Pizza Place Management Project

Programming Methodologies

Alberto Charabati

7031859

2023

About the Project

The Pizza Place Management System is designed to streamline the management of pizzas and orders in a pizza place setting. Its objective is to provide an organized and efficient approach to handling different types of pizzas, such as Margherita and Pepperoni, while allowing for customizations and toppings. The system enables the creation, preparation, and delivery of pizzas, ensuring consistency and enhancing the overall customer experience.

The project leverages object-oriented programming principles and design patterns to build a flexible and scalable system. The system allows the pizza place to offer a variety of pizzas while providing the ability to add toppings and apply different eating strategies to cater to customer preferences.
The **Strategy pattern** is employed in the Pizza Place Management System to provide different offers and pricing strategies based on the customer's choice of eating in or taking away. The system incorporates the EatStrategy interface, which defines the common interface for different eating strategies. The TakeAway and EatIn classes implement this interface and provide specific implementations for calculating the cost and customisation of a pizza based on the chosen strategy.

Moreover, the **Template pattern** is utilized to define a common structure for pizza order management. The Pizza class provides a template method (PizzaOrderManagement()), creating a skeleton specifying the sequence of steps required to process an order. This ensures consistency in the order management process while allowing specific pizza types, such as Margherita or Pepperoni, to implement their unique variations of the steps, defining the overall order management structure that can be customized by different pizza types.

In addition, the **Abstract Factory pattern** is incorporated to encapsulate the creation of pizza objects. PizzaFactory, GourmetFactory and PizzaFactoryInterface serve as the factory responsible for creating specific types of pizzas, such as MargheritaPizza and PepperoniPizza. This separation of responsibilities simplifies the creation process and allows for easy extension by introducing new pizza types.

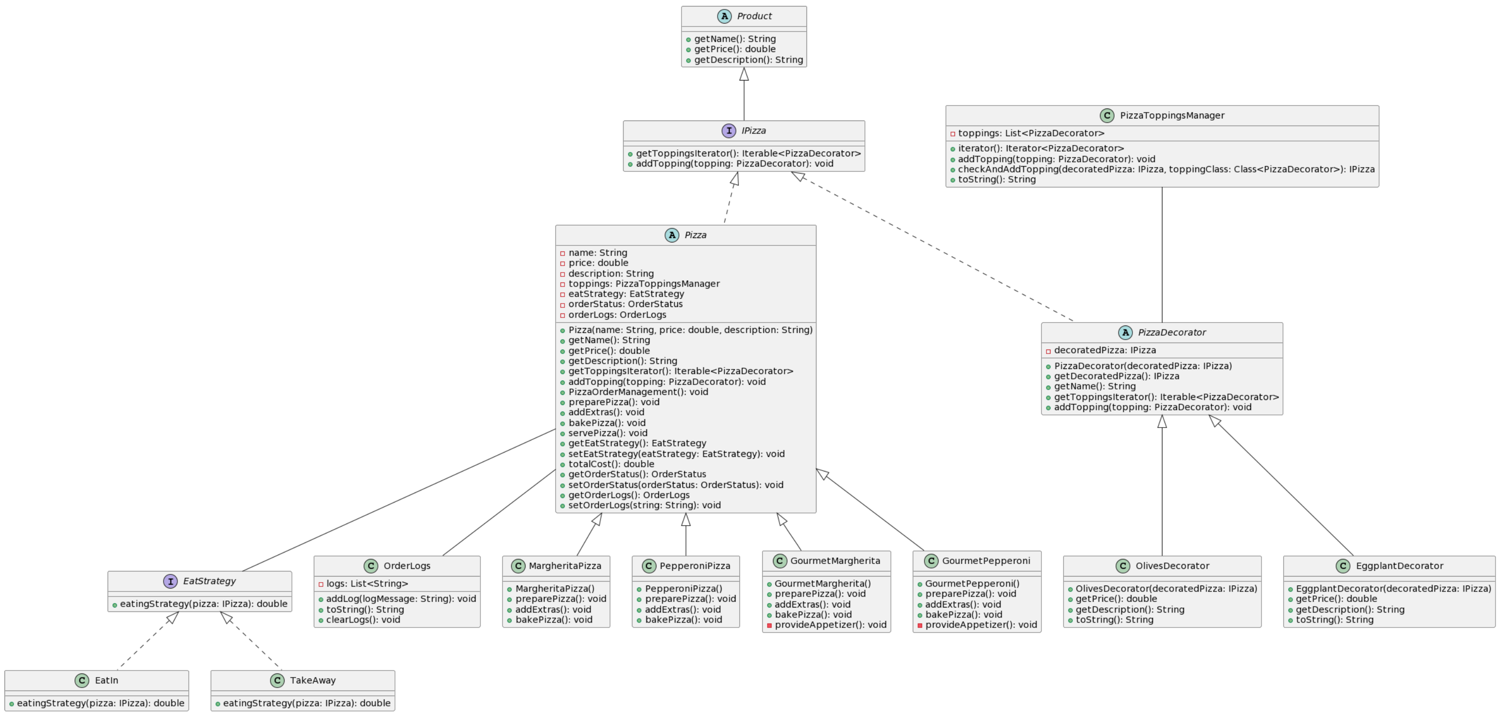
Finally, The **Decorator pattern** is implemented in the project, where pizzas serve as the base component, and decorators (toppings) can be dynamically added to enhance their flavor. This approach allows the pizza place to offer a wide range of pizza combinations without the need for numerous subclasses for each variation.

