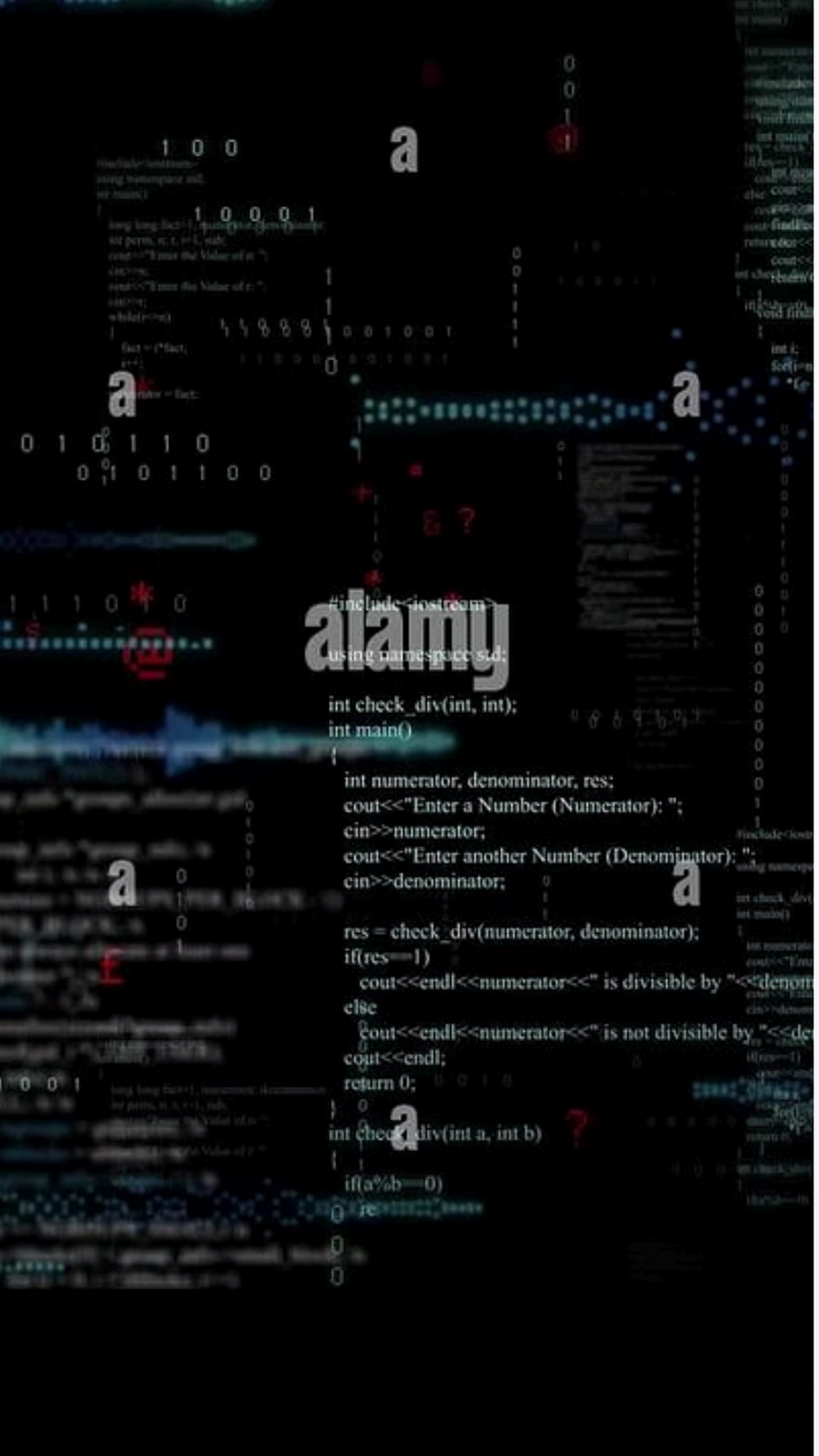
# University Management System

OOP Project in C++

**Ahmad Jamal Almashharawi**

Under the supervision of Dr. Samer Hanna




# Project Overview

A comprehensive management system built using Object-Oriented Programming principles in C++ to manage university entities including students and professors, demonstrating core OOP concepts and advanced practices.

