- если метод реализован в header, то он становится inline
- static переменные не определяются в классе, определяются в другом месте?
- с 11 версии переменные можно инициализировать по умолчанию, до этого можно было инициализировать только static const
- protected - к полям имеют доступ производные
- static методы не принимают this, могут вызываться если нет объекта
- .* - объкт.*указаетль на метод
- ->* - указатель на объкт.*указаетль на метод
- в методах мы имеем доступ к полям своего объектв и к полям объектов того же класса
- private-наследование - это полная замена интерфейса, то есть было базовое понятие, на основе него мы породили новый класс с совершенно другим интерфейсом. Дочерний класс не имеет очевидной связи с родительским классом, но использует его в своей реализации.
- при наследовании protected члены public будут менять уровень на protected, то есть так же как при private-наследовании происходит подмена интерфейса, но интерфейс базового мы оставляем доступным для производных производного
- при public-наследовании все члены наследуются со совим уровнем доступа, то есть происходит расширение интерфеса или подмена базового на производный
- если часть интерфейса хочу скрыть, а часть оставить, то можно использовать private/protected наследование, но можно вернуть доступ
- конструктор по умолчанию обнуляет поля
- в разделе инициализации объекта еще нет, можно инициализировать базы и поля объекта
- деструктор вызывается неявно, не принимает параметров. Для автоматических объектов деструктор вызывается после выхода из блока, для внешних и внешних-статических и для статических после выполнения программы, для временных - после вычисления выражения. Деструкторы объектов выполняются в порядке, обратном созданию объектов, delete так же приводит к вызову деструктора. Когда явно определяем – когда захватываем ресурсы. Не мб const, volatile, static, мб virtual
- Выделение общей базы
-- над понятиями выполняем одни и те же действия (единая схема использования) – троллейбус трамвай
-- общее в операциях над объектами
-- в операциях общая реализация
-- если 2 понятия фигурируют вместе в обсуждении проекта

- Расщепление
-- два подмножества операций класса используются независимо друг от друга в разных местах программы, то есть один объект не должен выполнять несколько ролей
-- операции не имеют связанную реализацию
-- понятие фигурирует в двух никак не связных дискуссиях о проекте

- круглые скобочки - бинарный, может быть перегружен несколько раз с 14 версии (примеры!!)
- квадратные скобочки - бинарный, слева объект, справа все что угодно, поэтому может быть создан ассоциативный массив
- new и delete перегружаются как статические члены класса (так же могут быть перегружены как глобальные объекты)

-арх домен - предоставляет общие механизмы и структуры для управления данными и в общем-то управления всей нашей системой, как единым целым.

Порождающие паттерны - задача - они создают объекты, их цель состоит в том, чтобы частично решить следующую проблему: когда мы используем полиморфизм, мы используем указатель или ссылку на базовое понятие, но возникает вопрос с созданием объектов. Нам надо создавать конкретный объект, но полиморфных конструкторов не существует.

Фабричный метод - идея в том, чтобы разнести на 2 задачи принятие рещения, какой объект создавать и непосредственно создание. Причем при создании объекта отвязаться от конкретного типа.
используется когда:
- подмена одного объекта на другой, то есть, когда нам нужно создавать какой-то объект в методе, мы можем его легко подменить на другой
- решение какой продукт создавать и создание отделены
основан на полиморфизме
преимущества:
- облегчается создание новых классов
- избавляем методы от привязки к конкретным классам
есть реализация, при которой объект создаётся один раз! это является альтернативой одиночке

Абстрактная фабрика - порождает семейство объектов, развитие идеи фабричного метода
нам нужно создавать объекты разных классов, но они должны относиться к какой-то одной группе. можно создавать ветви креаторов под каждый продукт, но тогда мы теряем связь между продуктами (пример с графической библиотекой: ручка, кисточка, сцена)
Преимущества те же, что у фабричного метода
- не нужно контроллировать, что будут созданы объекты из разных семейств
Минус:
абстрактная фабрика накладывает ограничвения на продукты, то есть все фабрики должны поддерживать базовый интерфейс, что не всегда возможно

Строитнель
Могут быть объекты, которые создаются поэтапно
идея: создать объекты, которые будут отвечать за создание сложных объектов
выделяем так же класс, который будет контроллировать создание объекта
используем когда:
- есть сложные объекты, которые создаются поэтапно
- если объект создается не сразу, данные, необходимые для создания объекта подготавливаются поэтапно
преимущества:
- создание продукта пошагово
- вынесение создания и контроля в отдельные классы
минусы:
- конкретные строители должны базироваться на одних и тех же данных, то есть надо обеспечить, чтоюы можно было заменить одного строителя на другой

Прототип
мы рабтаем с ссылкой или указателем на объект и надо создать копию объекта. можно создать класс, который будет всё это создавать, там будет свич, вещь получится нерасширяемой
решаем проблему с помощью данного паттрена
мы добавляем метод clone в базовый класс, который возвращает указатель на себя, а в производных реализуем копирование, соответствующее данному объекту

Одиночка
возникают задачи, когда должен быть создан гарантированно один объект данного класса
как можно это сделать?
убрать конструктор без параметров
делают статический метод, который по необходимости создает объект. это реализуется с помощью статичексого члена:
static shared_ptr<Product> myInstance(new Product());
return myInstance;
конструктор находится в private
запрет копирования (конструктор и оператор присваивания = delete)
Недостатки:
- мы как бы создаем глобальный объект, доступ к объекту идет через глобальный интерфейс, то есть через имя класса
- проблема с подменой, мы можем создавать объект этого класса, то есть мы лишаемся возможности подмены на этапе выполненя, то есть мы на этапе выпллнения решаем, какой объект нам нужен, в данном случае мы не можем его создать ???