The Pizza Place project follows SOLID principles and object-oriented programming concepts. Interfaces, abstract classes, inheritance, and polymorphism are employed to achieve modularity, extensibility, and code reuse. Principles such as the Responsibility Principle is achieved by separating responsibilities into different classes, the Open-Closed Principle by allowing easy extension through subclassing, and the Dependency Inversion Principle by depending on abstractions rather than concrete implementations.

By employing these design patterns and adhering to SOLID principles, the Pizza Place Management System achieves maintainability, flexibility, and scalability, making it a robust solution for managing pizzas and orders in a pizza place environment.

Design Patterns

The project made implements the **Decorator**, **Template**, **Abstract Factory**, and **Strategy** design patterns explicitly, as well as the application of the Iterator Java library (pattern not implemented, Java Iterator used).

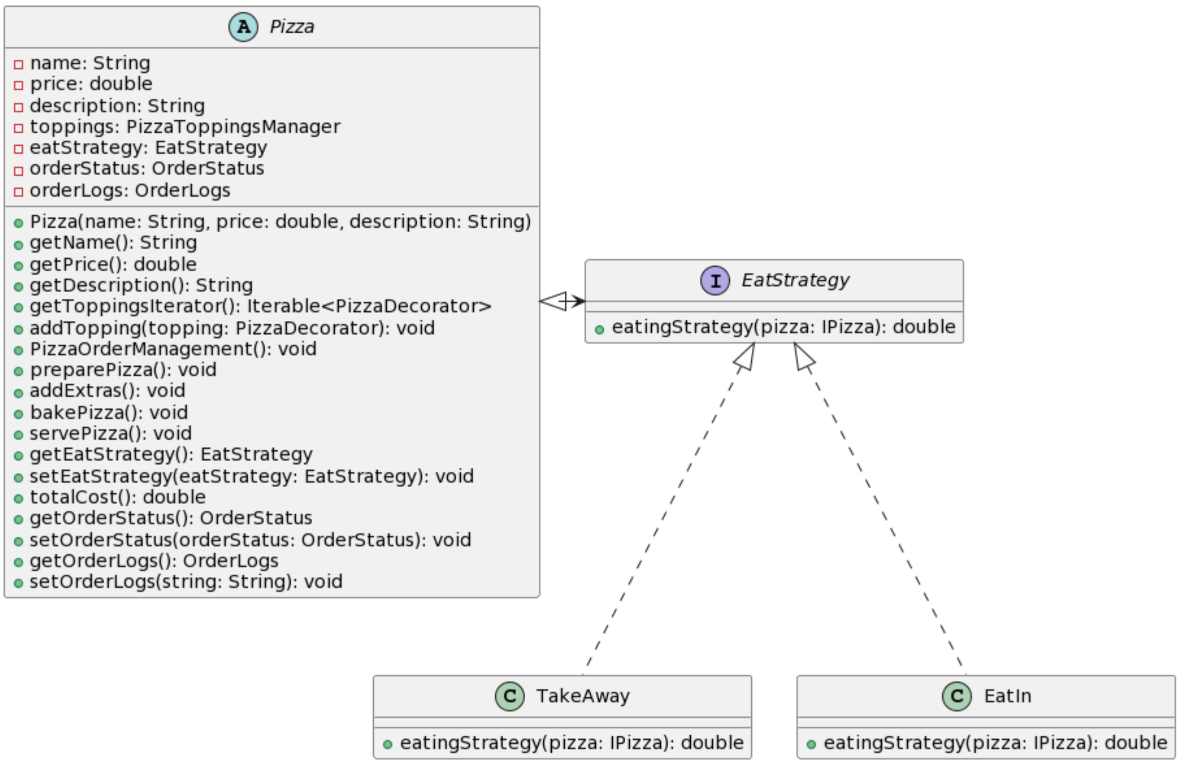


Product
getName(): String
getPrice(): double
getDescription(): String
IPizza
getToppingsIterator(): Iterable<PizzaDecorator>
addTopping(topping: PizzaDecorator): void
PizzaToppingsManager
toppings: List<PizzaDecorator>
iterator(): Iterator<PizzaDecorator>
addTopping(topping: PizzaDecorator): void
checkAndAddTopping(decoratedPizza: IPizza, toppingClass: Class<PizzaDecorator>): IPizza
toString(): String
Pizza
name: String
price: double
description: String
toppings: PizzaToppingsManager
eatStrategy: EatStrategy
orderStatus: OrderStatus
orderLogs: OrderLogs
Pizza(name: String, price: double, description: String)
getName(): String
getPrice(): double
getDescription(): String
getToppingsIterator(): Iterable<PizzaDecorator>
