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: FAF-222,
Grosu Renat

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 collection of five fundamental design guidelines that aim to improve the quality of software systems, enhancing their maintainability, scalability, and robustness. Introduced by Robert C. Martin, these principles are widely adopted in object-oriented programming and software engineering practices. Each principle addresses specific aspects of software design, ensuring that the resulting code is flexible and resilient to change. Below is an overview of the five SOLID principles:

1. **Single Responsibility Principle (SRP)**: This principle states that a class should have only one reason to change, meaning it should encapsulate a single functionality or responsibility. By adhering to SRP, developers can create classes that are more focused and easier to understand. When changes occur, they will only affect a single class, minimizing the potential for introducing bugs in other areas of the application. This clear separation of concerns promotes cleaner code architecture and enhances maintainability over time.

2. **Open/Closed Principle (OCP)**: According to this principle, software entities, such as classes and modules, should be open for extension but closed for modification. This means that existing code should not be altered to introduce new functionality. Instead, developers should add new code (e.g., new classes or methods) that extends the behavior of the existing codebase. By following the OCP, software systems become more adaptable to changing requirements, enabling easier enhancements and reducing the risk of introducing defects into stable code.

3. **Liskov Substitution Principle (LSP)**: This principle emphasizes that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program. In practical terms, this means that derived classes must honor the contract established by their base classes. Violating this principle can lead to unexpected behaviors, making code less reliable and harder to understand. Adhering to LSP encourages proper inheritance design and promotes a more predictable and robust system.

4. **Interface Segregation Principle (ISP)**: ISP posits that no client should be forced to depend on methods it does not use. In other words, interfaces should be tailored to specific clients, rather than having a single, large interface that encompasses many functionalities. This principle encourages the creation of smaller, more focused interfaces, leading to a cleaner and more maintainable codebase. By implementing

ISP, developers can ensure that classes remain relevant to their use cases without being burdened by unnecessary dependencies.

5. **Dependency Inversion Principle (DIP)**: This principle states that high-level modules should not depend on low-level modules; both should depend on abstractions. Additionally, abstractions should not depend on details; rather, details should depend on abstractions. By following DIP, developers can decouple their code, making it more flexible and easier to test. This principle encourages the use of interfaces and abstract classes, allowing for more versatile and reusable components.

In summary, the SOLID principles provide a comprehensive framework for designing robust and maintainable software systems. By incorporating these principles into software development practices, developers can create systems that are easier to understand, extend, and modify, leading to better long-term outcomes for both developers and users.

INTRODUCTION

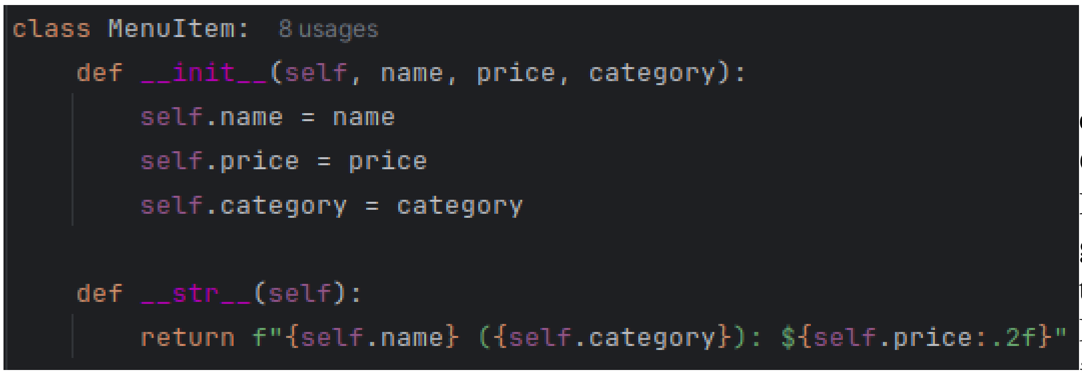
In this project, I implemented a simple restaurant menu management system to demonstrate the practical application of two SOLID principles: **Single Responsibility Principle (SRP)** and **Open/Closed Principle (OCP)**. The system includes functionalities for managing menu items, creating orders, and processing payments using different methods.

The **SRP** was applied to ensure that each class has only one well-defined responsibility, and the **OCP** was implemented to allow new payment methods to be added without altering the existing code structure.

