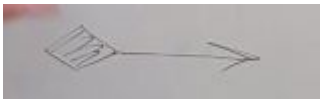
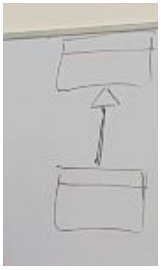
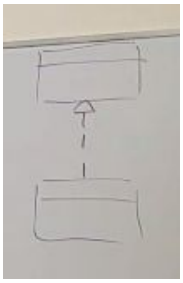
https://www.youtube.com/watch?v=OciYV27U_0g&list=PLF_E34yXYlPnM7jDyFd4WeWLlMvZRVxPB&index=1

Лекция 9

Связи uml:

1. Связи - при изменении одного объекта - это влияет на другой объект. 
2. Ассоциации:
   Существует связь между одним объектом и другим (один использует другой)
   
   Агрегация (Ставится ромбик на стороне объекта, который агрегирует) Когда ромбик не заштрихован это вариант, когда агрегирующий объект не отвечает за агрегируемый. (shared_ptr)
   
   Композиция их жизненные циклы совпадают
   
3. Обобщение. (Наследование с поддержкой полиморфных свойств) (Особенность: производный класс должен полностью поддерживать интерфейс базового класса (не расширять и не сужать))
   
   Реализация (Тоже наследование, но с условием, что производный класс может расширять базовый класс)



“ + ” означает, что это метод.

Шаблон - это конкретная реализация чего-либо.

Паттерн - это шаблон для решения какой-то задачи. Паттерн адаптируем к своей задаче.

Мы рассматриваем паттерное проектирование.

1. Порождающие паттерны:

Основная задача: создание объектов.

Фабричный метод:

Разнести на 2 задачи принятие решения какой объект мы будем создавать и непосредственно создание объекта. При создании объекта отвязаться от конкретного типа.

Выделяем базовый класс, задача которого порождать. И есть конкретные классы, которые будут создавать конкретный продукт.

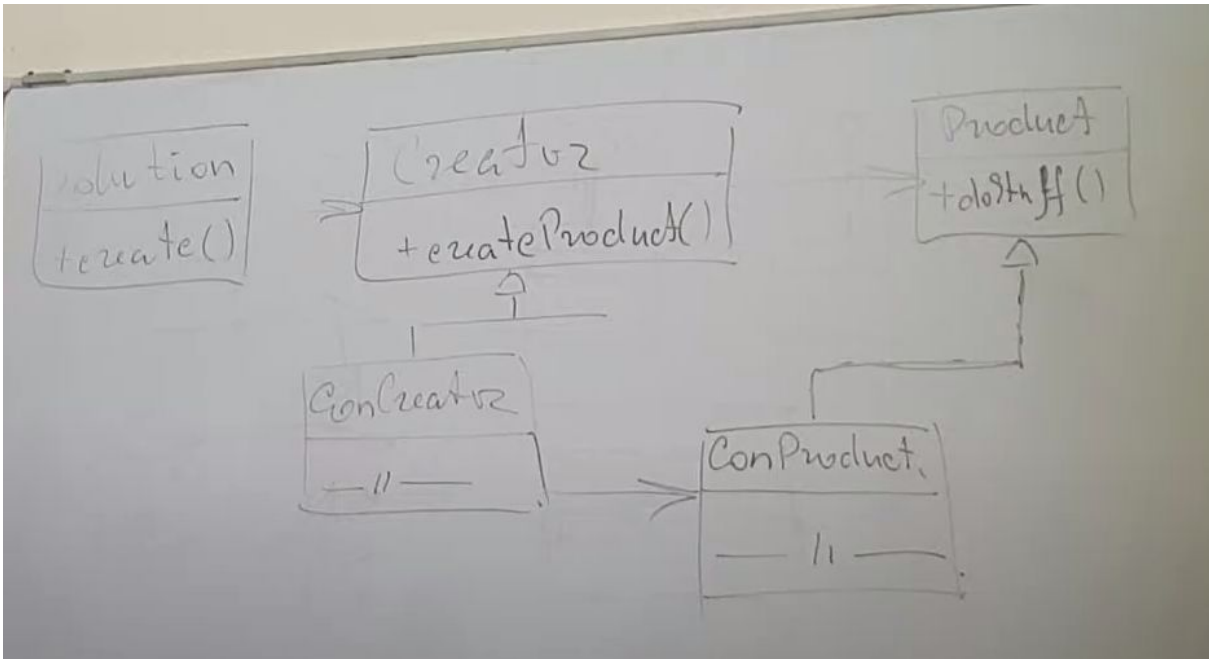
Основная задача, которая решается - это задача подмены одного метода другим.

Основан на полиморфизме (подмене)

Облегчается добавление новых классов.