Пул объектов
предоставляет набор готовых объектов, которые мы можем использовать
когда используем:
- ситуация, когда создание объекта - трудоемкий процесс (такие объекты объединяем в пул)
- в один момент надо держать определённое количество объектов в системе (например, количество потоков свободных в данный момент)
обязательно после возврата в пул нужно вернуть ему его начальное состояние

Структурные паттерны - структурные решения, какая-либо декомпозиция классов с использованием схем наследования, включения

Адаптер
создание объектов, которые в разных местах программы используются по-разному, то есть объект выступает как бы в разных ролях, соотвественно, для каждой роли могут быть свои интерфейсы. Они могут частично пересекаться,

но попытка объединения всё равно приведёт к избыточному интерфейсу и к обладанию несколькими ответственностями у одного объекта.
Идея - подменить интерфейс, чтобы в зависимости от ситуации использовать объект по-разному.
Кроме того, у нас может быть класс, который мы хотим встроить в систему, но в системе у нас другой интерфейс, то есть нужен адаптер.
преимущества:
- класс с любым интерфейсом может быть использован в программе
- позволяет не создавать классы с несколькими ответственностями

Декоратор
проблема: надо добавить функционал
решение1: просто наследоваться от классов (пусть есть базовый и два наследника, нужно добавить функционал одинаковый в оба класса, соответственно делаем у каждого наследника по наследнику).
Недостатки: дублирование кода в двух наследниках, разрастание иерархии
решение2:
Идея: добавление этого функционала выносится в отдельный класс
преимущества:
- сокращается иерархия
- декорировать можем во время выполнения
- избавляемся от дублирования кода, код уходит в декоратор
Недостатки:
- убрать обёртку проблематично, приходится создавать новые обёртки

Компоновщик
Часто мы работаем с разными объектами однотипно, то есть выполняем те же операции. Иногда нужно выполнять те же действия над группой: например, выполнение преобразований над объектами на сцене - над несколькими моделями
Идея: компоновка объектов в древовидную структуру, в которой над иерархией могут быть выполнены те же действия, что над простым объектом.
преимущества:
- имеем возможность во время выполнения собирать сложный объект, выполнять над сборкой общих действий
- клиент не думает, над чем он работает: сложным или простым объектом

Заместитель

Позволяет работать не с реальным объектом, а с объектом, его подменяющим. Подменяющий может давать или не давать доступ к основному объекту, может заниматься статистикой запросов. Реальные объекты могут выполнять некоторые действия очень долго, заместитель же может сохранять предыдущее состояние этого объекта или запоминать предыдущие аналогичные запросы и повторно не обращаться к основному объекту, чтобы сократить время выполнения. При этом если реальный объект изменился, то прокси должен узнать об этом.

Преимущества:

- можно контролировать объект незаметно для пользователя
- прокси может работать, когда объекта нет (сказать, что устарел)

Недостатки:

- падает время доступа, так как все обращения идут через прокси (например, на сбор статистики тратится время)
- хранение истории обращений в прокси => память

Мост

Решаемые проблемы:

- при большой иерархии часто бывает такое, что для одного объекта бывает несколько реализаций и во время работы хочется поменять реализацию
- постоянно возникает дублирование кода при разрастании реализации

Идея: разделить понятие самого объекта и реализации в отдельные классы

Преимущества:

- сокращение количества классов
- сделаем систему более гибкой (во время работы можем менять реализацию) => уходим от дублирования кода
- независимо от логики можем менять реализацию

Недостатки:

- вынесение связи включения на уровень базового класса

Используем:

- во время выполнения надо менять реализацию
- большая иерархия и по разным ветвям иерархии будут одинаковые реализации, следовательно, их нужно вынести в реализацию

Фасад

Идея: есть группа объектов, связанных между собой и эти связи довольно жёсткие и чтобы извне не работать с каждым объектом в отдельности мы можем все эти объекты объединить в один класс - фасад, который будет представлять интерфейс для работы с объединениям этих объектов
- нам не нужно извне работать с мелкими объектами (внутри целый мир, а снаружи простой интерфейс, то есть уменьшается количество связей)
- фасад может контролировать целостность системы объектов

