Порождающие паттерны

- Фабричный метод — это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.
- Абстрактная фабрика — это порождающий паттерн проектирования, который позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.
- Строитель — это порождающий паттерн проектирования, который позволяет создавать сложные объекты пошагово. Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений объектов.
- Прототип — это порождающий паттерн проектирования, который позволяет копировать объекты, не вдаваясь в подробности их реализации.
- Одиночка — это порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему лобальную точку доступа.

Фабричный метод

- Применимость: Паттерн можно часто встретить в любом C++ коде, где требуется гибкость при создании продуктов.
- Признаки применения паттерна: Фабричный метод можно определить по создающим методам, которые возвращают объекты продуктов через абстрактные типы или интерфейсы. Это позволяет переопределять типы создаваемых продуктов в подклассах

Плюсы:

- Избавляет класс от привязки к конкретным классам продуктов.
- Выделяет код производства продуктов в одно место, упрощая поддержку кода.
- Упрощает добавление новых продуктов в программу.
- Реализует принцип открытости/закрытости.

Минусы:

- Может привести к созданию больших параллельных иерархий классов, так как для каждого класса продукта надо создать свой подкласс создателя.

Абстрактная фабрика

- Применимость: Паттерн можно часто встретить в C++ коде, особенно там, где требуется создание семейств продуктов (например, внутри фреймворков).

- Признаки применения паттерна: Паттерн можно определить по методам, возвращающим фабрику, которая, в свою очередь, используется для создания конкретных продуктов, возвращая их через абстрактные типы или интерфейсы.

Плюсы:

- Гарантирует сочетаемость создаваемых продуктов.
- Избавляет клиентский код от привязки к конкретным классам продуктов.
- Выделяет код производства продуктов в одно место, упрощая поддержку кода.
- Упрощает добавление новых продуктов в программу.
- Реализует принцип открытости/закрытости.

Минусы:

- Усложняет код программы из-за введения множества дополнительных классов.
- Требует наличия всех типов продуктов в каждой вариации.

Строитель

Применимость: Паттерн можно часто встретить в C++ коде, особенно там, где требуется пошаговое создание продуктов или конфигурация сложных объектов.

Признаки применения паттерна: Строителя можно узнать в классе, который имеет один создающий метод и несколько методов настройки создаваемого продукта. Обычно, методы настройки вызывают для удобства цепочкой (например, someBuilder->setValueA(1)->setValueB(2)->create()).

Плюсы:

- Позволяет создавать продукты пошагово.
- Позволяет использовать один и тот же код для создания различных продуктов.
- Изолирует сложный код сборки продукта от его основной бизнес-логики.

Минусы:

- Усложняет код программы из-за введения дополнительных классов.
- Клиент будет привязан к конкретным классам строителей, так как в интерфейсе директора может не быть метода получения результата.

Прототип

Признаки применения паттерна: Прототип легко определяется в коде по наличию методов clone, copy и прочих.

Плюсы:

- Позволяет клонировать объекты, не привязываясь к их конкретным классам.
- Меньше повторяющегося кода инициализации объектов.
- Ускоряет создание объектов.
- Альтернатива созданию подклассов для конструирования сложных объектов.

Минусы:

- Сложно клонировать составные объекты, имеющие ссылки на другие объекты.

Одиночка

Применимость: Многие программисты считают Одиночку антипаттерном, поэтому его всё реже и реже можно встретить в C++ коде.

Признаки применения паттерна: Одиночку можно определить по статическому создающему методу, который возвращает один и тот же объект.

Плюсы:

- Гарантирует наличие единственного экземпляра класса.
- Предоставляет к нему глобальную точку доступа.
- Реализует отложенную инициализацию объекта-одиночки.
- Нарушает принцип единственной ответственности класса.

Минусы:

- Маскирует плохой дизайн.
- Проблемы мультипоточности.
- Требует постоянного создания Mock-объектов при юнит-тестировании.

