Ministry of Education and Research of the Republic of Moldova
Technical University of Moldova
Department of Software and Automation Engineering

REPORT

Laboratory work No. 0

Discipline: Techniques and Mechanisms of Software Design

Elaborated: Durbailo Daniel, FAF-222

Checked: asist. univ. Furdui Alexandru

Chișinău 2024

Topic: SOLID Principles

Task: Implement 2 SOLID letters in a simple project.

Theory

SOLID is an acronym for a group of five good principles (rules) in computer programming. SOLID allows programmers to write code that is easier to understand and change later on. SOLID is often used with systems that use an object-oriented design.

SOLID was promoted by Robert C. Martin but the name itself was created by Michael Feathers

SOLID Principles:

1.Single **responsibility** principle - Class has one job to do. Each change in requirements can be done by changing just one class.

2.**Open/closed** principle - Class is happy (open) to be used by others. Class is not happy (closed) to be changed by others.

3.Liskov **substitution** principle - Class can be replaced by any of its children. Children classes inherit parent's behaviours.

4.**Interface segregation** principle - When classes promise each other something, they should separate these promises (interfaces) into many small promises, so it's easier to understand.

5.Dependency inversion principle - When classes talk to each other in a very specific way, they both depend on each other to never change. Instead classes should use promises (interfaces, parents), so classes can change as long as they keep the promise.

Introduction

In this project, I implemented a simple vehicle information management system to demonstrate the practical application of two SOLID principles: Single Responsibility Principle (SRP) and Open/Closed Principle (OCP). The system manages vehicle data (make, model, and price) and supports printing vehicle details in different formats (text and JSON).

Single Responsibility Principle (SRP): Each class has a single, well-defined responsibility. The `Vehicle` class is responsible for holding vehicle-related data, such as the make, model, and price. It does not concern itself with how the information is presented. The `Printer` interface and its concrete implementations (e.g., `TextVehiclePrinter` and `JsonVehiclePrinter`) are responsible for formatting and printing vehicle details in different ways.

Open/Closed Principle (OCP): The system is open for extension but closed for modification. New formatting or output methods can be added by creating new classes that implement the `Printer` interface, such as `TextVehiclePrinter` for text format and `JsonVehiclePrinter` for JSON format. This allows adding new functionality without changing the existing code structure.

This approach ensures that the system remains flexible and easy to extend with new printing formats or additional vehicle details.

SRP IMPLEMENTATION

Product Class: The `Product` class is responsible only for managing and storing product data (name and price). It doesn't handle how the product details are displayed or processed, keeping its responsibility limited to product attributes.

```
class Vehicle:
    def __init__(self, make, model, price):
        self.make = make
        self.model = model
        self.price = price

    2 usages (2 dynamic)
    def get_make(self):
        return self.make

    2 usages (2 dynamic)
    def get_model(self):
        return self.model

    2 usages (2 dynamic)
    def get_price(self):
        return self.price
```

2. Printer Interface and Implementations: The `Printer` interface and its concrete implementations (`TextProductPrinter` and `JsonProductPrinter`) are responsible only for

formatting and printing the product details. Each printer class focuses on one task—whether it's printing in text format or JSON format.

By following SRP, each class focuses on a single concern:

- The `Product` class focuses on data handling.

- The printer classes focus on output formatting.

This separation of concerns makes the system easy to maintain and modify without affecting other parts of the code.

```
class TextVehiclePrinter(Printer):
    1 usage
    def print_vehicle(self, vehicle):
        print(f"Vehicle: {vehicle.get_make()} {vehicle.get_model()}, Price: {vehicle.get_price()} USD")

# Concrete implementation for JSON printing
1 usage
class JsonVehiclePrinter(Printer):
    1 usage
    def print_vehicle(self, vehicle):
        vehicle_data = {
            "make": vehicle.get_make(),
            "model": vehicle.get_model(),
            "price": vehicle.get_price()
        }
        print(json.dumps(vehicle_data, indent=4))
```

OCP IMPLEMENTATION

Printer Interface: The `Printer` interface defines the contract for printing a product. It abstracts the printing behavior, allowing different implementations (e.g., text, JSON) without changing the existing code.

```
class Printer(ABC):
    @abstractmethod
    def print_vehicle(self, vehicle):
        pass
```

Concrete Printer Classes: The system is closed for modification because existing classes like `Product`, `TextProductPrinter`, and `JsonProductPrinter` don't need to be altered when adding new functionality. To support a new format (for example, XML printing), you can

create a new class (`XmlProductPrinter`) that implements the `Printer` interface without changing any of the existing code.

By adhering to OCP:

- I can extend the system with new printer types (e.g., `XmlProductPrinter`, `HtmlProductPrinter`), enhancing flexibility.

- Existing classes remain unchanged, reducing the risk of introducing bugs or breaking current functionality.

This structure allows you to scale the system with minimal disruption, making it more adaptable to future requirements.

RESULTS

```
if __name__ == "__main__":
    # Create a vehicle
    car = Vehicle( make: "Tesla",  model: "Model S",  price: 85000)

    # Print vehicle details using text format
    text_printer = TextVehiclePrinter()
    text_printer.print_vehicle(car)

    # Print vehicle details using JSON format
    json_printer = JsonVehiclePrinter()
    json_printer.print_vehicle(car)
```

```
Vehicle: Tesla Model S, Price: 85000 USD
{
    "make": "Tesla",
    "model": "Model S",
    "price": 85000
}
```

CONCLUSION

By implementing both the Single Responsibility Principle and the Open/Closed Principle in this vehicle information management system, the code is modular, maintainable, and easy to extend. The SRP ensures that each class has a clear, focused responsibility, making the system easier to manage and modify. For example, the `Vehicle` class is solely responsible for handling vehicle data, while the different `Printer` classes handle how that data is presented

The OCP allows new features, such as additional formatting methods (e.g., XML or CSV), to be added without modifying the existing code. This increases flexibility and reduces the risk of introducing bugs when extending the system.

Overall, applying these SOLID principles has resulted in a clean, scalable, and future-proof design for the vehicle information management system.