### Record Management

Efficiently manage student and professor data.

### Data Persistence

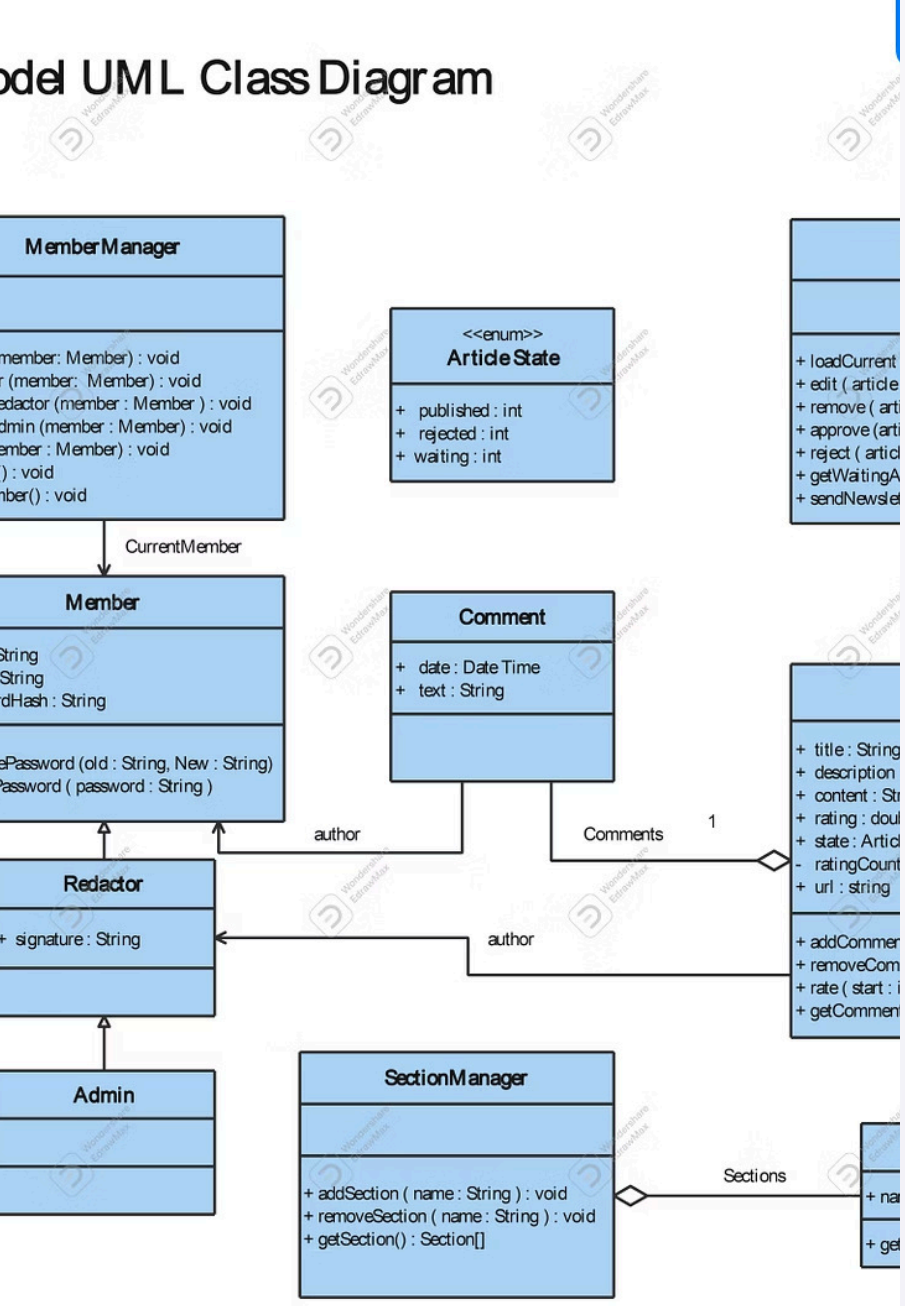
file-based storage for all records.

### Robust Handling

Integrated exception handling and input validation.

### Generic Repository

Template-based pattern for flexible data operations.




