In Partial Fulfillment of the Requirements for the

CS 223 - Object-Oriented Programming

"Four Principles of Object-Oriented Programming"

Presented to:

Dr. Unife O. Cagas
Professor V

Presented by:

Jovi D. Beling
BSCS 2A2 Student

" Student and Employee"

Project Title

Project Description

This code demonstrates the concepts of abstraction, encapsulation, inheritance, polymorphism in object-oriented programming. Abstraction and encapsulation are achieved through the Person class. It serves as a base class for other classes and provides common functionality and abstraction. The constructor initializes the _name and _age attributes for encapsulation. The display_info method is an abstract method that must be implemented by subclasses. Inheritance and polymorphism are demonstrated through the Student and Teacher classes. The Student class inherits from the Person class and adds a student_id attribute. The Teacher class also inherits from the Person class and adds an employee_id attribute. Both classes override the display_info method to provide class-specific information. The display_person_info function accepts a Person object or its subclasses and calls the display_info method. This showcases polymorphism, where the same function can operate on different types of objects. In the usage section, a Student object and a Teacher object are created with specified attributes. The display_person_info function is called with both objects, demonstrating polymorphism in action.

Objectives:

1. The primary objective is to demonstrate fundamental object-oriented programming principles, such as abstraction, encapsulation, inheritance, and polymorphism, in a simple and understandable context.

2. Provide a framework where users can easily extend or modify the existing classes to suit their needs, encouraging scalability and flexibility in the codebase.

3. Show real-world examples of how these concepts can be used to model relationships and behaviors, in this case, with Person, Student, and Teacher classes.

4. Highlight how polymorphism allows functions to operate on different types of objects through a common interface, promoting code reuse and flexibility.

5. Help users grasp the design decisions behind creating base classes and derived classes, providing a clear understanding of why these structures are beneficial.

6. Demonstrate how code reuse is achieved through inheritance, and emphasize the importance of encapsulation for maintaining a clean and maintainable codebase.

7. Offer insights into how the existing code can be expanded, suggesting potential new classes or features that build on the current design.

Importance and Contribution of the Project

The project showcases fundamental object-oriented programming principles that are crucial for building modular, scalable, and maintainable software systems. By utilizing abstraction, encapsulation, inheritance, and polymorphism, the project promotes code reusability, extensibility, and flexibility in managing different types of entities (students and teachers in this case). Understanding and implementing these concepts are essential for software developers to design robust and organized code structures that facilitate easier maintenance and future enhancements.

Four Principles of Object-Oriented Programming with code

Class:

A class is a blueprint or template for creating objects. It defines the attributes (properties) and behaviors (methods) that its instances will have. In the code snippet, Person, Student, and Teacher are classes.

```
class Person:
    def __init__(self, name, age):
```

Object:

An object, also known as an instance, is a concrete realization of a class. It contains specific data and can perform the methods defined by its class. In the code snippet, student1 and teacher1 are objects created from the Student and Teacher classes, respectively.

```
student1 = Student("Alyn", 21, "S001")
teacher1 = Teacher("Mr. Smith", 40, "T001")
```

Inheritance:

Inheritance is a mechanism where a class (subclass or child class) inherits attributes and methods from another class (base class or parent class). This allows subclasses to reuse code and extend or override base class functionality. In the code snippet, Student and Teacher classes inherit from the Person class, gaining its attributes and behaviors.

```
class Student(Person):

class Teacher(Person):
```

Encapsulation:

Encapsulation restricts direct access to certain components of an object, promoting a well-defined interface for interaction. It helps protect internal state and reduces complexity. In the code snippet, encapsulation is implemented by prefixing attribute names with an underscore (e.g., _name, _age), signaling that these attributes should not be accessed directly. The Person class provides getter methods get_name() and get_age() to access these attributes.

```
class Person:
    def __init__(self, name, age):
        self._name = name
        self._age = age

    def get_name(self):
        return self._name

    def get_age(self):
        return self._age
```

Polymorphism:

Polymorphism allows different classes to be treated as if they were instances of a common base class, enabling shared behavior. This feature simplifies code and improves flexibility. In the code snippet, the display_person_info() function takes any Person object (or its subclasses) and calls the display_info() method. Although the function doesn't know whether it's dealing with a Student or Teacher, it can interact with both due to polymorphism.

```
def display_person_info(person):
    print(person.display_info())
```

Abstraction:

Abstraction is the concept of providing an interface while hiding implementation details. It allows different implementations to share the same abstract interface. In the code snippet, the Person class serves as an abstract base class with the display_info() method marked as abstract, requiring subclasses to implement their version. This design approach creates a clear interface while allowing flexibility in subclass implementations.

```
class Person:
    def display_info(self):
        raise NotImplementedError("Subclasses must implement abstract method")
```