Структурные паттерны

- Адаптер — это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе.
- Мост — это структурный паттерн проектирования, который разделяет один или несколько классов на две отдельные иерархии — абстракцию и реализацию, позволяя изменять их независимо друг от друга.
- Компоновщик — это структурный паттерн проектирования, который позволяет сгруппировать множество объектов в древовидную структуру, а затем работать с ней так, как будто это единичный объект.
- Декоратор — это структурный паттерн проектирования, который позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».
- Фасад — это структурный паттерн проектирования, который предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку.
- Легковес — это структурный паттерн проектирования, который позволяет вместить бóльшее количество объектов в отведённую оперативную память. Легковес экономит память, разделяя общее состояние объектов между собой, вместо хранения одинаковых данных в каждом объекте.
- Заместитель — это структурный паттерн проектирования, который позволяет подставлять вместо реальных объектов специальные объекты-заменители. Эти объекты перехватывают вызовы к оригинальному объекту, позволяя сделать что-то до или после передачи вызова оригиналу

Адаптер

Применимость: Паттерн можно часто встретить в C++ коде, особенно там, где требуется конвертация разных типов данных или совместная работа классов с разными интерфейсами.

Признаки применения паттерна: Адаптер получает конвертируемый объект в конструкторе или через параметры своих методов. Методы Адаптера обычно совместимы с интерфейсом одного объекта. Они делегируют вызовы вложенному объекту, превратив перед этим параметры вызова в формат, поддерживаемый вложенным объектом.

Плюсы:

- Отделяет и скрывает от клиента подробности преобразования различных интерфейсов.

Минусы:

- Усложняет код программы из-за введения дополнительных классов.

Мост

Применимость: Паттерн Мост особенно полезен когда вам приходится делать кросс-платформенные приложения, поддерживать несколько типов баз данных или работать с разными поставщиками похожего API (например, cloud-сервисы, социальные сети и т. д.)

Признаки применения паттерна: Если в программе чётко выделены классы «управления» и несколько видов классов «платформ», причём управляющие объекты делегируют выполнение платформам, то можно сказать, что у вас используется Мост.

Плюсы:

- Позволяет строить платформо-независимые программы.
- Скрывает лишние или опасные детали реализации от клиентского кода.
- Реализует принцип открытости/закрытости.

Минусы:

- Усложняет код программы из-за введения дополнительных классов.

Компоновщик

Применимость: Паттерн Компоновщик встречается в любых задачах, которые связаны с построением дерева. Самый простой пример — составные элементы GUI, которые тоже можно рассматривать как дерево.

Признаки применения паттерна: Если из объектов строится древовидная структура, и со всеми объектами дерева, как и с самим деревом работают через общий интерфейс.

Плюсы:

- Упрощает архитектуру клиента при работе со сложным деревом компонентов.
- Облегчает добавление новых видов компонентов.

Минусы:

- Создаёт слишком общий дизайн классов.

Декоратор

Применимость: Паттерн можно часто встретить в C++ коде, особенно в коде, работающем с потоками данных.

Признаки применения паттерна: Декоратор можно распознать по создающим методам, которые принимают в параметрах объекты того же абстрактного типа или интерфейса, что и текущий класс.

Плюсы:

- Большая гибкость, чем у наследования.
- Позволяет добавлять обязанности на лету.
- Можно добавлять несколько новых обязанностей сразу.
- Позволяет иметь несколько мелких объектов вместо одного объекта на все случаи жизни.

Минусы:

- Трудно конфигурировать многократно обёрнутые объекты.
- Обилие крошечных классов.

Фасад

Применимость: Паттерн часто встречается в клиентских приложениях, написанных на C++, которые используют классы-фасады для упрощения работы со сложными библиотеки или API.

Признаки применения паттерна: Фасад угадывается в классе, который имеет простой интерфейс, но делегирует основную часть работы другим классам. Чаще всего, фасады сами следят за жизненным циклом объектов сложной системы.

Плюсы:

- Изолирует клиентов от компонентов сложной подсистемы.

Минусы:

- Фасад рискует стать божественным объектом, привязанным ко всем классам программы.

Легковес

Применимость: Весь смысл использования Легковеса — в экономии памяти. Поэтому, если в приложении нет такой проблемы, то вы вряд ли найдёте там примеры Легковеса.

Признаки применения паттерна: Легковес можно определить по создающим методам класса, которые возвращают закешированные объекты, вместо создания новых.

Плюсы:

Экономит оперативную память.

Минусы:

Расходует процессорное время на поиск/вычисление контекста. Усложняет код программы из-за введения множества дополнительных классов.

Заместитель

Применимость: Паттерн Заместитель применяется в C++ коде тогда, когда надо заменить настоящий объект его суррогатом, причём незаметно для клиентов настоящего объекта. Это позволит выполнить какие-то добавочные поведения до или после основного поведения настоящего объекта.

