Ministry of Education, Culture 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: FAF-222,
Bostan Victor

Checked: asist. univ.,
Furdui Alexandru

Chișinău 2024

Topic: SOLID Principles

Task: Implement 2 SOLID letters in a simple project.

THEORY

The SOLID principles are a set of design guidelines aimed at improving software maintainability, scalability, and robustness. They were introduced by Robert C. Martin and are widely used in object-oriented programming. The five principles are: Single Responsibility Principle (SRP), Open/Closed Principle (OCP), Liskov Substitution Principle (LSP), Interface Segregation Principle (ISP), and Dependency Inversion Principle (DIP). Each principle focuses on a specific aspect of software design, ensuring that code is flexible and resilient to change:

1. Single Responsibility Principle (SRP): A class should have only one reason to change.
2. Open/Closed Principle (OCP): Software entities should be open for extension but closed for modification.
3. Liskov Substitution Principle (LSP): Subtypes must be substitutable for their base types without altering the correctness of the system.
4. Interface Segregation Principle (ISP): Clients should not be forced to depend on interfaces they do not use.
5. Dependency Inversion Principle (DIP): High-level modules should not depend on low-level modules; both should depend on abstractions.

INTRODUCTION

In this project, I implemented a simple product management system to demonstrate the practical application of two SOLID principles: Single Responsibility Principle (SRP) and Open/Closed Principle (OCP). The system includes functionality for managing products and generating reports in different formats (text and JSON). The SRP was applied to ensure that each class has only one well-defined responsibility, and the OCP was implemented to allow new report formats to be added without altering the existing report generation logic.

SRP IMPLEMENTATION

The Single Responsibility Principle states that a class should have only one reason to change, meaning it should focus on a single responsibility. This was applied in two parts of the project: the Product class and the ProductManager class.

The Product class is responsible solely for storing product-related data, such as the product's name and price. It does not handle any business logic related to managing or processing products. By limiting the responsibility of this class to only holding data, we ensure that changes related to product representation don't affect other parts of the system.

```
3  class Product:
4      def __init__(self, name: str, price: float):
5          self.name = name
6          self.price = price
```

The ProductManager class is responsible for managing the list of products. It provides methods to add new products and retrieve the list of products. The class has no knowledge of how products are represented (as handled by the Product class) and does not concern itself with report generation. This clear separation of concerns ensures that each class follows SRP.

```
 8   class ProductManager:
 9       def __init__(self):
10           self.products = []
11
12       def add_product(self, product: Product):
13           self.products.append(product)
14
15       def get_products(self):
16           return self.products
```

OCP IMPLEMENTATION

The Open/Closed Principle dictates that classes should be open for extension but closed for modification. This means that new features or functionality can be added without modifying existing code. To implement OCP in this project, I designed a report generation system where new report formats can be added without changing the existing classes.

The ReportGenerator base class defines the structure for generating reports but does not implement the actual report generation logic. This base class includes a generate() method that is meant to be overridden by subclasses, which allows us to extend the report generation functionality by creating new classes for each report format without altering the ReportGenerator itself.

```
18  class ReportGenerator:
19      def generate(self, products):
20          raise NotImplementedError("Subclasses should implement this!")
```

To extend the report generation feature, I created two subclasses: TextReportGenerator and JSONReportGenerator. The TextReportGenerator generates a product report in plain text format, while the JSONReportGenerator generates a report in JSON format. Both of these classes extend the base ReportGenerator class and implement their own version of the generate() method. By following OCP, we can add new report formats (e.g., XML or CSV) in the future by simply creating new subclasses without altering the existing code.

```python
class TextReportGenerator(ReportGenerator):
    def generate(self, products):
        report = "Product Report (Text Format)\n"
        report += "----------------------------\n"
        for product in products:
            report += f"Product: {product.name}, Price: {product.price}\n"
        return report

class JSONReportGenerator(ReportGenerator):
    def generate(self, products):
        products_data = [{"name": product.name, "price": product.price} for product in products]
        return json.dumps(products_data, indent=4)

```

And here is an example of some results we can obtain using this system:

```python
if __name__ == "__main__":
    product1 = Product("Laptop", 1200.00)
    product2 = Product("Smartphone", 800.00)

    manager = ProductManager()
    manager.add_product(product1)
    manager.add_product(product2)

    text_report = TextReportGenerator()
    print(text_report.generate(manager.get_products()))

    json_report = JSONReportGenerator()
    print(json_report.generate(manager.get_products()))

```

```
Product Report (Text Format)
----------------------------
Product: Laptop, Price: 1200.0
Product: Smartphone, Price: 800.0

[
    {
        "name": "Laptop",
        "price": 1200.0
    },
    {
        "name": "Smartphone",
        "price": 800.0
    }
]
PS D:\TMPS_Labs>
```

CONCLUSION

By implementing both the Single Responsibility Principle and the Open/Closed Principle in this project, I ensured that the code is modular, maintainable, and easy to extend. The SRP allows each class to focus on a single task, making the system easier to understand and modify when needed. The OCP enables new features to be added (such as new report formats) without altering the existing code, which increases flexibility and reduces the risk of introducing bugs. Overall, applying these SOLID principles has resulted in a clean, scalable, and future-proof design.