addTopping(topping: PizzaDecorator): void
PizzaOrderManagement(): void
preparePizza(): void
addExtras(): void
bakePizza(): void
servePizza(): void
getEatStrategy(): EatStrategy
setEatStrategy(eatStrategy: EatStrategy): void
totalCost(): double
getOrderStatus(): OrderStatus
setOrderStatus(orderStatus: OrderStatus): void
getOrderLogs(): OrderLogs
setOrderLogs(string: String): void
PizzaDecorator
decoratedPizza: IPizza
PizzaDecorator(decoratedPizza: IPizza)
getDecoratedPizza(): IPizza
getName(): String
getToppingsIterator(): Iterable<PizzaDecorator>
addTopping(topping: PizzaDecorator): void
EatStrategy
eatingStrategy(pizza: IPizza): double
OrderLogs
logs: List<String>
addLog(logMessage: String): void
toString(): String
clearLogs(): void
MargheritaPizza
MargheritaPizza()
preparePizza(): void
addExtras(): void
bakePizza(): void
PepperoniPizza
PepperoniPizza()
preparePizza(): void
addExtras(): void
bakePizza(): void
GourmetMargherita
GourmetMargherita()
preparePizza(): void
addExtras(): void
bakePizza(): void
provideAppetizer(): void
GourmetPepperoni
GourmetPepperoni()
preparePizza(): void
addExtras(): void
bakePizza(): void
provideAppetizer(): void
OlivesDecorator
OlivesDecorator(decoratedPizza: IPizza)
getPrice(): double
getDescription(): String
toString(): String
EggplantDecorator
EggplantDecorator(decoratedPizza: IPizza)
getPrice(): double
getDescription(): String
toString(): String
EatIn
eatingStrategy(pizza: IPizza): double
TakeAway
eatingStrategy(pizza: IPizza): double

Strategy Pattern:

The Strategy pattern allows the selection of an algorithm or behavior at runtime by encapsulating each algorithm or behavior into a separate strategy class. It promotes flexibility by enabling clients to choose different strategies without modifying their implementation.