Признаки применения паттерна: Класс заместителя чаще всего делегирует всю настоящую работу своему реальному объекту. Заместители часто сами следят за жизненным циклом своего реального объекта.

Плюсы:

- Позволяет контролировать сервисный объект незаметно для клиента.
- Может работать, даже если сервисный объект ещё не создан.
- Может контролировать жизненный цикл служебного объекта.

Минусы:

- Усложняет код программы из-за введения дополнительных классов.
- Увеличивает время отклика от сервиса.

Поведенческие

- Цепочка обязанностей — это поведенческий паттерн проектирования, который позволяет передавать запросы последовательно по цепочке обработчиков. Каждый последующий обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи.
- Команда — это поведенческий паттерн проектирования, который превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций.
- Итератор — это поведенческий паттерн проектирования, который даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.
- Посредник — это поведенческий паттерн проектирования, который позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник.
- Снимок — это поведенческий паттерн проектирования, который позволяет сохранять и восстанавливать прошлые состояния объектов, не раскрывая подробностей их реализации.
- Наблюдатель — это поведенческий паттерн проектирования, который создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.
- Состояние — это поведенческий паттерн проектирования, который позволяет объектам менять поведение в зависимости от своего состояния. Извне создаётся впечатление, что изменился класс объекта.

- Стратегия — это поведенческий паттерн проектирования, который определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы.
- Шаблонный метод — это поведенческий паттерн проектирования, который определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Паттерн позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.
- Посетитель — это поведенческий паттерн проектирования, который позволяет добавлять в программу новые операции, не изменяя классы объектов, над которыми эти операции могут выполняться

Цепочка обязанностей

Применимость: Паттерн встречается в C++ не так уж часто, так как для его применения нужна цепь объектов, например, связанный список.

Признаки применения паттерна: Цепочку обязанностей можно определить по спискам обработчиков или проверок, через которые пропускаются запросы. Особенно если порядок следования обработчиков важен.

Плюсы:

- Уменьшает зависимость между клиентом и обработчиками.
- Реализует принцип единственной обязанности.
- Реализует принцип открытости/закрытости.

Минусы:

- Запрос может остаться никем не обработанным.

Команда

Плюсы:

- Убирает прямую зависимость между объектами, вызывающими операции, и объектами, которые их непосредственно выполняют.
- Позволяет реализовать простую отмену и повтор операций.
- Позволяет реализовать отложенный запуск операций.
- Позволяет собирать сложные команды из простых.
- Реализует принцип открытости/закрытости.

Минусы:

- Усложняет код программы из-за введения множества дополнительных классов.

Применимость: Паттерн можно часто встретить в C++ коде, особенно когда нужно откладывать выполнение команд, выстраивать их в очереди, а также хранить историю и делать отмену.

Признаки применения паттерна: Классы команд построены вокруг одного действия и имеют очень узкий контекст. Объекты команд часто подаются в обработчики событий элементов GUI. Практически любая реализация отмены использует принципа команд.

Итератор

Применимость: Паттерн можно часто встретить в C++ коде, особенно в программах, работающих с разными типами коллекций, и где требуется обход разных сущностей.

Признаки применения паттерна: Итератор легко определить по методам навигации (например, получения следующего/предыдущего элемента и т. д.). Код использующий итератор зачастую вообще не имеет ссылок на коллекцию, с которой работает итератор. Итератор либо принимает коллекцию в параметрах конструкторе при создании, либо возвращается самой коллекцией.

Плюсы:

- Упрощает классы хранения данных.
- Позволяет реализовать различные способы обхода структуры данных.
- Позволяет одновременно перемещаться по структуре данных в разные стороны.

Минусы:

- Не оправдан, если можно обойтись простым циклом.

Посредник

Применимость: Пожалуй, самое популярное применение Посредника в C++ коде — это связь нескольких компонентов GUI одной программы.

Плюсы:

- Устраняет зависимости между компонентами, позволяя повторно их использовать.
- Упрощает взаимодействие между компонентами.
- Централизует управление в одном месте.

Минусы:

- Посредник может сильно раздуться.

Снимок

Применимость: Снимок на C++ чаще всего реализуют с помощью сериализации. Но это не единственный, да и не самый эффективный метод сохранения состояния объектов во время выполнения программы.

