**Паттерны проектирования**

**Паттерн проектирования Singleton (Одиночка)**

**Цель паттерна**

Singleton (Одиночка) используется для обеспечения существования только одного экземпляра определённого класса в приложении. Паттерн также предоставляет глобальную точку доступа к этому экземпляру.

**Основные задачи:**

Контролировать создание экземпляра, чтобы гарантировать его уникальность.

Управлять доступом к общему ресурсу или состоянию.

Предотвратить создание дополнительных экземпляров.

**Причины возникновения паттерна**

Проблемы с множественными экземплярами: В некоторых случаях, например, при работе с ресурсами системы (база данных, файловая система, сетевые соединения), создание нескольких экземпляров может привести к конфликтам, неэффективному использованию ресурсов или непредсказуемым ошибкам.

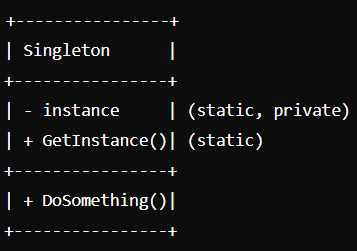
Нужна единая точка управления: Для управления конфигурацией, логированием или состоянием приложения требуется доступ из разных частей программы к единому объекту.

**Структура паттерна**

Приватный конструктор: Запрещает создание экземпляров вне класса.

Приватное статическое поле: Содержит ссылку на единственный экземпляр класса.

Публичный статический метод (или свойство): Предоставляет доступ к экземпляру и создаёт его при необходимости.



**Результаты использования паттерна**

**Преимущества:**

Гарантирует наличие единственного экземпляра.

Легко обеспечивает глобальный доступ к объекту.

Упрощает управление общими ресурсами.

**Недостатки:**

Нарушает принцип единственной ответственности (SRP), так как класс контролирует и создание экземпляра, и его поведение.

Сложность тестирования из-за скрытых зависимостей.

Возможны проблемы с многопоточностью, если не предусмотрена безопасная реализация.

**Практический пример использования паттерна**

**Пример: Логирование**

В системах логирования важно, чтобы все сообщения записывались в единый файл. Использование нескольких экземпляров логгера могло бы привести к конфликтам при записи. **(Пример 1).**

**Особенности реализации:**

Потокобезопасность: Использование lock предотвращает создание нескольких экземпляров в многопоточной среде.

Ленивая инициализация: Объект создаётся только при первом запросе.

Использование в реальных приложениях

Настройки приложения: Хранение конфигураций, доступных для всех компонентов.

База данных: Управление соединением к базе данных (Connection Pool).

Файловая система: Управление доступом к единственному ресурсу (например, к файлу).

**Паттерн проектирования Factory Method (Фабричный метод)**

**Цель паттерна**

Factory Method позволяет делегировать создание объектов подклассам, обеспечивая гибкость и расширяемость.

**Паттерн используется, когда:**

Невозможно заранее определить конкретный тип создаваемого объекта.

Нужно упростить создание объектов для клиента.

Требуется избежать жёсткой зависимости от конкретных классов.

**Причины возникновения паттерна**

Упрощение создания объектов: Если логика создания объектов сложна или часто меняется, её следует изолировать.

Принцип открытости/закрытости (OCP): Легче добавлять новые типы объектов, не изменяя существующий код.

Устранение жёсткой привязки: Клиенты не должны зависеть от конкретных классов, чтобы быть более гибкими и переиспользуемыми.

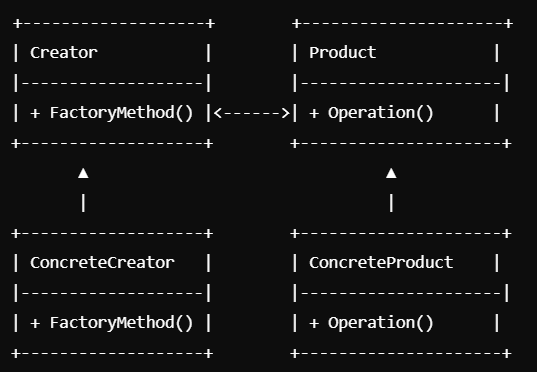
**Структура паттерна**

Абстрактный продукт: Определяет интерфейс для объектов, которые будут создаваться.

Конкретные продукты: Реализуют абстрактный интерфейс.

Создатель (Factory): Содержит фабричный метод для создания объектов.

Конкретные создатели: Реализуют фабричный метод, возвращая конкретные объекты.



**Результаты использования паттерна**

**Преимущества:**

Уменьшает зависимость клиента от конкретных классов.

Обеспечивает расширяемость: можно добавлять новые продукты, не меняя код клиента.