SRP IMPLEMENTATION

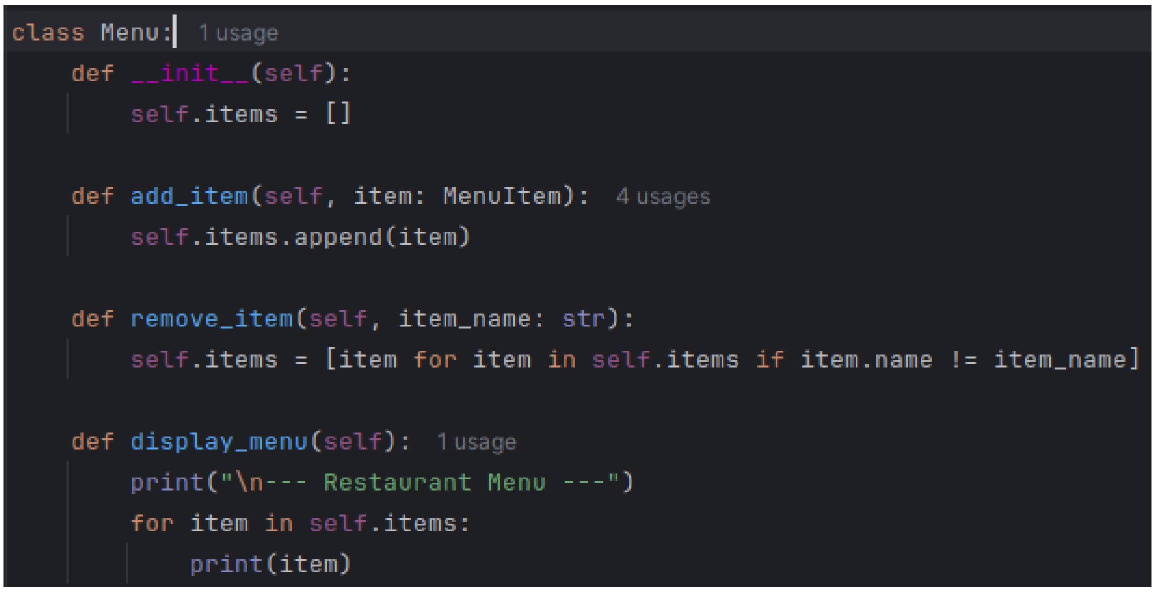
The **Single Responsibility Principle** states that a class should have only one reason to change, meaning it should focus on a single responsibility. This principle was applied in the following components of the project:

- **MenuItem Class**: Responsible for representing an individual menu item, including its name, price, and category. It focuses solely on the data related to each item.



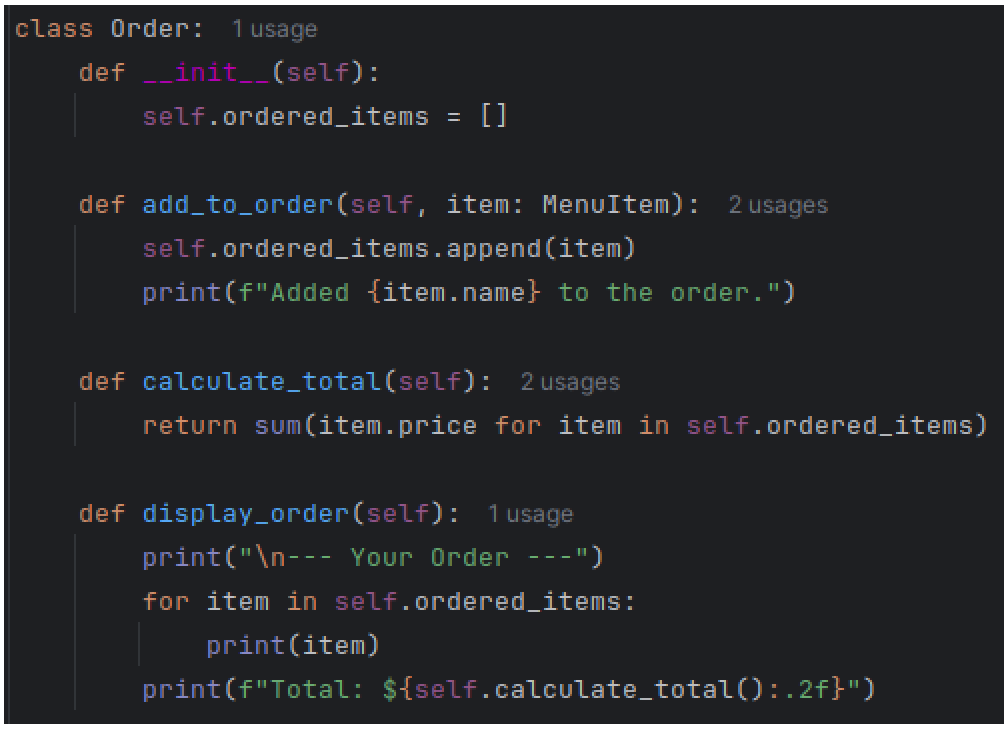
```python
class MenuItem:  8 usages
    def __init__(self, name, price, category):
        self.name = name
        self.price = price
        self.category = category

    def __str__(self):
        return f"{self.name} ({self.category}): ${self.price:.2f}"
```