Плюсы:

- Не нарушает инкапсуляции исходного объекта.
- Упрощает структуру исходного объекта. Ему не нужно хранить историю версий своего состояния.

Минусы:

- Требует много памяти, если клиенты слишком часто создают снимки.
- Может повлечь дополнительные издержки памяти, если объекты, хранящие историю, не + освобождают ресурсы, занятые устаревшими снимками.

Наблюдатель

Применимость: Наблюдатель можно часто встретить в C++ коде, особенно там, где применяется событийная модель отношений между компонентами. Наблюдатель позволяет отдельным компонентам реагировать на события, происходящие в других компонентах.

Признаки применения паттерна: Наблюдатель можно определить по механизму подписки и методам оповещения, которые вызывают компоненты программы.

Плюсы:

- Издатели не зависят от конкретных классов подписчиков и наоборот.
- Вы можете подписывать и отписывать получателей на лету.
- Реализует принцип открытости/закрытости.

Минусы:

- Подписчики оповещаются в случайном порядке.

Состояние

Применимость: Паттерн Состояние часто используют в C++ для превращения в объекты громоздких стейт-машин, построенных на операторах switch.

Признаки применения паттерна: Методы класса делегируют работу одному вложенному объекту.

Плюсы:

- Избавляет от множества больших условных операторов машины состояний.
- Концентрирует в одном месте код, связанный с определённым состоянием.
- Упрощает код контекста.

Минусы:

- Может неоправданно усложнить код, если состояний мало и они редко меняются.

Стратегия

Применимость: Стратегия часто используется в C++ коде, особенно там, где нужно подменять алгоритм во время выполнения программы. Многие примеры стратегии можно заменить простыми lambda-выражениями.

Признаки применения паттерна: Класс делегирует выполнение вложенному объекту абстрактного типа или интерфейса.

Плюсы:

- Горячая замена алгоритмов на лету.
- Изолирует код и данные алгоритмов от остальных классов.
- Уход от наследования к делегированию.
- Реализует принцип открытости/закрытости.

Минусы:

- Усложняет программу за счёт дополнительных классов.
- Клиент должен знать, в чём состоит разница между стратегиями, чтобы выбрать подходящую.

Шаблонный метод

Применимость: Шаблонные методы можно встретить во многих библиотечных классах C++. Разработчики создают их, чтобы позволить клиентам легко и быстро расширять стандартный код при помощи наследования.

Признаки применения паттерна: Класс заставляет своих потомков реализовать методы-шаги, но самостоятельно реализует структуру алгоритма.

Плюсы:

- Облегчает повторное использование кода.

Минусы:

- Вы жёстко ограничены скелетом существующего алгоритма.
- Вы можете нарушить принцип подстановки Барбары Лисков, изменяя базовое поведение одного из шагов алгоритма через подкласс.
- С ростом количества шагов шаблонный метод становится слишком сложно поддерживать.

Посетитель

Применимость: Посетитель нечасто встречается в C++ коде из-за своей сложности и нюансов реализазации.

Плюсы:

- Упрощает добавление операций, работающих со сложными структурами объектов.
- Объединяет родственные операции в одном классе.
- Посетитель может накапливать состояние при обходе структуры элементов.

Минусы:

- Паттерн не оправдан, если иерархия элементов часто меняется.
- Может привести к нарушению инкапсуляции элементов.

Связи между паттернами

Порождающие паттерны

- Многие архитектуры начинаются с применения Фабричного метода (более простого и расширяемого через подклассы) и эволюционируют в сторону Абстрактной фабрики, Прототипа или Строителя (более гибких, но и более сложных).
- Классы Абстрактной фабрики чаще всего реализуются с помощью Фабричного метода, хотя они могут быть построены и на основе Прототипа.

