PROPOSED PATTERN

Many design patterns are applied for solving existing problems in the project. These design patterns are Command Pattern, Simple Factory Pattern, Template Method Pattern, Strategy Pattern and Decorator Pattern.

Firstly, customers create a waffle (Waffle Order) or beverage (Beverage Order) order and the waitress (Invoker) delivers these orders to the cook (Receiver). The cook works with Waffle Factory for waffle and with Beverage Factory for beverage. To do this, the cook uses makeWaffle() and makeBeverage() methods (Simple Factory Pattern).

Waffle Factory creates waffles and to do this create Waffle interface. There are two kinds of waffles such as Circular and Square in Waffle Shop and these waffle types implement Waffle interface. The customers had chosen waffle ingredients according to the waffle they had previously wanted. The waffle is decorated according to these waffle ingredients (Decorator Pattern). There are 3 Decorator classes for waffle decoration such as Chocolate Decorator, Fruit Decorator and Condiment Decorator. Cost of the waffle is calculated according to decoration ingredients.

Finally, Beverage Factory is responsible for hot and cold beverages creation and creates Beverage interface. Hot Beverage and Cold Beverage classes implement this interface. Subclasses such as tea, coffee, cola etc. extend these classes. The distinction between them is the difference in the preparation of hot and cold beverages (Template Method Pattern). Also, there are some differences in subclasses. Finally, hook() is used for each beverage and the customer is asked if some extra condiments are wanted and then Strategy pattern is used for the mode of payment done by the customer.

After applying the mentioned design patterns in the project, several SOLID principles are likely to have been adhered to:

1. **Single Responsibility Principle (SRP):**
- Each class or module in your design should have only one reason to change. For example:
- The WaffleFactory and BeverageFactory classes each have a single responsibility: creating instances of their respective products (waffles and beverages).
- The Command classes (representing orders) have the responsibility of encapsulating a request as an object, thereby separating the command from its invocation.

2. **Open/Closed Principle (OCP):**
- Classes should be open for extension but closed for modification. This principle is evident in the use of the decorator pattern for extending the behavior of objects without altering their structure.
- For example, adding new types of waffle decorations or beverage condiments can be done without modifying existing code by creating new decorator classes.

3. **Liskov Substitution Principle (LSP):**
- Objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program. This principle is generally followed in inheritance relationships.
- In your design, subclasses of Waffle (e.g., CircularWaffle, SquareWaffle) and subclasses of Beverage (e.g., Tea, Coffee) should adhere to this principle, ensuring that they can be used interchangeably where the base class is expected.

4. **Interface Segregation Principle (ISP):**
- Clients should not be forced to depend on interfaces they don't use. This principle is often observed in the use of interfaces in your design:
- Waffle and Beverage interfaces are kept focused on a specific set of methods related to waffle and beverage creation, respectively.

- Clients can depend on these interfaces without being affected by the details of specific implementations.

5. **Dependency Inversion Principle (DIP):**
- High-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details; details should depend on abstractions.
- This principle is demonstrated in your design through the use of interfaces and factories:
- High-level modules (e.g., Cook, Waitress) depend on abstractions (e.g., Waffle, Beverage interfaces) rather than concrete implementations.
- Low-level modules (e.g., WaffleFactory, BeverageFactory) provide implementations that adhere to these abstractions.

By adhering to these SOLID principles, design is likely to be more maintainable, extensible, and loosely coupled, making it easier to manage and evolve over time. After applying the mentioned design patterns in the project, several SOLID principles are likely to have been adhered to:

1. **Single Responsibility Principle (SRP):**
- Each class or module in your design should have only one reason to change. For example:
- The WaffleFactory and BeverageFactory classes each have a single responsibility: creating instances of their respective products (waffles and beverages).
- The Command classes (representing orders) have the responsibility of encapsulating a request as an object, thereby separating the command from its invocation.

2. **Open/Closed Principle (OCP):**
- Classes should be open for extension but closed for modification. This principle is evident in the use of the decorator pattern for extending the behavior of objects without altering their structure.

- For example, adding new types of waffle decorations or beverage condiments can be

done without modifying existing code by creating new decorator classes.

3. **Liskov Substitution Principle (LSP):**
- Objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program. This principle is generally followed in inheritance relationships.
- In your design, subclasses of Waffle (e.g., CircularWaffle, SquareWaffle) and subclasses of Beverage (e.g., Tea, Coffee) should adhere to this principle, ensuring that they can be used interchangeably where the base class is expected.

4. **Interface Segregation Principle (ISP):**
- Clients should not be forced to depend on interfaces they don't use. This principle is often observed in the use of interfaces in your design:
- Waffle and Beverage interfaces are kept focused on a specific set of methods related to waffle and beverage creation, respectively.
- Clients can depend on these interfaces without being affected by the details of specific implementations.

5. **Dependency Inversion Principle (DIP):**
- High-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details; details should depend on abstractions.
- This principle is demonstrated in your design through the use of interfaces and factories:
- High-level modules (e.g., Cook, Waitress) depend on abstractions (e.g., Waffle, Beverage interfaces) rather than concrete implementations.
- Low-level modules (e.g., WaffleFactory, BeverageFactory) provide implementations that adhere to these abstractions.

By adhering to these SOLID principles, design is likely to be more maintainable, extensible, and loosely coupled, making it easier to manage and evolve over time.