Hardware and Software Used

Hardware:

- Laptop
- Cellphone

Software:

- Visual Studio Code
- Online GDB

Output:

```
PS C:\Users\Admin\Documents\Beling> & C:/ProgramData/anaconda3/python.exe c:/Users/Admin/Documents/Beling/OOP
Student: Alyn, Age: 21, Student ID: S001
Teacher: Mr. Smith, Age: 40, Employee ID: T001
```

Description:

The code creates a Student object student1 with the name "Alyn", age 21, and student ID "S001". It creates a Teacher object teacher1 with the name "Mr. Smith", age 40, and employee ID "T001". The display_person_info(student1) function call uses polymorphism to call the display_info() method specific to the Student class, resulting in the output: Student: Alyn, Age: 21, Student ID: S001. Similarly, display_person_info(teacher1) calls the display_info() method specific to the Teacher class, producing the output: Teacher: Mr. Smith, Age: 40, Employee ID: T001.

Code Documentation:

```
# Abstraction and encapsulation

class Person:

# The `Person` class serves as a base class for other classes, providing common functionality and abstraction.

    def __init__(self, name, age):# The constructor initializes the `_name` and `_age` attributes for encapsulation.

        self._name = name

        self._age = age


    def display_info(self):

# This is an abstract method, indicating that subclasses must implement their own version of it.

        raise NotImplementedError("Subclasses must implement abstract method")
```

```
def get_name(self):
        # Getter method to access the `name` attribute in an encapsulated manner.
        return self._name
    def get_age(self):
        # Getter method to access the `age` attribute in an encapsulated manner.
        return self._age


# Inheritance and polymorphism
class Student(Person):
    # `Student` class inherits from the `Person` class, utilizing inheritance to gain the `Person`'s functionality.
    def __init__(self, name, age, student_id):
        # The constructor initializes the attributes specific to `Student`, using `super()` to call the `Person` constructor.
        super().__init__(name, age)
        self._student_id = student_id


    def display_info(self):
        # This method implements the abstract method from the base `Person` class.
        # It returns a formatted string containing `Student`-specific information.
        return f"Student: {self._name}, Age: {self._age}, Student ID: {self._student_id}"


class Teacher(Person):
    # `Teacher` class also inherits from `Person`, demonstrating inheritance and polymorphism.
    def __init__(self, name, age, employee_id):
        # The constructor initializes `Teacher`-specific attributes and calls the base class constructor.
        super().__init__(name, age)
        self._employee_id = employee_id
```

```
  def display_info(self):
      # This method implements the abstract method, returning a formatted string with `Teacher`-specific information.
  return f"Teacher: {self._name}, Age: {self._age}, Employee ID: {self._employee_id}"


# Polymorphism
def display_person_info(person):
    # This function accepts a `Person` object or its subclasses and calls `display_info`.
    # This is an example of polymorphism, where the same function can operate on different types of objects.
    print(person.display_info())


# Usage
student1 = Student("Alyn", 21, "S001")
# Create a `Student` object with specified name, age, and student ID.

teacher1 = Teacher("Mr. Smith", 40, "T001")
# Create a `Teacher` object with specified name, age, and employee ID.

display_person_info(student1)
# Calls `display_person_info` with a `Student` object, demonstrating polymorphism.

display_person_info(teacher1)
# Calls `display_person_info` with a `Teacher` object, also demonstrating polymorphism.
```

User Guide:

1. Understanding the Classes:

- The code defines two classes: Person, which serves as a base class, and two subclasses, Student and Teacher.
- Person class encapsulates common attributes like name and age, with a method for displaying information.
- Student and Teacher inherit from Person and provide their specific implementation for displaying information.

2. Creating Instances:

- To create a Student instance, use the Student class constructor with parameters for name, age, and student ID.
- To create a Teacher instance, use the Teacher class constructor with parameters for name, age, and employee ID.

3. Displaying Information:

- Use the display_person_info function to display information about a person.
- This function accepts a Person object or its subclasses and calls the display_info method.

4. Customization (Optional):

- You can customize the display_info method in subclasses (Student and Teacher) to modify how information is displayed.
- For instance, you might want to include additional details specific to students or teachers in their respective display_info methods.

5. Understanding Polymorphism:

- Notice how the display_person_info function can accept both Student and Teacher objects without modification.
- This demonstrates polymorphism, where the same function can operate on different types of objects, as long as they inherit from the same base class (Person).

6. Extending Functionality (Optional):

- You can extend the functionality of the classes by adding more methods or attributes as per your requirements.
- For example, you might add methods to calculate grades for students or manage courses for teachers.

References:

https://www.w3schools.com/python/default.asp

https://chatgpt.com/c/221e367f-9d64-4acf-8395-0988ad5fb2b1