- Фабричный метод можно использовать вместе с Итератором, чтобы подклассы коллекций могли создавать подходящие им итераторы.
- Прототип не опирается на наследование, но ему нужна сложная операция инициализации. Фабричный метод, наоборот, построен на наследовании, но не требует сложной инициализации.
- Фабричный метод можно рассматривать как частный случай Шаблонного метода. Кроме того, Фабричный метод нередко бывает частью большого класса с Шаблонными методами.
- Строитель концентрируется на построении сложных объектов шаг за шагом. Абстрактная фабрика специализируется на создании семейств связанных продуктов. Строитель возвращает продукт только после выполнения всех шагов, а Абстрактная фабрика возвращает продукт сразу же.
- Абстрактная фабрика может быть использована вместо Фасада для того, чтобы скрыть платформо-зависимые классы.
- Абстрактная фабрика может работать совместно с Мостом. Это особенно полезно, если у вас есть абстракции, которые могут работать только с некоторыми из реализаций. В этом случае фабрика будет определять типы создаваемых абстракций и реализаций.
- Абстрактная фабрика, Строитель и Прототип могут быть реализованы при помощи Одиночки.
- Строитель позволяет пошагово сооружать дерево Компоновщика.
- Паттерн Строитель может быть построен в виде Моста: директор будет играть роль абстракции, а строители — реализации.
- Если Команду нужно копировать перед вставкой в историю выполненных команд, вам может помочь Прототип.
- Архитектура, построенная на Компоновщиках и Декораторах, часто может быть улучшена за счёт внедрения Прототипа. Он позволяет клонировать сложные структуры объектов, а не собирать их заново.
- Фасад можно сделать Одиночкой, так как обычно нужен только один объект□фасад.
- Паттерн Легковес может напоминать Одиночку, если для конкретной задачи у вас получилось свести количество объектов к одному. Но помните, что между паттернами есть два кардинальных отличия:
    1. В отличие от Одиночки, вы можете иметь множество объектов□легковесов.
    2. Объекты-легковесы должны быть неизменяемыми, тогда как объект□одиночка допускает изменение своего состояния.

Структурные паттерны

- Мост проектируют загодя, чтобы развивать большие части приложения отдельно друг от друга. Адаптер применяется постфактум, чтобы заставить несовместимые классы работать вместе.
- Адаптер меняет интерфейс существующего объекта. Декоратор улучшает другой объект без изменения его интерфейса. Причём Декоратор поддерживает рекурсивную вложенность, чего не скажешь об Адаптере.
- Адаптер предоставляет классу альтернативный интерфейс. Декоратор предоставляет расширенный интерфейс. Заместитель предоставляет тот же интерфейс.
- Фасад задаёт новый интерфейс, тогда как Адаптер повторно использует старый. Адаптер оборачивает только один класс, а Фасад оборачивает целую подсистему. Кроме того, Адаптер позволяет двум существующим интерфейсам работать сообща, вместо того, чтобы задать полностью новый.
- Мост, Стратегия и Состояние (а также слегка и Адаптер) имеют схожие структуры классов — все они построены на принципе «композиции», то есть делегирования работы другим объектам. Тем

не менее, они отличаются тем, что решают разные проблемы. Помните, что паттерны — это не только рецепт построения кода определённым образом, но и описание проблем, которые привели к данному решению.

- Цепочку обязанностей часто используют вместе с Компоновщиком. В этом случае запрос передаётся от дочерних компонентов к их родителям.
- Вы можете обходить дерево Компоновщика, используя Итератор.
- Вы можете выполнить какое-то действие над всем деревом Компоновщика при помощи Посетителя.
- Компоновщик часто совмещают с Легковесом, чтобы реализовать общие ветки дерева и сэкономить при этом память.
- Компоновщик и Декоратор имеют похожие структуры классов из-за того, что оба построены на рекурсивной вложенности. Она позволяет связать в одну структуру бесконечное количество объектов. Декоратор оборачивает только один объект, а узел Компоновщика может иметь много детей. Декоратор добавляет вложенному объекту новую функциональность, а Компоновщик не добавляет ничего нового, но «суммирует» результаты всех своих детей. Но они могут и сотрудничать: Компоновщик может использовать Декоратор, чтобы переопределить функции отдельных частей дерева компонентов.
- Цепочка обязанностей и Декоратор имеют очень похожие структуры. Оба паттерна базируются на принципе рекурсивного выполнения операции через серию связанных объектов. Но есть и несколько важных отличий. Обработчики в Цепочке обязанностей могут выполнять произвольные действия, независимые друг от друга, а также в любой момент прерывать дальнейшую передачу по цепочке. С другой стороны Декораторы расширяют какое-то определённое действие, не ломая интерфейс базовой операции и не прерывая выполнение остальных декораторов.
- Стратегия меняет поведение объекта «изнутри», а Декоратор изменяет его «снаружи».
- Декоратор и Заместитель имеют схожие структуры, но разные назначения. Они похожи тем, что оба построены на принципе композиции и делегируют работу другим объектам. Паттерны отличаются тем, что Заместитель сам управляет жизнью сервисного объекта, а обёртывание Декораторов контролируется клиентом.
- Легковес показывает, как создавать много мелких объектов, а Фасад показывает, как создать один объект, который отображает целую подсистему.
- Посредник и Фасад похожи тем, что пытаются организовать работу множества существующих классов. о Фасад создаёт упрощённый интерфейс к подсистеме, не внося в неё никакой добавочной функциональности. Сама подсистема не знает о существовании Фасада. Классы подсистемы общаются друг с другом напрямую. о Посредник централизует общение между компонентами системы. Компоненты системы знают только о существовании Посредника, у них нет прямого доступа к другим компонентам.
- Фасад похож на Заместитель тем, что замещает сложную подсистему и может сам её инициализировать. Но в отличие от Фасада, Заместитель имеет тот же интерфейс, что его служебный объект, благодаря чему их можно взаимозаменять.