Инкапсулирует логику создания объектов.

**Недостатки:**

Увеличивает сложность из-за создания дополнительных классов.

Может привести к избыточной абстракции, если продукты просты.

**Практический пример: (пример 2)**

**Паттерн проектирования Builder (Строитель)**

**Цель паттерна**

Builder используется для пошагового построения сложных объектов. Основная задача паттерна — отделить процесс конструирования объекта от его представления, чтобы один и тот же процесс мог создавать разные представления объекта.

**Применяется, когда:**

Нужно создать объект с большим количеством параметров.

Некоторые параметры объекта могут быть опциональными.

Необходимо избежать конструктора с длинным списком аргументов.

**Причины возникновения паттерна**

Сложность объектов: Когда объект содержит много взаимозависимых частей, простой конструктор становится громоздким и неудобным.

Гибкость: Требуется гибко изменять процесс создания объекта без изменения его структуры.

Повторное использование: Часто встречаются схожие процессы создания объектов с небольшими отличиями.

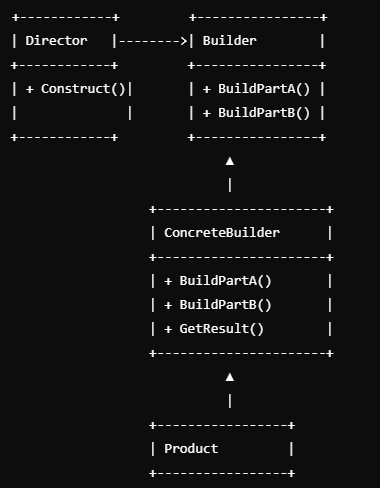
**Структура паттерна**

Продукт: Сложный объект, который нужно создать.

Строитель (Builder): Определяет интерфейс для пошагового построения продукта.

Конкретный строитель (Concrete Builder): Реализует методы интерфейса и определяет, как именно создаются части продукта.

Директор (Director): Управляет процессом построения объекта, вызывая шаги строителя в определённой последовательности.



Результаты использования паттерна

**Преимущества:**

Обеспечивает пошаговое создание сложных объектов.

Позволяет использовать один и тот же код для создания разных объектов.

Инкапсулирует код сборки, уменьшая зависимость клиента от структуры продукта.

**Недостатки:**

Увеличивает сложность из-за необходимости создания дополнительных классов.

Может быть избыточным для простых объектов.

**Практический пример использования паттерна: (пример 3);**

**Пример: Построение сложного объекта (Дом)**

**Особенности реализации**

Директор: Управляет процессом сборки, но не знает деталей реализации.

Строитель: Позволяет изменить логику создания объекта без изменения клиентского кода.

Продукт: Может быть представлен как сложный объект с множеством взаимосвязанных параметров.

**Использование в реальных приложениях**

Конструкторы запросов: Создание сложных SQL-запросов.

Создание документов: Генерация отчётов (PDF, HTML, DOC).

Игровая индустрия: Построение игровых персонажей или объектов с различными характеристиками.

**Паттерн проектирования Adapter (Адаптер)**

**Цель паттерна**

Adapter используется для обеспечения совместимости между двумя интерфейсами, которые не могут работать вместе напрямую. Паттерн позволяет объекту с одним интерфейсом работать с другим интерфейсом без изменения их исходного кода.

**Применяется, когда:**

Необходимо использовать существующий класс с несовместимым интерфейсом.

Нужно адаптировать класс для работы с другой библиотекой или API.

Хочется избежать изменений в существующем коде.

**Причины возникновения паттерна**

Наследие старого кода: Требуется интегрировать старый код (legacy code) с новым, не изменяя его.

Изоляция зависимостей: Нужно минимизировать зависимости от сторонних библиотек, предоставляя единый интерфейс.

Гибкость: Требуется адаптировать существующий функционал к новым требованиям, не переписывая его.

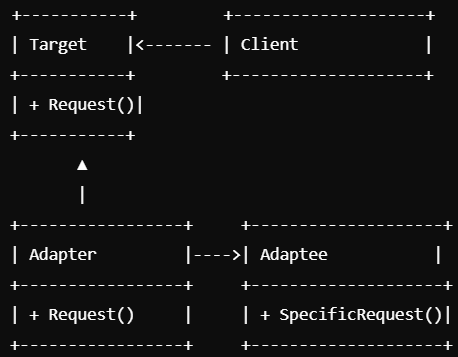
**Структура паттерна**

Целевой интерфейс (Target): Интерфейс, который клиент ожидает использовать.

Адаптируемый класс (Adaptee): Класс с несовместимым интерфейсом, который нужно адаптировать.