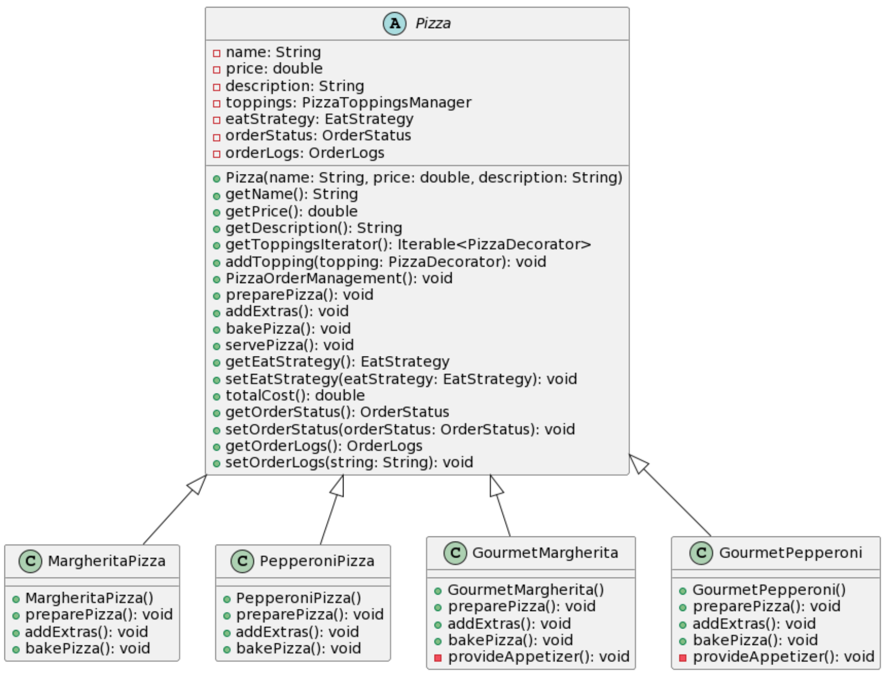
- The Strategy pattern is utilized to provide different offers to customers based on their eating preferences:
- The EatStrategy interface defines the strategy contract, with the eatingStrategy method allowing different strategies to be implemented.
- The TakeAway and EatIn classes implement the EatStrategy interface, providing strategies for customers who prefer take-away or eat-in options.
- The Pizza class has a reference to an EatStrategy object, allowing clients to select a specific strategy.
- When a client chooses the eat-in strategy, the EatIn strategy is applied, and the client gets the Olives add-on for free on their pizza order.
- When a client chooses the take-away strategy, the TakeAway strategy is applied, and the price of the pizza is discounted by 10%.


A Pizza
name: String
price: double
description: String
toppings: PizzaToppingsManager
eatStrategy: EatStrategy
orderStatus: OrderStatus
orderLogs: OrderLogs
Pizza(name: String, price: double, description: String)
getName(): String
getPrice(): double
getDescription(): String
getToppingsIterator(): Iterable<PizzaDecorator>
addTopping(topping: PizzaDecorator): void
PizzaOrderManagement(): void
preparePizza(): void
addExtras(): void
bakePizza(): void
servePizza(): void
getEatStrategy(): EatStrategy
setEatStrategy(eatStrategy: EatStrategy): void
totalCost(): double
getOrderStatus(): OrderStatus
setOrderStatus(orderStatus: OrderStatus): void
getOrderLogs(): OrderLogs
setOrderLogs(string: String): void
I EatStrategy
eatingStrategy(pizza: IPizza): double
C TakeAway
eatingStrategy(pizza: IPizza): double
C EatIn
eatingStrategy(pizza: IPizza): double

Template Method Pattern:

The Template Method pattern provides a skeletal structure for an algorithm, allowing subclasses to redefine certain steps while keeping others unchanged. It encapsulates the overall algorithm in a template method, allowing subclasses to provide their specific implementations for certain steps.