# System Architecture

## UML Class Diagram

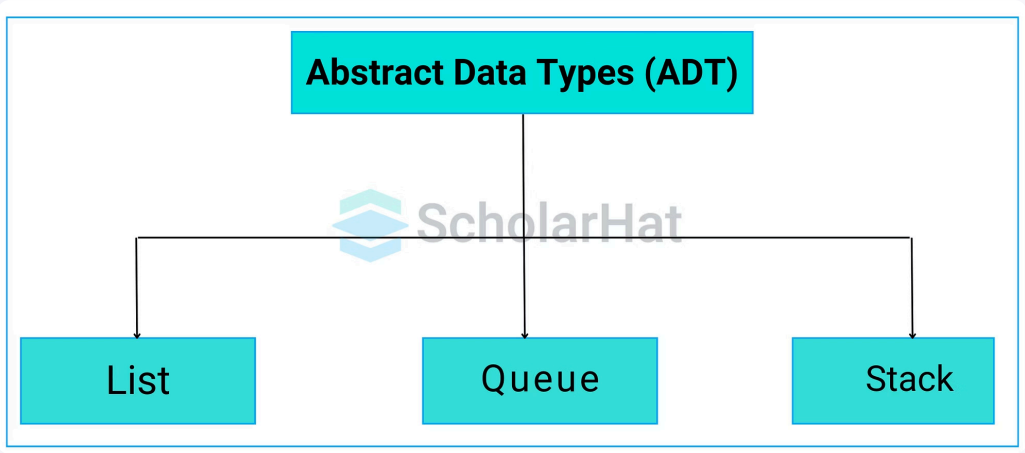
[UML Class Diagram will be inserted here]

This diagram illustrates all classes, their relationships, and the inheritance hierarchy, providing a clear visual of the system's structure.