Адаптер (Adapter): Класс, реализующий целевой интерфейс и переводящий вызовы клиента в формат, понятный адаптируемому классу.

Клиент (Client): Работает с объектами через целевой интерфейс.



Результаты использования паттерна

**Преимущества:**

Позволяет использовать существующий класс без изменения его кода.

Сохраняет гибкость и расширяемость.

Изолирует клиент от адаптируемого класса.

**Недостатки:**

Увеличивает сложность из-за добавления дополнительных классов.

Может привести к снижению производительности, если адаптация включает сложные преобразования.

**Практический пример использования паттерна**

Пример: Адаптация несовместимого интерфейса USB-флешки для работы через старый разъём PS/2. **(пример 4)**

**Паттерн проектирования Facade (Фасад)**

**Цель паттерна**

Facade предоставляет простой унифицированный интерфейс для работы с набором сложных подсистем. Основная задача паттерна — сокрытие сложности системы за единым интерфейсом, удобным для клиента.

**Применяется, когда:**

Нужно упростить взаимодействие с большой и сложной системой.

Требуется минимизировать зависимость клиентского кода от деталей реализации подсистем.

Необходимо обеспечить гибкость в развитии подсистем без изменения интерфейса клиента.

**Причины возникновения паттерна**

Сложные подсистемы: В больших системах взаимодействие с множеством классов затрудняет разработку и тестирование.

Необходимость упрощения: Клиенты часто не нуждаются в доступе ко всем деталям подсистемы и работают только с определёнными её функциями.

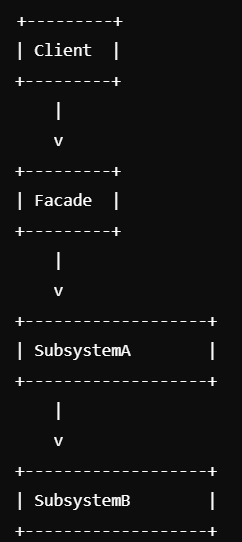
Изоляция клиентов: Уменьшение зависимости клиента от изменений в подсистемах.

**Структура паттерна**

Фасад (Facade): Определяет унифицированный интерфейс для клиента.

Подсистемы (Subsystems): Реализуют основную функциональность системы, но не знают о фасаде.

Клиент (Client): Работает с системой через фасад.



**Результаты использования паттерна**

**Преимущества:**

Сокрытие сложности системы за простым интерфейсом.

Уменьшение зависимости клиента от деталей реализации подсистем.

Упрощение тестирования и поддержки.

**Недостатки:**

Может скрывать слишком много деталей, что ограничивает возможности клиента.

Риск создания "Бога-объекта", если фасад начинает выполнять слишком много функций.

**Практический пример использования паттерна**

**Пример: Управление домашним кинотеатром (пример 5).**

**Особенности реализации**

Сокрытие реализации: Подсистемы остаются независимыми от фасада.

Гибкость: Подсистемы могут использоваться напрямую, если это необходимо.

Разделение ответственности: Фасад не выполняет задачи подсистем, он только координирует их.

**Использование в реальных приложениях**

UI Frameworks: Предоставление фасада для работы с множеством элементов интерфейса.

Библиотеки доступа к данным: Упрощение работы с базой данных.

API-интеграции: Унификация интерфейса для работы с внешними сервисами.

**Паттерн проектирования Observer (Наблюдатель)**

**Цель паттерна**

Observer предназначен для создания механизма "издатель-подписчик". Этот механизм позволяет одному объекту (издателю) уведомлять множество других объектов (наблюдателей) о произошедших изменениях в своём состоянии, не зная ничего о конкретных наблюдателях.

**Применяется, когда:**

Изменение состояния одного объекта требует автоматического обновления других объектов.

Требуется реализовать слабую связанность между компонентами системы.

**Причины возникновения паттерна**

Динамическая зависимость: Требуется автоматически уведомлять несколько объектов о событии, происходящем в другом объекте.

Изоляция компонентов: Уменьшение зависимости между источником событий и их обработчиками.

Гибкость: Позволяет динамически добавлять или удалять подписчиков.

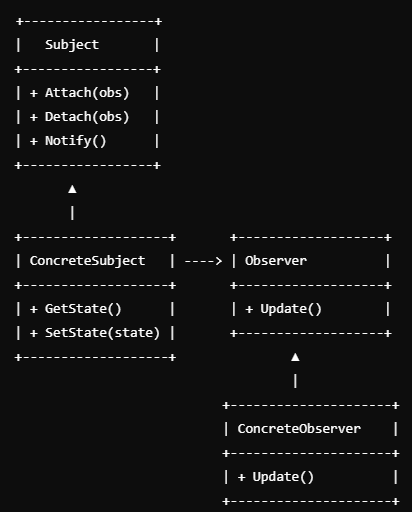
**Структура паттерна**

Издатель (Subject): Хранит список наблюдателей и предоставляет методы для их добавления/удаления. Уведомляет наблюдателей об изменениях.