Избавляем методы от привязки к конкретным классам.

Фабричный метод используем когда мы разделяем принятие решения о создании объекта и само создание.

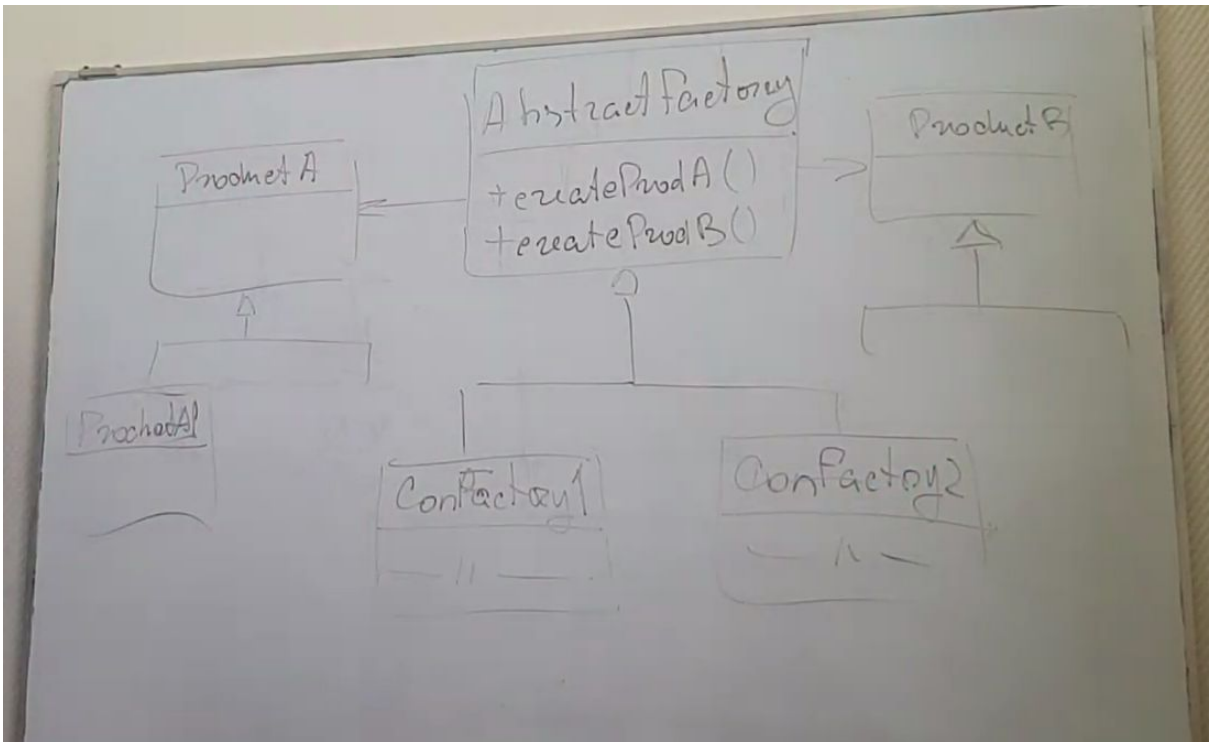


Фабричный метод создает продукты какой-либо одной группы. А если нужны продукты связанные между собой, то:

Абстрактная фабрика:

Задача: порождать семейство объектов. Разных объектов. (Абстрактная фабрика развитие фабричного метода с добавлением функционала, то что создаются объекты одного семейства)