# System Architecture - Core Classes

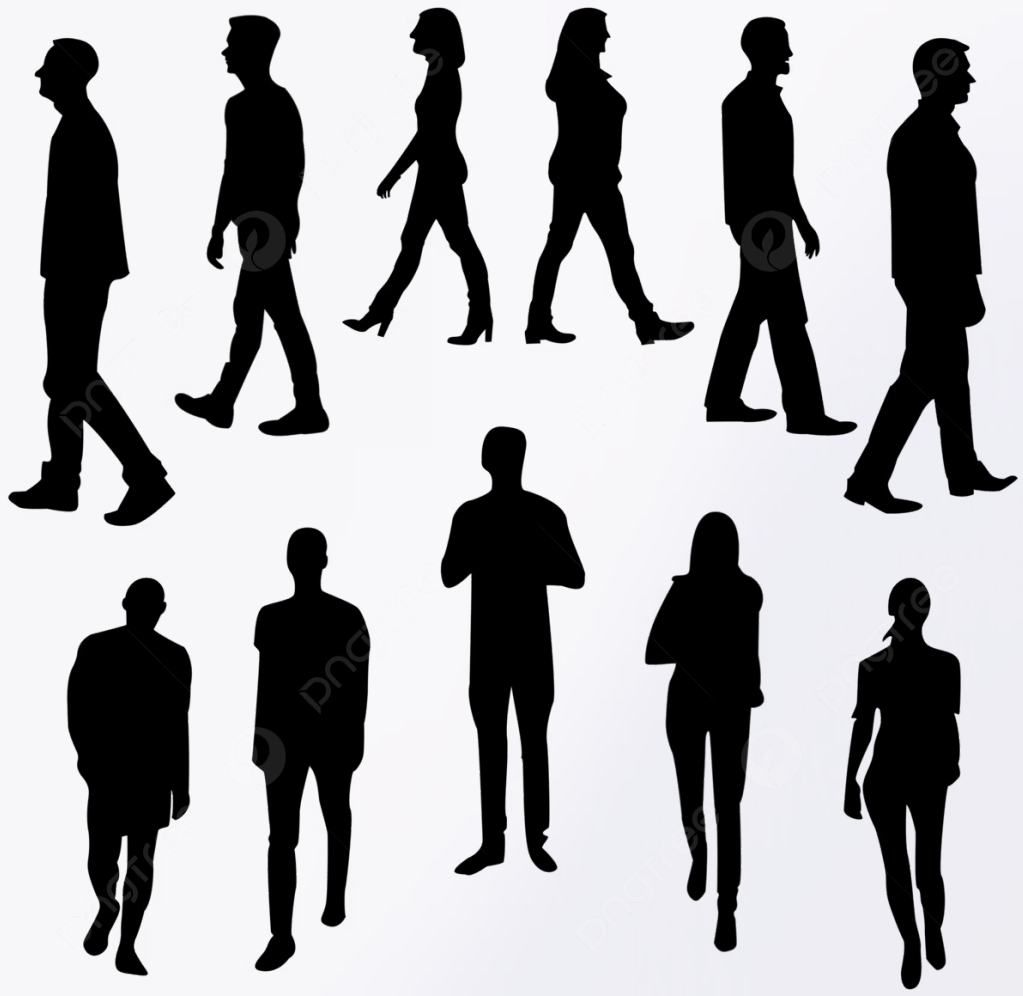
## Utility Layer

- **InputHelper:** Safe and templated input handling.
- **Address (Struct):** Data structure for location information.




## Domain Model

- **Person (Abstract):** Base class with pure virtual methods.
- **Student:** Inherits Person, manages GPA and quiz scores.
- **Professor:** Inherits Person, manages salary details.



## Repository & Services

- **UniversityRepository:** Template-based data storage.
- **FileService:** Handles all file I/O operations.

These core components form the backbone of the University Management System, ensuring modularity, scalability, and maintainability.

# OOP Concepts Demonstrated

- ✓ **Structs:** Used for Address to encapsulate location data.
- ✓ **Arrays:** Employed for managing lastQuizScores[3].
- ✓ **Pointers:** Utilized for dynamic memory allocation.
- ✓ **Classes:** Person, Student, Professor as core entities.
- ✓ **Constructors:** Default and parameterized for object initialization.
- ✓ **Destructors:** Essential for proper memory cleanup.
- ✓ **Static Members:** e.g., Student::count for tracking instances.
- ✓ **Inheritance:** Student and Professor extending Person.
- ✓ **Composition:** Person class owning an Address object.
- ✓ **Method Overriding:** Polymorphic display and save methods.
- ✓ **Method Overloading:** e.g., updateInfo with varying parameters.
- ✓ **Templates:** Generic Repository and FileService for type safety.
- ✓ **Exception Handling:** Robust validation and error management.

# Advanced Concepts - Self-Studied

### Version Control (GitHub)

Managed source code, tracked changes, and maintained a structured workflow.

### SOLID Principles Applied

Implemented Single Responsibility, Open/Closed, and Liskov Substitution principles.

### Advanced Input Validation

Developed InputHelper for custom validation, preventing runtime errors and repetitive logic.

### File Stream Operations

Utilized C++ fstream for external data persistence in .txt files, ensuring data integrity.

# Key System Features

### Safe Input Handling

Template-based validation with automatic error recovery.

### Data Persistence

Records saved to text files in a structured format.

### Memory Management

Automatic cleanup via destructors and RAII, preventing leaks.

### Polymorphism

Abstract base class design enabling runtime polymorphic behavior.

### Exception Handling

Robust validation for GPA range and file operations.

# User Interface Flow

### Main Menu

Choose between Student or Professor management.

### Student Menu

Add, View, or Save student records.