Наблюдатель (Observer): Определяет интерфейс для получения уведомлений от издателя.

Конкретный издатель (ConcreteSubject): Реализует поведение издателя, содержит данные и изменяет своё состояние.

Конкретный наблюдатель (ConcreteObserver): Реагирует на уведомления издателя.



Результаты использования паттерна

**Преимущества:**

Уменьшение связанности между издателем и наблюдателями.

Динамическое добавление или удаление подписчиков.

Централизация уведомлений об изменениях.

**Недостатки:**

Возможность утечки памяти при неправильной реализации отписки наблюдателей.

Может вызывать каскадное обновление, что приводит к снижению производительности.

**Практический пример использования паттерна**

**Пример: Уведомление об изменении акций на бирже (пример 6)**

**Особенности реализации**

Слабая связанность: Наблюдатели не зависят от реализации издателя.

Механизм подписки: Издатель может позволить подписчикам подписываться и отписываться.

Асинхронность: Для уведомления можно использовать асинхронные вызовы, чтобы избежать блокировки.

**Использование в реальных приложениях**

Графические интерфейсы: Наблюдение за изменениями пользовательского ввода.

Реакция на события: Уведомление о состоянии объектов в играх или моделировании.

Механизмы подписки: Реализация систем подписки на новости, биржевые данные или уведомления.

**Паттерн проектирования Strategy (Стратегия)**

**Цель паттерна**

Strategy предоставляет способ инкапсуляции семейства алгоритмов, делая их взаимозаменяемыми и позволяя выбирать алгоритм во время выполнения.

**Применяется, когда:**

У программы есть несколько алгоритмов для выполнения одной задачи, и выбор алгоритма должен происходить динамически.

Избегается использование множества условных операторов (if, switch) для выбора поведения.

**Причины возникновения паттерна**

Повторное использование кода: Потребность отделить алгоритмы от их контекста, чтобы не дублировать их реализацию.

Расширяемость: Требуется добавить новые алгоритмы без изменения существующего кода.

Гибкость: Необходимо динамически изменять поведение объекта.

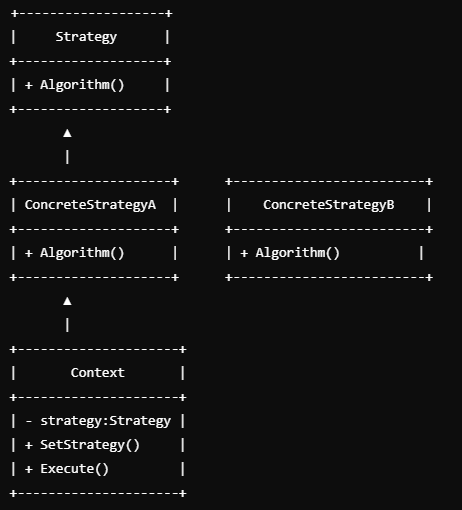
Снижение связности: Избавление от жесткого связывания контекста и конкретной реализации.

**Структура паттерна**

Контекст (Context): Использует ссылку на объект-стратегию для выполнения определённого поведения.

Интерфейс стратегии (Strategy): Определяет общий интерфейс для всех алгоритмов.

Конкретная стратегия (ConcreteStrategy): Реализует конкретное поведение, используя интерфейс стратегии.



Результаты использования паттерна

**Преимущества:**

Упрощение поддержки и расширения кода.

Устранение дублирования логики.

Инкапсуляция алгоритмов делает их независимыми от других частей системы.

Возможность выбора и изменения алгоритма во время выполнения.

**Недостатки:**

Усложнение программы за счёт увеличения числа классов.

Наличие дополнительных объектов может привести к увеличению объема памяти.

**Практический пример использования паттерна**

**Пример: Система расчёта стоимости доставки в зависимости от способа доставки (наземный, авиа, морской). (пример 7).**

**Особенности реализации**

Инкапсуляция: Алгоритмы выделяются в отдельные классы.

Расширяемость: Добавление новой стратегии требует создания нового класса, не затрагивая существующий код.

Контекст: Контекстный класс легко переключается между стратегиями.

**Использование в реальных приложениях**

Игры: Выбор поведения персонажей (агрессивное, защитное, нейтральное).

Системы оплаты: Реализация различных способов расчета стоимости услуг.

Криптография: Переключение между различными алгоритмами шифрования.