Паттерны поведения - все остальные

Стратегия
проблема: во время выполнения надо менять реализацию какого-либо метода, мы можем делать производные с разными реализациями и осуществлять миграцию между этими классами - неудобно, так как, чтобы осуществить миграцию, надо работать с конкретными объектами, не работать через базовый класс
Идея: это изменяемое действие вынести в отдельный класс - стратегию и в основном классе держать указатель на этот объект стратегии (через базовый можем подменять)
Паттерн стратегия определяет семейство возможных алгоритмов

Ряд паттернов, которые дают возможность обработать запросы:

Команда
В системе возможны разные запросы.
Идея: обернуть запрос в отдельный класс, причём этот класс мб как простым, так и составным
Преимущества:
- уменьшается зависимость между объектами (например, одна подсистема общается с другой посредством команд, нам не нужно держать связи между конкретными объектами, чтобы всё это выполнить)
- формирование команды и ее выполнение могут быть сделаны в разное время, или даже формировать очередь из команд
- Можно добавить к команде композит, тогда мы сможем выполнять сложные команды из нескольких команд

Цепочка обязанностей

Композит из команд актуален тогда, когда один объект обрабатывает все запросы, однако это не всегда так, поэтому возникла идея создать цепочку обработчиков. Паттерн позволяет передавать запрос последовательно по цепочке обработчиков, каждый сам решает обрабатывать ли запрос и передавать ли следующему.
Преимущества и условия использования:
- один и тот же запрос может быть обработан разными способами
- есть четкая последовательность запросов
- на этапе выполнения мы формируем цепочку

Подписчик-издатель
очень часто запрос надо передавать не одному объекту, а многим объектам, и это должно выполняться на этапе выполнения, причем у класса, который получает инфу на обработку, может не один метод принимать, а несколько.
задает механизм: тот, кто хочет принять событие, подписывается у издателя, если он подписался, он получает, если не подписался, то не получает. Получаем механизм, при котором группа объектов реагирует на изменение одного объекта, а этот объект естественно оповещает подписчиков, вызывая соответствующие методы.
Преимущества:
- подписчик и издатель независимы
- схема гибкая: можно подписаться и отписаться
Недостатки:
- нужно держать список подписчиков
- нет порядка в оповещении
- если издателей много, если подписчики сами являются издателями, если это сложная система, то:
-- при shared_ptr проблемно всё это развязать
-- затраты по памяти
-- работать с такой кашей сложно

Посредник
НО можно вынести все связи в отдельный объект
Тогда каждый объект будет обращаться к объекту-связи, а он уже будет искать, с кем нужно связаться. Объект-посредник берёт на себя задачу переадресации сообщений.

Паттерны команда, цепочка обязанностей, подписчик-издатель и посредник реализуют обработку запросов: команда четко фиксирует куда идет запрос и вызывает обработчика запроса, цепочка обязанностей – запрос проходит по цепочке обработчиков и кто может, запрос обрабатывает, подписчик-издатель отправляет на множество подписчиков запрос, вызывает их метод, а посредник берёт на себя функцию связи между объектами или выполняет переадресацию запросов одному или нескольким объектам. В зависимости от задачи выбирается тот или иной паттерн.

Посетитель

Проблема, связанная с изменением интерфейса. Мы не можем в производном классе ни сузить, ни расширить интерфейс. А если необходимо расширить, то можно использовать адаптер, с помощью которого мы можем добавить функционал на основе того функционала, который у нас есть. Однако Визитор позволяет расширять функционал или подменять на этапе выполнения.
Визитор фактически собирает разный функционал разных классов.
Преимущества:
- можем добавить функционал как для всех объектов, так и для нескольких
- в один класс добавляем один и тот же функционал для разных объектов
Недостатки:
- расширяется иерархия => visitor надо менять
- меняется иерархия => полиморфизм опять не срабатывает

Опекун

Выполняем какие-то операции и в какой-то момент хотим откатиться. Можно хранить состояния объекта в самом объекте, но тогда объект получится тяжёлым (+не надо грузить сохранением истории). Идея в том, чтобы вынести хранение старых состояний в отдельном объекте, который позволит вернуться к предыдущему состоянию в случае надобности.
Используется, если
- нужно возвращаться к предыдущему состоянию объекта
Преимущества:
- объекту не надо хранить свои предыдущие состояния
Недостатки:
- опекуном надо управлять (логика возвращения к предыдущим состояниям: какие-то снимки нужны, какие-то не нужны, как-то надо чистить, получается, расходуем зря память).

Шаблонный метод

Скелет какого-либо метода – мы делим публичный шаблонный метод на несколько этапов (protected-методов), вызываем их в шаблонном методе, а protected-методы реализуем отдельно, возможно подменяем их в полимрфных классах.

Состояние
Для каждого состояния выделяется свой класс. Такой подход используется редко, мы использовали другой – мы писали обработчики состояний

Свойство – шаблон, который даёт возможность предельно формализовать свойство объектов – геттеры и сеттеры
Проблемы:
- копирование объекта и копирование свойства, то есть в какой-то момент свойство может начать жить независимо от объекта

КМС – конечная модель состояний
Идея: Задать возможные переходы и осуществлять эти переходы
Паттерн состоит из 3 классов: переход, кмс, активный экземпляр