- **Menu Class**: Manages the list of items available in the restaurant menu. It provides methods to add, remove, and display items without involving order or payment logic.



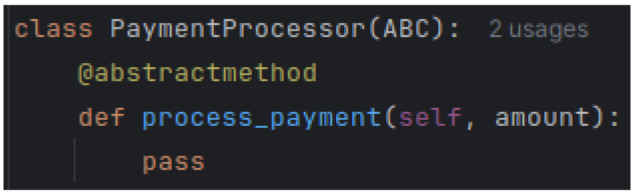
```
class Menu:  1 usage
    def __init__(self):
        self.items = []

    def add_item(self, item: MenuItem):  4 usages
        self.items.append(item)

    def remove_item(self, item_name: str):
        self.items = [item for item in self.items if item.name != item_name]

    def display_menu(self):  1 usage
        print("\n--- Restaurant Menu ---")
        for item in self.items:
            print(item)
```

- **Order Class**: Handles the items that a customer orders. It manages adding items to the order and calculating the total cost, ensuring that it does not interfere with the menu or payment functionalities.



```
class Order:  1 usage
    def __init__(self):
        self.ordered_items = []

    def add_to_order(self, item: MenuItem):  2 usages
        self.ordered_items.append(item)
        print(f"Added {item.name} to the order.")

    def calculate_total(self):  2 usages
        return sum(item.price for item in self.ordered_items)

    def display_order(self):  1 usage
        print("\n--- Your Order ---")
        for item in self.ordered_items:
            print(item)
        print(f"Total: ${self.calculate_total():.2f}")
```

- **PaymentProcessor Classes**: Separate classes are dedicated to processing payments using different methods (cash or card), focusing exclusively on handling transactions.

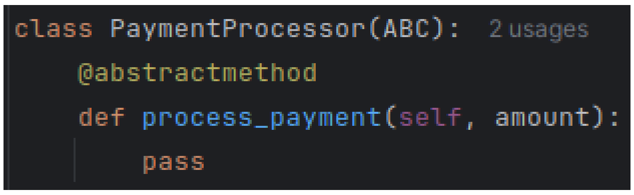


```
class PaymentProcessor(ABC):  2 usages
    @abstractmethod
    def process_payment(self, amount):
        pass
```

OCP IMPLEMENTATION

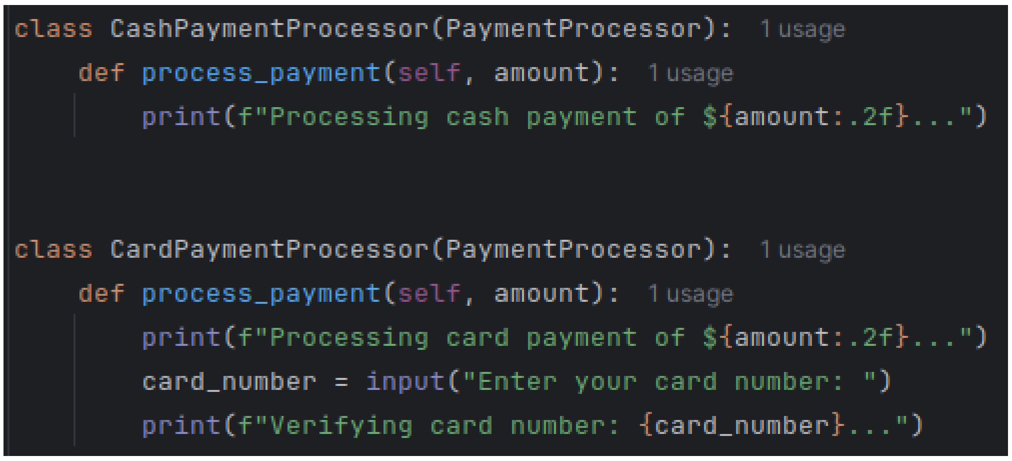
The **Open/Closed Principle** dictates that classes should be open for extension but closed for modification. To adhere to this principle, the payment processing logic was designed to allow new payment methods to be integrated without changing the existing code.

