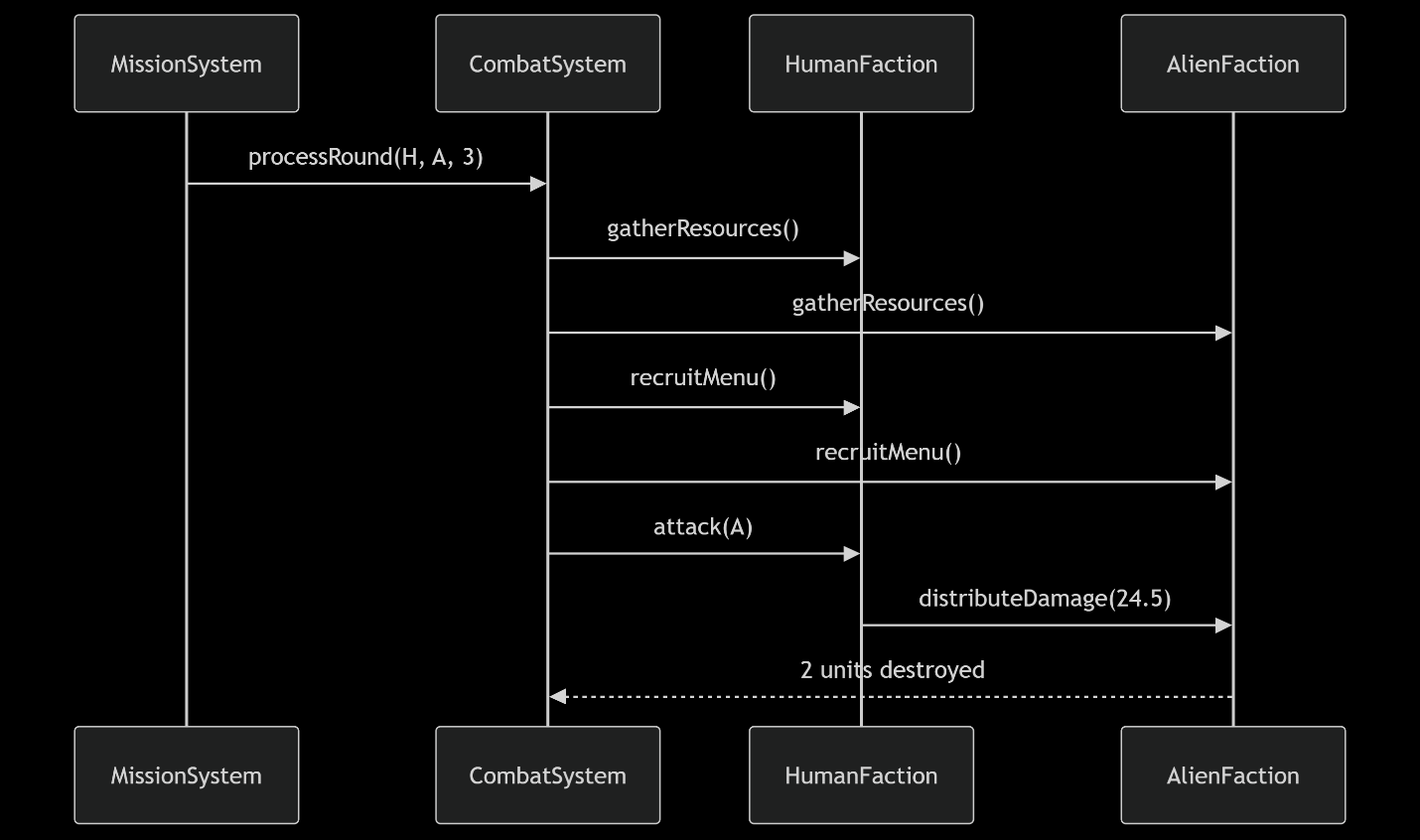
* **RAII Principle: Destructors automatically handle memory cleanup:**

**A black background with white text

AI-generated content may be incorrect.**

**Key Characteristics of RAII**

1. **Acquisition in Constructors**
   * **Resources (memory, files, locks, etc.) are allocated when an object is created**
   * **Example: Opening a file in a class constructor**
2. **Release in Destructors**
   * **Resources are automatically freed when the object goes out of scope**
   * **Prevents leaks even if exceptions occur**
3. **Ownership Semantics**
   * **The object that acquires a resource owns it**
   * **No manual delete or close() calls needed**
4. **Execution Flowchart**



**6. Conclusion**

* **This project successfully demonstrates the power of object-oriented programming in game development, particularly through:**
* **Effective Polymorphism**
* **The Faction and Unit hierarchies allowed for clean faction asymmetry while maintaining shared combat logic.**
* **Virtual methods (recruitMenu(), gatherResources()) enabled faction-specific behaviors without code duplication.**
* **Robust Resource Management**
* **The RAII-compliant Faction class ensured automatic memory cleanup, preventing leaks.**
* **Dynamic army expansion (expandArmy()) handled growth efficiently using bitwise operations.**
* **Strategic Gameplay Through OOP**
* **The 70/30 damage distribution system encouraged tactical unit composition.**
* **Encapsulation (private army/resources) maintained data integrity while allowing controlled modification.**
* **Key Learnings:**
* **Inheritance + Polymorphism are ideal for modeling game entities with shared behaviors but unique implementations.**
* **Encapsulation is critical for managing complex game state without exposing unsafe operations.**
* **RAII eliminates manual memory management pitfalls, especially in dynamic systems.**
* **The project achieved its goal of creating an extensible, memory-safe strategy game while reinforcing core OOP principles. The architecture proves that clean abstractions can handle asymmetric gameplay elegantly.**