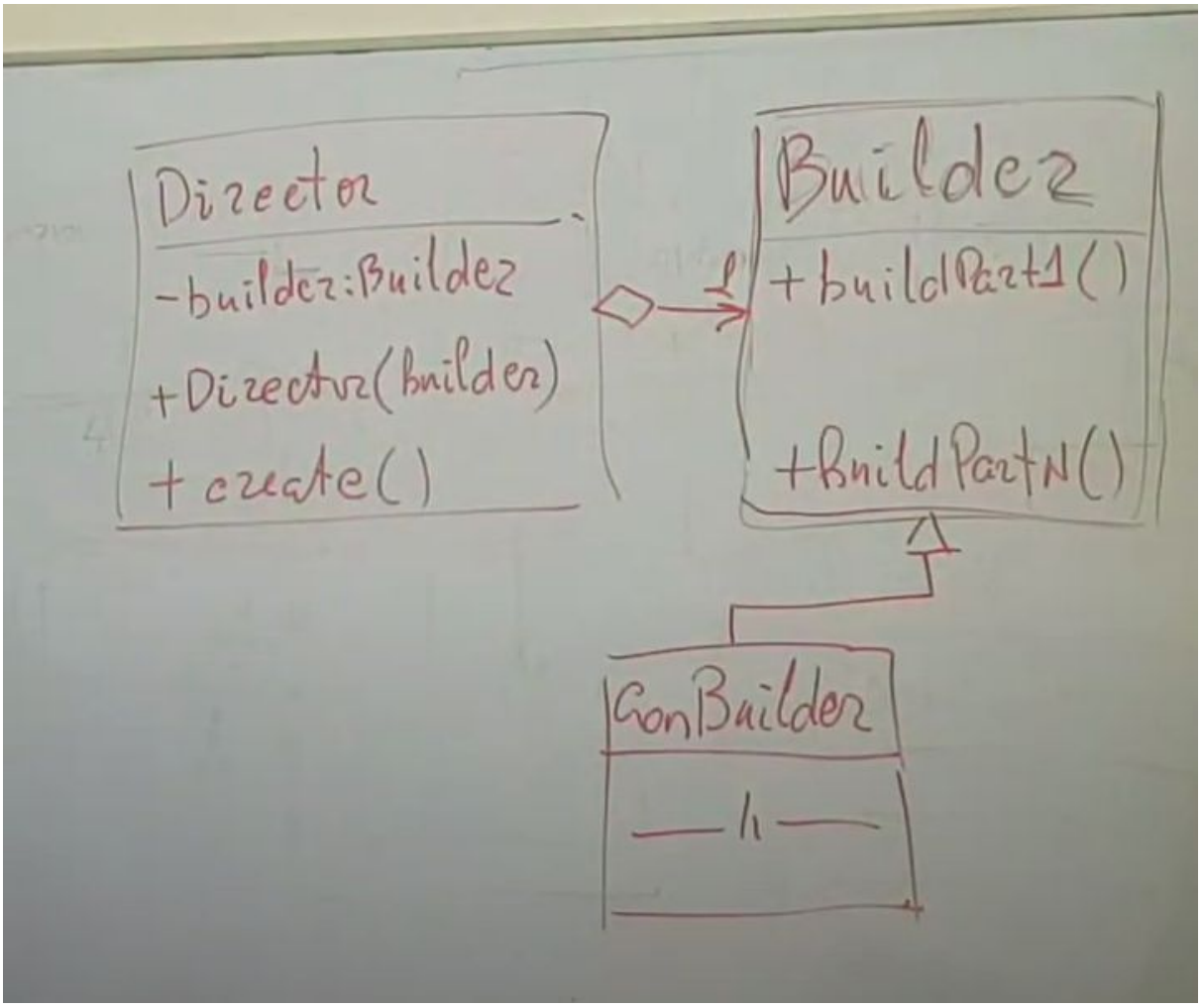
Использование такое же, как и в фабричном методе. Плюс в том, что невозможно создать объекты из разных семейств. Минус накладывает требования на продукты (В разных семействах должны быть представлены все продукты, которые определяет базовая Абстрактная фабрика).



Паттерн строитель:

Строитель - это класс, который включает в себя этапы создания объекта(Сложного объекта). Также мы выделяем еще один класс, объекты которого будут отвечать за контроль и контролировать за созданием объекта.

Строитель создает объект. A director контролирует создание объектов. Его задача пройтись по всем этапам, создать объект и проконтролировать, как выполняются эти этапы. Он должен подготовить данные для выполнения этих этапов и после того, как создастся объект отдать готовый объект. Идет разделение ответственности. Один создает, другой контролирует создание. Вся логика ложится на director (Он задает порядок создания и осуществляет контроль).