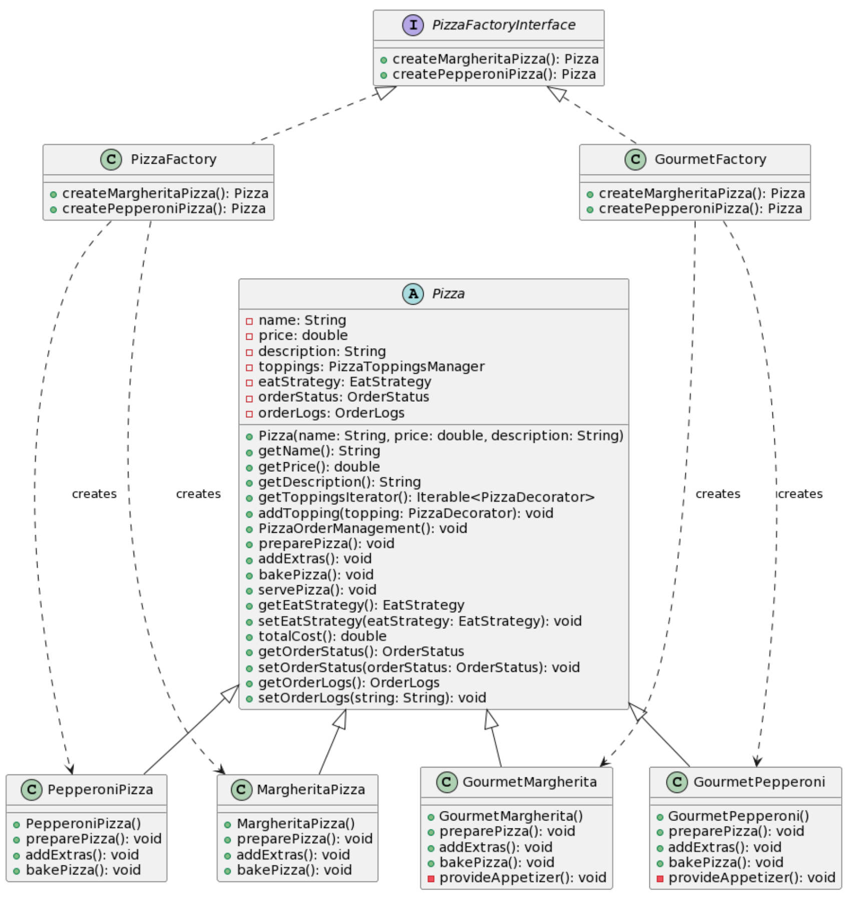
- In the project The Template Method pattern is employed for managing the pizza order process:
- The Pizza class defines the overall order management process as a template method (PizzaOrderManagement()).
- The template method outlines the common steps for ordering a pizza, including preparing, adding extras, baking, and serving.
- Concrete pizza classes (MargheritaPizza and PepperoniPizza) override the template method to provide their specific implementations for each step, based on the pizza type.
- Common steps like ServePizza() can be defined for all of the instances, while individual steps can be defined by the subclasses, while keeping the order defined by the Template.
- By using the Template Method pattern, the project ensures a consistent order management process across different pizza types while allowing flexibility for individual steps to be customized.



Abstract Factory Pattern:

The Abstract Factory pattern is utilized in the Pizza Place Management System to provide a unified interface for creating different types of pizzas without exposing the concrete implementation details. This creational pattern encapsulates the object creation logic in separate concrete factory classes, allowing the client code to create instances of pizzas through a common interface.