- **PaymentProcessor Abstract Class**: Defines the structure for processing payments but does not implement specific payment logic. This class serves as a base for all payment processing strategies.



```
class PaymentProcessor(ABC):  2 usages
    @abstractmethod
    def process_payment(self, amount):
        pass
```

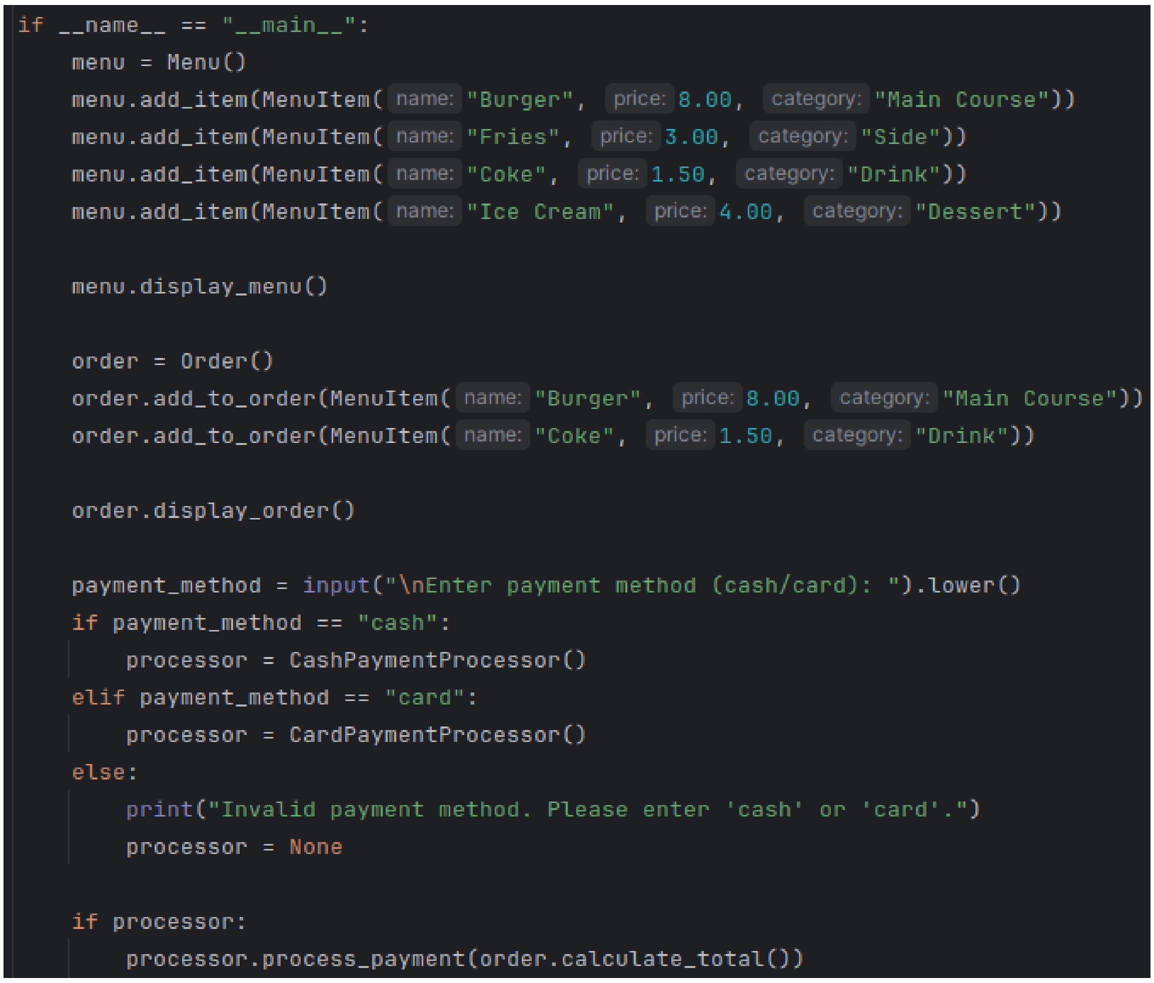
- **CashPayment Processor and CardPaymentProcessor Classes**: These classes extend the PaymentProcessor abstract class and implement their own payment logic. New payment methods, like mobile payments or digital wallets, can be added by creating new classes that inherit from the PaymentProcessor class without modifying the existing payment processors.



