# **Department of Software Engineering**

**SE-210: Software Design and Architecture**

**Class: BESE-13AB**

**Lab 05: Refactoring a Fragile C++ Codebase**

**Date: March 6, 2024**

# **Instructor: Zunera Zahid**

# Identifying and Rectifying Rotten Design: Refactoring a Fragile C++ Codebase

#include <iostream>

#include <string>

// Base class representing animals

class Animal {

protected:

std::string name;

public:

Animal(std::string name) : name(name) {}

virtual void makeSound() = 0;

void display() {

std::cout << "Name: " << name << std::endl;

}

};

// Derived class representing dogs

class Dog : public Animal {

public:

Dog(std::string name) : Animal(name) {}

void makeSound() override {

std::cout << name << " says Woof!" << std::endl;

}

};

// Derived class representing cats

class Cat : public Animal {

public:

Cat(std::string name) : Animal(name) {}

void makeSound() override {

std::cout << name << " says Meow!" << std::endl;

}

};

// Class representing a Zoo

class Zoo {

private:

Animal\* animals[100]; // Array of pointers to animals

int numAnimals; // Number of animals currently in the zoo

public:

Zoo() : numAnimals(0) {}

void addAnimal(Animal\* animal) {

animals[numAnimals++] = animal;

}

void makeSounds() {

for (int i = 0; i < numAnimals; ++i) {

animals[i]->makeSound();

}

}

};

int main() {

Zoo zoo;

Dog dog1("Buddy");

Dog dog2("Max");

Cat cat1("Whiskers");

Cat cat2("Snowball");

zoo.addAnimal(&dog1);

zoo.addAnimal(&dog2);

zoo.addAnimal(&cat1);

zoo.addAnimal(&cat2);

zoo.makeSounds();

return 0;

}

# Part 1: Code Analysis and Design Issues Identification

1. Analyze the provided C++ code implementing a Zoo management system using inheritance.
2. Identify and explain four design issues present in the code related to:

* Fragility
* Rigidity
* Viscosity
* Immobility

Refactoring Challenges:

1. Identify Fragility Issues: Look for areas in the code where changes could potentially cause unexpected breakages. Address these by decoupling classes and ensuring robust error handling.
2. Address Rigidity: Make the code more flexible by reducing tight coupling and using composition over inheritance. This allows for easier modification and extension of functionality.
3. Reduce Viscosity: Remove any obstacles preventing developers from using more efficient solutions, such as composition over inheritance. Encourage best practices and refactor where necessary to improve code maintainability.
4. Resolve Immobility: Ensure that code is modular and reusable by designing classes with clear responsibilities and dependencies. Extract common functionality into separate modules for better reuse.

# Part 2: Code Refactoring and Improvement

1. Refactor the provided code to address the identified design issues.
2. Rewrite the code using composition instead of inheritance where appropriate.
3. Ensure that the refactored code maintains functionality similar to the original code but with improved design principles.
4. Provide comments explaining the changes made and how they address the identified design issues.

# Submission Requirements:

1. Submit a document containing:

* Analysis of design issues in the original code.
* Refactored code with comments explaining the changes.

1. Ensure readability and clarity in your explanations and code comments.