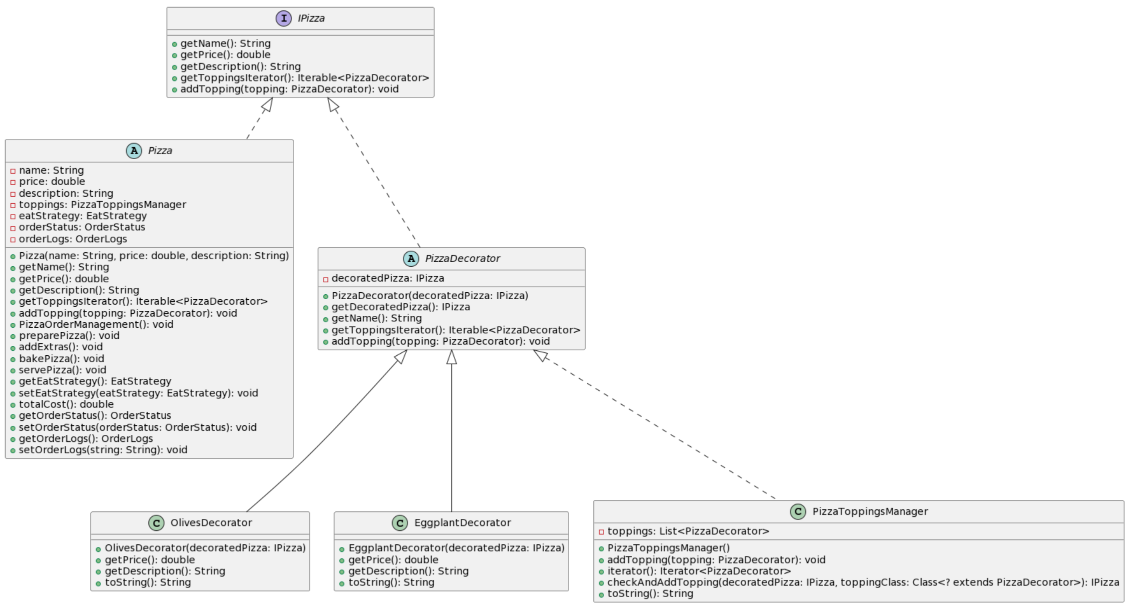
- In the project, the Abstract Factory pattern is implemented using the PizzaFactoryInterface and its concrete implementations, PizzaFactory and GourmetFactory.
- The PizzaFactoryInterface serves as the abstract factory responsible for defining the common interface for creating pizzas. It declares two methods, createMargheritaPizza() and createPepperoniPizza(), which are implemented by the concrete factory classes.
- The PizzaFactory class and the GourmetFactory class are concrete implementations of the PizzaFactoryInterface. The PizzaFactory class creates regular pizzas and the GourmetFactory class specializes in creating gourmet pizzas, implementing the same createMargheritaPizza() and createPepperoniPizza() methods, but returning instances of GourmetMargherita and GourmetPepperoni respectively. These concrete factory classes encapsulate the object creation logic for their respective pizza types, adhering to the common interface defined by the PizzaFactoryInterface.
- By utilizing the Abstract Factory pattern, the Pizza Place Management System decouples the client code from the specific implementations of pizzas and allows easy substitution of concrete factory classes without affecting the client's code. This promotes modularity, extensibility, and flexibility, as new types of pizzas can be added by introducing new concrete factory classes implementing the PizzaFactoryInterface, without modifying the existing client code.
- Overall, the Abstract Factory pattern enhances the maintainability and scalability of the Pizza Place Management System by providing a unified and flexible approach to creating different types of pizzas.



Decorator Design Pattern:

The Decorator pattern is applied in the Pizza Place Management System to enhance the functionality of pizzas dynamically, without modifying their core structure. This structural pattern allows for the addition of new toppings or behaviors to pizzas at runtime. It promotes flexibility and extensibility by utilizing a decorator class that wraps the original pizza object and provides additional functionality.

- In the project, the PizzaDecorator class serves as the base decorator, implementing the IPizza interface. It maintains a reference to the decorated pizza object and provides methods to modify or extend its behavior. Concrete decorators such as OlivesDecorator and EggplantDecorator extend the PizzaDecorator class, adding specific toppings to the pizzas.
- Each decorator wraps an instance of the IPizza interface, allowing for the dynamic addition of toppings to the base pizza. The decorators can be stacked and chained, enabling the customization of pizzas with multiple toppings.
- By employing the Decorator pattern, the Pizza Place Management System avoids the need for creating multiple subclasses for each pizza variation or topping combination. It promotes code reusability, modularity, and flexibility, as new toppings can be easily added without affecting the existing pizza classes. This pattern allows the pizza place to offer a wide range of pizza combinations and customization options, enhancing the customer experience and satisfying various preferences.



IPizza
getName(): String
getPrice(): double
getDescription(): String
getToppingsIterator(): Iterable<PizzaDecorator>
addTopping(topping: PizzaDecorator): void
Pizza
name: String
price: double
description: String
toppings: PizzaToppingsManager
eatStrategy: EatStrategy
orderStatus: OrderStatus
orderLogs: OrderLogs
Pizza(name: String, price: double, description: String)
getName(): String
getPrice(): double
getDescription(): String
getToppingsIterator(): Iterable<PizzaDecorator>
addTopping(topping: PizzaDecorator): void
PizzaOrderManagement(): void
preparePizza(): void
addExtras(): void
bakePizza(): void
servePizza(): void
getEatStrategy(): EatStrategy
setEatStrategy(eatStrategy: EatStrategy): void
totalCost(): double
getOrderStatus(): OrderStatus
setOrderStatus(orderStatus: OrderStatus): void
getOrderLogs(): OrderLogs
setOrderLogs(string: String): void
PizzaDecorator
decoratedPizza: IPizza
PizzaDecorator(decoratedPizza: IPizza)
getDecoratedPizza(): IPizza
getName(): String
getToppingsIterator(): Iterable<PizzaDecorator>
addTopping(topping: PizzaDecorator): void
OlivesDecorator
OlivesDecorator(decoratedPizza: IPizza)
getPrice(): double
getDescription(): String
toString(): String
EggplantDecorator
EggplantDecorator(decoratedPizza: IPizza)
getPrice(): double
getDescription(): String
toString(): String
PizzaToppingsManager
toppings: List<PizzaDecorator>
PizzaToppingsManager()
addTopping(topping: PizzaDecorator): void
iterator(): Iterator<PizzaDecorator>
checkAndAddTopping(decoratedPizza: IPizza, toppingClass: Class<? extends PizzaDecorator>): IPizza
toString(): String