Поведенческие паттерны

- Обработчики в Цепочке обязанностей могут быть выполнены в виде Команд. В этом случае множество разных операций может быть выполнено над одним и тем же контекстом, коим является запрос. Но есть и другой подход, в котором сам запрос является Командой, посланной

по цепочке объектов. В этом случае одна и та же операция может быть выполнена над множеством разных контекстов, представленных в виде цепочки.

- Команду и Снимок можно использовать сообща для реализации отмены операций. В этом случае объекты команд будут отвечать за выполнение действия над объектом, а снимки будут хранить резервную копию состояния этого объекта, сделанную перед самым запуском команды.

- Команда и Стратегия похожи по духу, но отличаются масштабом и применением: о Команду используют, чтобы превратить любые разнородные действия в объекты. Параметры операции превращаются в поля объекта. Этот объект теперь можно логировать, хранить в истории для отмены, передавать во внешние сервисы и так далее. о С другой стороны, Стратегия описывает разные способы произвести одно и то же действие, позволяя взаимозаменять эти способы в каком□то объекте контекста.

- Посетитель можно рассматривать как расширенный аналог Команды, который способен работать сразу с несколькими видами получателей.

- Снимок можно использовать вместе с Итератором, чтобы сохранить текущее состояние обхода структуры данных и вернуться к нему в будущем, если потребуется.

- Посетитель можно использовать совместно с Итератором. Итератор будет отвечать за обход структуры данных, а Посетитель — за выполнение действий над каждым её компонентом.

- Разница между Посредником и Наблюдателем не всегда очевидна. Чаще всего они выступают как конкуренты, но иногда могут работать вместе.

    1. Цель Посредника — убрать обоюдные зависимости между компонентами системы. Вместо этого они становятся зависимыми от самого посредника. С другой стороны, цель Наблюдателя — обеспечить динамическую одностороннюю связь, в которой одни объекты косвенно зависят от других.
    2. Довольно популярна реализация Посредника при помощи Наблюдателя. При этом объект посредника будет выступать издателем, а все остальные компоненты станут подписчиками и смогут динамически следить за событиями, происходящими в посреднике. В этом случае трудно понять, чем же отличаются оба паттерна.
    3. Но Посредник имеет и другие реализации, когда отдельные компоненты жёстко привязаны к объекту посредника. Такой код вряд ли будет напоминать Наблюдателя, но всё же останется Посредником.
    4. Напротив, в случае реализации посредника с помощью Наблюдателя представим такую программу, в которой каждый компонент системы становится издателем. Компоненты могут подписываться друг на друга, в то же время не привязываясь к конкретным классам. Программа будет состоять из целой сети Наблюдателей, не имея центрального объекта-Посредника.

- Состояние можно рассматривать как надстройку над Стратегией. Оба паттерна используют композицию, чтобы менять поведение основного объекта, делегируя работу вложенным объектам-помощникам. Однако в Стратегии эти объекты не знают друг о друге и никак не связаны. В Состоянии сами конкретные состояния могут переключать контекст.

- Шаблонный метод использует наследование, чтобы расширять части алгоритма. Стратегия использует делегирование, чтобы изменять выполняемые алгоритмы на лету. Шаблонный метод

работает на уровне классов. Стратегия позволяет менять логику отдельных объектов