Workshop Guideline: Applying SOLID Principles in a Human Simulation Video Game

Eng. Liliana Marcela Olarte, M.Sc.
Eng. Carlos Andrés Sierra, M.Sc.
Lecturers
Computer Engineering Program
School of Engineering
Universidad Nacional de Colombia

Workshop Objective:
Guide students to understand and apply the SOLID principles of object-oriented design by designing and extending a simple human simulation video game, where each human can have different professions and behaviors.

Workshop Steps and Instructions:

1. **Introduction to SOLID Principles (10 min)**
   - Briefly review the SOLID principles:
     - **S**ingle Responsibility Principle (SRP)
     - **O**pen/Closed Principle (OCP)
     - **L**iskov Substitution Principle (LSP)
     - **I**nterface Segregation Principle (ISP)
     - **D**ependency Inversion Principle (DIP)
   - Explain the exercise: Design a basic simulation of humans in a video game, where each human can have a profession (e.g., Doctor, Engineer, Artist) and perform actions. The design must allow easy extension for new professions and behaviors.
2. **Exercise Setup (10 min)**
   - **Base scenario:** There is a `Human` class with basic attributes (name, age). Each human can have a profession and perform profession-specific actions.
   - **Goal:** Apply SOLID principles to make the system extensible and maintainable.
3. **Step-by-Step Activity**

Carlos Andrés Sierra, Computer Engineer, M.Sc. in Computer Engineering, Lecturer at Universidad Nacional de Colombia.

Any comment or concern regarding this workshop can be sent to Carlos A. Sierra at: *casierrav@unal.edu.co*.

a. **Single Responsibility Principle (SRP)**
Ensure that the `Human` class only handles generic human data (name, age). Create separate classes for professions (e.g., `Doctor`, `Engineer`).
Discussion: Why is it better to separate profession logic from the human entity?

b. **Open/Closed Principle (OCP)**
Design the system so you can add new professions (e.g., `Artist`, `Teacher`) without modifying existing code. Use inheritance or composition to extend behaviors.
Discussion: How does your design allow for easy extension?

c. **Liskov Substitution Principle (LSP)**
Ensure that any subclass or profession can be used wherever a generic profession is expected. For example, a `Doctor` or `Engineer` can be assigned to a `Human` without breaking the simulation.
Discussion: What would violate LSP in your design?

d. **Interface Segregation Principle (ISP)**
If professions have unique actions (e.g., `Doctor` can `heal`, `Artist` can `paint`), define small, specific interfaces (e.g., `IHealable`, `IPaintable`). Avoid forcing all professions to implement all actions.
Discussion: How does ISP improve your code's flexibility?

e. **Dependency Inversion Principle (DIP)**
Use abstractions (interfaces or abstract classes) for professions and actions. The `Human` class should depend on abstractions, not concrete profession classes.
Discussion: How does DIP help with testing and extending your simulation?

4. **Extension Challenge (20 min)**

- **Add a new profession:** For example, add a `Musician` profession with a `playMusic()` action. Show that you can add this without changing existing code.
- **Reflection:** Which SOLID principles made this extension easy?

5. **Group Discussion and Sharing (10 min)**

- Share your designs and discuss:
  - What challenges did you face?
  - How did SOLID principles help?
  - How would you further improve your design?

6. **Wrap-Up**

- Summarize key takeaways about SOLID principles and extensible design.
- Encourage students to apply these principles in future projects.

Optional: Starter Code Example (Python)

```
class Human:
    def __init__(self, name, age, profession):
        self.name = name
        self.age = age
        self.profession = profession

    def perform_action(self):
        self.profession.perform_action()

class Profession:
    def perform_action(self):
        pass

class Doctor(Profession):
    def perform_action(self):
        print("Healing a patient.")

class Engineer(Profession):
    def perform_action(self):
        print("Building a bridge.")

# Extension: Add Musician
class Musician(Profession):
    def perform_action(self):
        print("Playing music.")

# Usage
h1 = Human("Alice", 30, Doctor())
h2 = Human("Bob", 25, Musician())
h1.perform_action()
h2.perform_action()
```