SOLID

- **Single Responsibility Principle (SRP):**
- The Single Responsibility Principle states that a class should have only one reason to change, meaning that it should have only one responsibility or job.
    - The project demonstrates the SRP by dividing responsibilities among different classes:
    - Each concrete pizza class (MargheritaPizza, PepperoniPizza, GourmetMargherita, GourmetPepperoni) has the responsibility of defining its specific pizza type and implementing the steps of preparing, adding extras, baking, and serving that pizza type.
    - The PizzaFactory class is responsible for creating different types of pizzas and acts as a factory for pizza objects.
    - The PizzaToppingsManager class is responsible for managing the toppings of a pizza, preventing duplicate toppings, and providing an iterable interface for accessing the toppings.
    - The OrderLogs class is responsible for managing the logs of an order and provides methods for adding logs, generating a string representation of the logs, and clearing the logs.

- **Open-Closed Principle (OCP):**
- The Open-Closed Principle states that entities (classes, modules, functions, etc.) should be open for extension but closed for modification. It promotes the idea that existing code should not be modified when new features or behaviors are added; instead, the code should be extended.
    - The project demonstrates the OCP by providing extensibility through inheritance and composition:
    - The Pizza class defines the common order management process as a template method but allows concrete pizza classes to provide their own implementations for specific steps, such as preparing, adding extras, and baking.
    - The PizzaDecorator abstract class allows additional behavior to be added to pizzas dynamically by wrapping them with decorator objects. New toppings can be added without modifying the existing pizza classes.
    - The EatStrategy interface allows different eating strategies to be defined separately from the pizza class. New strategies can be added by implementing the interface without modifying the existing pizza classes.

- **Liskov Substitution Principle (LSP):**
- The Liskov Substitution Principle states that objects of a superclass should be substitutable by objects of its subclasses without affecting the correctness of the program. It ensures that the behavior and contracts of the superclass are preserved in its subclasses.
    - The project adheres to the LSP by maintaining substitutability and preserving contracts:
    - The Pizza class serves as the base class for different pizza types (MargheritaPizza, PepperoniPizza, GourmetMargherita, GourmetPepperoni). Subclasses can be used interchangeably with the base class, preserving the common behavior defined in the Pizza class.
    - Decorator classes (OlivesDecorator and EggplantDecorator) extend the PizzaDecorator abstract class and can be substituted for it without affecting the core behavior of decorating pizzas.
    - The EatStrategy interface defines a contract for different eating strategies, and concrete strategy classes (TakeAway and EatIn) can be used interchangeably based on the chosen strategy.

- **Interface Segregation Principle (ISP):**
- The Interface Segregation Principle states that clients should not be forced to depend on interfaces they do not use. It promotes the idea of segregating interfaces into smaller and more specific ones, tailored to the needs of the clients.
    - The project adheres to the ISP by providing cohesive and specific interfaces:
    - The Product interface defines methods related to basic product information (getName, getPrice, getDescription).
    - The IPizza interface extends the Product interface and adds methods specific to pizzas (getToppingsIterator, addTopping).
    - The PizzaFactoryInterface interface provides methods specifically for pizza creation (createMargheritaPizza, createPepperoniPizza).
    - By segregating interfaces based on specific responsibilities, clients can depend on the interfaces they need, avoiding unnecessary dependencies.

- **Dependency Inversion Principle (DIP):**
- The Dependency Inversion Principle states that high-level modules should not depend on low-level modules; both should depend on abstractions. It promotes the idea of depending on abstractions rather than concrete implementations, allowing flexibility and ease of change.
    - The project follows the DIP through the following practices:
    - The Pizza class depends on abstractions such as the EatStrategy interface for handling different eating strategies, allowing different

strategies to be plugged into the Pizza class without modifying its implementation.

