Паттерн "цепочка обязанностей" (Chain of Responsibility) предназначен для организации обработки запроса последовательным прохождением через цепочку обработчиков. Каждый обработчик принимает решение о том, может ли он обработать запрос, и передает его следующему обработчику в цепочке, если сам не способен выполнить обработку.

Проблема, которую решает этот паттерн, заключается в том, чтобы избежать жесткой зависимости между отправителем запроса и объектами, способными его обработать. Вместо этого, объекты обрабатывают запрос последовательно, и каждый из них решает, следует ли ему обрабатывать запрос или передать его дальше по цепочке.

Преимущества использования паттерна "цепочка обязанностей" включают:

1. Разделение обязанностей: Обработчики в цепочке могут быть легко добавлены или удалены без изменения кода отправителя запроса. Это способствует более гибкой архитектуре.
2. Повышение поддержки открытости/закрытости: Вы можете добавлять новые обработчики без изменения существующего кода. Это поддерживает принцип открытости/закрытости, описанный в принципах SOLID.
3. Уменьшение связанности: Отправитель запроса не знает, какой конкретный обработчик в конечном итоге обработает запрос. Это позволяет изменять или переиспользовать части системы независимо друг от друга.

Пример иллюстрирует использование паттерна "цепочка обязанностей" для обработки различных методов оплаты при покупке в интернете. Каждый обработчик решает, может ли он обработать запрос оплаты, и, если нет, передает его следующему обработчику в цепочке.

<https://metanit.com/sharp/patterns/3.7.php>

**Цепочка Обязанностей (Chain of responsibility) -** поведенческий шаблон проектирования, который позволяет избежать жесткой привязки отправителя запроса к получателю. Все возможные обработчики запроса образуют цепочку, а сам запрос перемещается по этой цепочке. Каждый объект в этой цепочке при получении запроса выбирает, либо закончить обработку запроса, либо передать запрос на обработку следующему по цепочке объекту.

Класс Receiver с помощью конструктора и передаваемых в него параметров устанавливает возможные используемые системы платежей. При осуществлении платежа каждый отдельный объект PaymentHandler будет проверять установку у получателя определенного типа платежей. И если произойдет сопоставление типа платежей у получателя объекту PaymentHandler, то данный объект выполняет платеж. Если же необходимого способа платежей не будет определено, то деньги остаются в системе.

При этом преимуществом цепочки является и то, что она позволяет расположить последовательность объектов-обработчиков в ней в зависимости от их приоритета.

SOLID — это акроним, обозначающий пять основных принципов объектно-ориентированного программирования и проектирования. Эти принципы были введены Робертом Мартином и представляют собой руководство для создания гибких, поддерживаемых и расширяемых систем. Вот краткое описание каждого из принципов SOLID:

1. Принцип единственной ответственности (Single Responsibility Principle - SRP)

Каждый класс должен иметь только одну причину для изменения. Это означает, что класс должен выполнять только одну задачу или иметь только одну ответственность.

2. Принцип открытости/закрытости (Open/Closed Principle – OCP)

Программные сущности (классы, модули, функции и т. д.) должны быть открыты для расширения, но закрыты для модификации. Новый функционал должен добавляться без изменения существующего кода.

3. Принцип подстановки Барбары Лисков (Liskov Substitution Principle – LSP)

Объекты базового класса должны быть заменяемыми объектами производного класса без изменения корректности программы. Это означает, что любый экземпляр базового класса должен быть способен заменить экземпляр производного класса без нарушения работы программы.

4. Принцип разделения интерфейса (Interface Segregation Principle - ISP)

Клиенты не должны зависеть от интерфейсов, которые они не используют. Этот принцип подчеркивает, что интерфейсы должны быть небольшими, специфичными и соответствовать потребностям клиентов.

5. Принцип инверсии зависимостей (Dependency Inversion Principle – DIP)

Зависимости должны строиться на абстракциях, а не на деталях. Модули верхнего уровня не должны зависеть от модулей нижнего уровня. Оба типа модулей должны зависеть от абстракций.