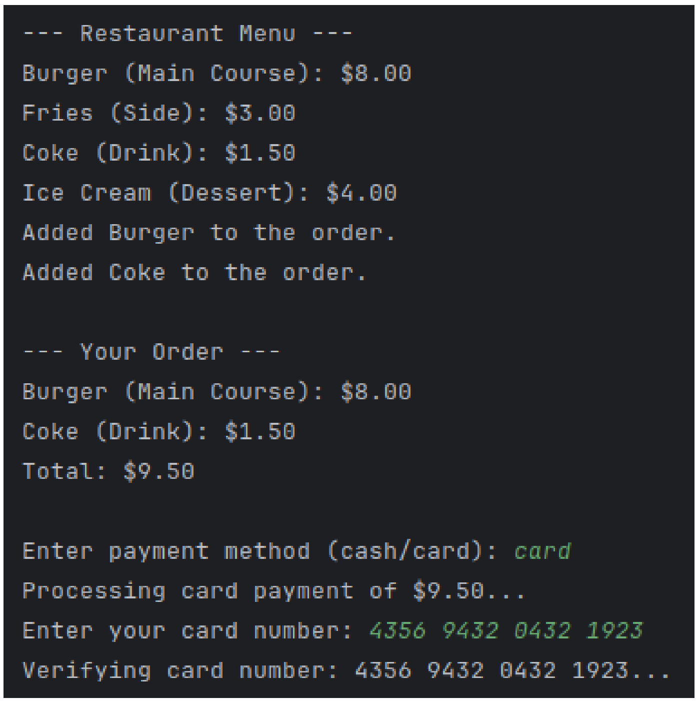
```
class CashPaymentProcessor(PaymentProcessor):  1 usage
    def process_payment(self, amount):  1 usage
        print(f"Processing cash payment of ${amount:.2f}...")


class CardPaymentProcessor(PaymentProcessor):  1 usage
    def process_payment(self, amount):  1 usage
        print(f"Processing card payment of ${amount:.2f}...")
        card_number = input("Enter your card number: ")
        print(f"Verifying card number: {card_number}...")
```

By following OCP, the system can easily accommodate new payment options in the future, enhancing its flexibility and extensibility.

RESULTS



```
if __name__ == "__main__":
    menu = Menu()
    menu.add_item(MenuItem( name: "Burger",  price: 8.00,  category: "Main Course"))
    menu.add_item(MenuItem( name: "Fries",  price: 3.00,  category: "Side"))
    menu.add_item(MenuItem( name: "Coke",  price: 1.50,  category: "Drink"))
    menu.add_item(MenuItem( name: "Ice Cream",  price: 4.00,  category: "Dessert"))

    menu.display_menu()

    order = Order()
    order.add_to_order(MenuItem( name: "Burger",  price: 8.00,  category: "Main Course"))
    order.add_to_order(MenuItem( name: "Coke",  price: 1.50,  category: "Drink"))

    order.display_order()

    payment_method = input("\nEnter payment method (cash/card): ").lower()
    if payment_method == "cash":
        processor = CashPaymentProcessor()
    elif payment_method == "card":
        processor = CardPaymentProcessor()
    else:
        print("Invalid payment method. Please enter 'cash' or 'card'.")
        processor = None

    if processor:
        processor.process_payment(order.calculate_total())
```



```
--- Restaurant Menu ---
Burger (Main Course): $8.00
Fries (Side): $3.00
Coke (Drink): $1.50
Ice Cream (Dessert): $4.00
Added Burger to the order.
Added Coke to the order.

--- Your Order ---
Burger (Main Course): $8.00
Coke (Drink): $1.50
Total: $9.50

Enter payment method (cash/card): card
Processing card payment of $9.50...
Enter your card number: 4356 9432 0432 1923
Verifying card number: 4356 9432 0432 1923...
```

CONCLUSION

By implementing both the Single Responsibility Principle and the Open/Closed Principle in this project, the code is modular, maintainable, and easy to extend. The SRP ensures that each class focuses on a single task, making the system easier to understand and modify when needed. The OCP allows new features, such as different payment methods, to be added without altering the existing code, increasing flexibility and reducing the risk of introducing bugs.

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