- The PizzaFactory class depends on the Pizza abstract class rather than concrete pizza classes, adhering to the principle of depending on abstractions.
- The decorator pattern allows pizzas to be decorated with toppings by depending on the PizzaDecorator abstract class, which is an abstraction of a decorator, rather than depending on concrete decorator implementations.
- By relying on abstractions, the project achieves loose coupling and allows for easy substitution of implementations without affecting the overall functionality.

In summary, the project demonstrates the application of the SOLID principles by adhering to the Single Responsibility Principle, Open-Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, and Dependency Inversion Principle. These principles contribute to the project's modularity, extensibility, maintainability, and flexibility by promoting separation of concerns, enabling easy extension without modification, ensuring substitutability, providing cohesive interfaces, and depending on abstractions rather than concrete implementations.

OOP

- **Encapsulation:**
- Encapsulation is the principle of bundling data and the methods that operate on that data into a single unit, known as a class. It provides control over access to the internal state of an object and ensures that object interactions occur through well-defined interfaces.
    - The Pizza class encapsulates the state and behavior related to a pizza, including its name, price, description, toppings, order status, order logs, and the steps for preparing, adding extras, baking, and serving the pizza.
    - The PizzaDecorator class encapsulates the behavior of adding toppings to pizzas by wrapping them and modifying their price and description.
    - The OrderLogs class encapsulates the functionality for managing order logs, such as adding logs, generating a string representation of logs, and clearing logs.

- **Inheritance:**
- Inheritance is a mechanism that allows classes to inherit properties and behaviors from a base class. It promotes code reuse, extensibility, and the concept of an "is-a" relationship between classes.
    - The Pizza class serves as the base class for specific pizza types (MargheritaPizza, PepperoniPizza, GourmetMargherita, GourmetPepperoni). The subclasses inherit the common behavior defined in the Pizza class while allowing customization of specific steps.
    - The PizzaDecorator class extends the Pizza class to provide a base decorator behavior for adding toppings to pizzas. Concrete decorator classes (OlivesDecorator and EggplantDecorator) further extend the decorator behavior.

- **Polymorphism:**
- Polymorphism is the ability of objects of different classes to be treated as objects of a common superclass. It allows methods to be overridden in subclasses, providing different implementations based on the specific class type.
    - The Pizza class defines abstract methods for preparing, adding extras, and baking pizzas. Concrete pizza classes override these methods to provide their own implementations.
    - The PizzaDecorator class overrides methods such as getPrice, getDescription, and getToppingsIterator to modify the behavior of decorated pizzas.

    - The OrderLogs class overrides the toString method to generate a string representation of the order logs.

- **Abstraction:**
- Abstraction is the process of identifying essential features and behaviors while hiding unnecessary details. It allows the creation of abstract classes and interfaces that define common behavior without specifying implementation details.
    - The Pizza class defines an abstract template for the order management process, specifying common steps and leaving specific implementations to subclasses.
    - The PizzaDecorator class acts as an abstraction for adding toppings to pizzas, providing a common interface for decorators to wrap pizzas.
    - The EatStrategy interface defines an abstraction for different eating strategies, allowing pizzas to switch between strategies without coupling to specific implementations.
    - The Product interface defines abstraction for common product properties (getName, getPrice, getDescription), which are inherited by the Pizza class.

- **Association:**
- Association represents a relationship between two or more objects, where objects may interact or have a dependency on each other. It allows for code reuse, modular design, and collaboration between objects.
    - The PizzaFactory class associates with the Pizza class to create instances of specific pizza types.
    - The PizzaDecorator class associates with the Pizza class to wrap pizzas and add toppings dynamically.
    - The PizzaToppingsManager class associates with the Pizza class to manage the toppings of a pizza and prevent duplicates.

In summary, the project demonstrates the application of key Object-Oriented Programming (OOP) principles, including encapsulation, inheritance, polymorphism, abstraction, and association. These principles contribute to code organization, code reuse, modularity, extensibility, and flexibility in the project. They facilitate better design, maintainability, and scalability by promoting separation of concerns, code reuse, and clear object interactions.