### Professor Menu

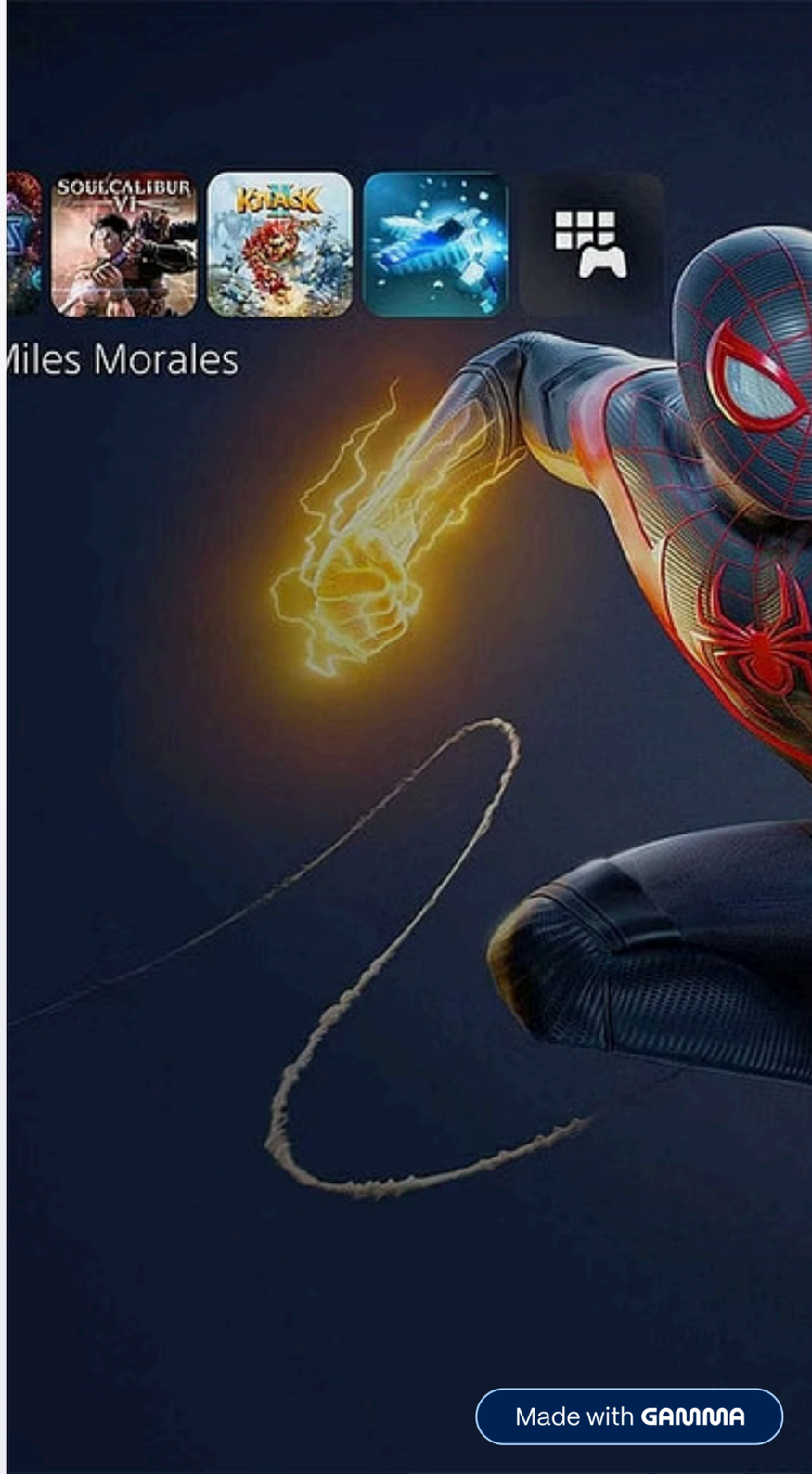
Add, View, or Save professor records.

## Key Features

- Input validation at every step.
- Comprehensive exception handling.
- Easy navigation with numbered menus.
- Clear user prompts and feedback.

## How to Use

1. Launch program → Main menu.
2. Select Student (1) or Professor (2).
3. Add records with validated input.
4. View all records on screen.
5. Save records to .txt files and exit safely.



# Code Quality Standards & Best Practices

## Code Organization

- Well-organized class separation.
- Comprehensive comments and documentation.
- Logical file structure and meaningful variable names.

## Design Patterns Applied

- Repository pattern for data management.
- Template pattern for generic operations.
- Abstract factory pattern (Person hierarchy) and RAII.

## Error Handling

- Try-catch blocks for exception handling.
- Input validation at every step with graceful recovery.
- User-friendly error messages for clarity.

# Conclusion & Demonstration

### 1 Key Achievements

Successfully implemented all required OOP concepts and created a functional, error-free system.

### 2 Advanced Application

Applied SOLID principles, design patterns, and comprehensive documentation.

### 3 Extended Learning

Self-studied advanced concepts beyond course scope, demonstrating initiative.

## Ready for Live Demonstration!

## Questions & Answers

[Contact Information Here]