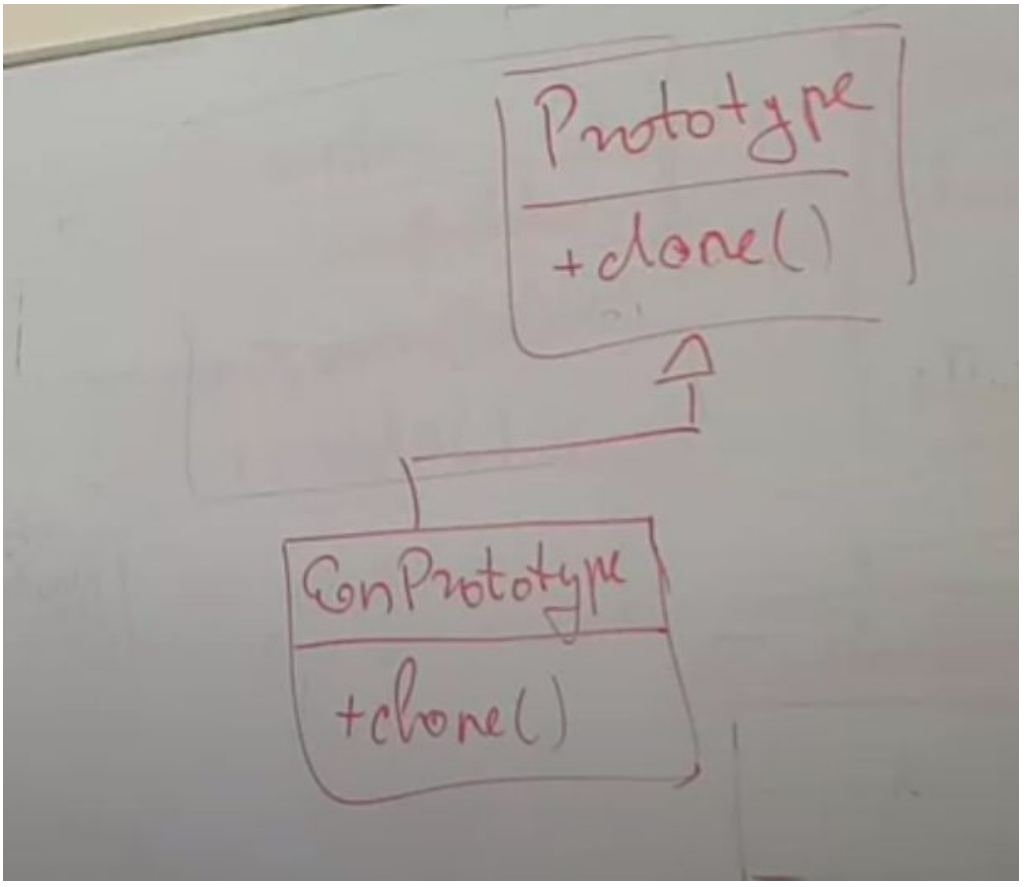
Используем: когда сложные объекты, которые создаются поэтапно.

Плюсы: Создание продукта пошагово. Вынесение создание и контроля в отдельные классы.

Минусы: Проблема с данными. Конкретные строители должны базироваться на одних и тех же данных.

Прототип.

В базовом классе добавляем метод clone, который возвращает указатель на себя. А производные классы реализуют этот clone под себя, возвращая уже указатель на подобный объект.



Одиночка.

(Лучше не использовать)

Лучше сказать, что это шаблон одиночка.

Возникают задачи, когда должен быть гарантированно создан только один объект какого-либо класса. Идея: Убрать конструкторы, чтобы извне нельзя было создать (убрать конструкторы из public части). Добавить static метод со static полем, которое будет создаваться только 1 раз.

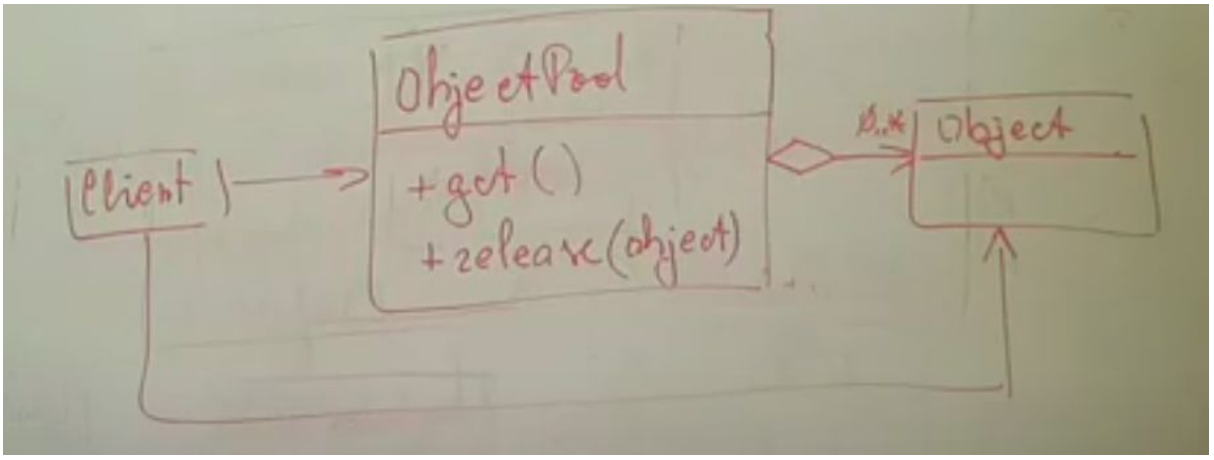
Недостаток: Глобальный объект и проблема с подменой(На этапе выполнения).

Пул объектов.

Предоставляет нам набор готовых объектов, которые мы можем использовать.

Держит объекты, может их создавать (расширяться) и по запросу отдает нам объект и может вернуть объект в пул (если он не нужен клиенту)

Для каждого объекта, который мы держим в пуле, нам нужно установить, используется или нет



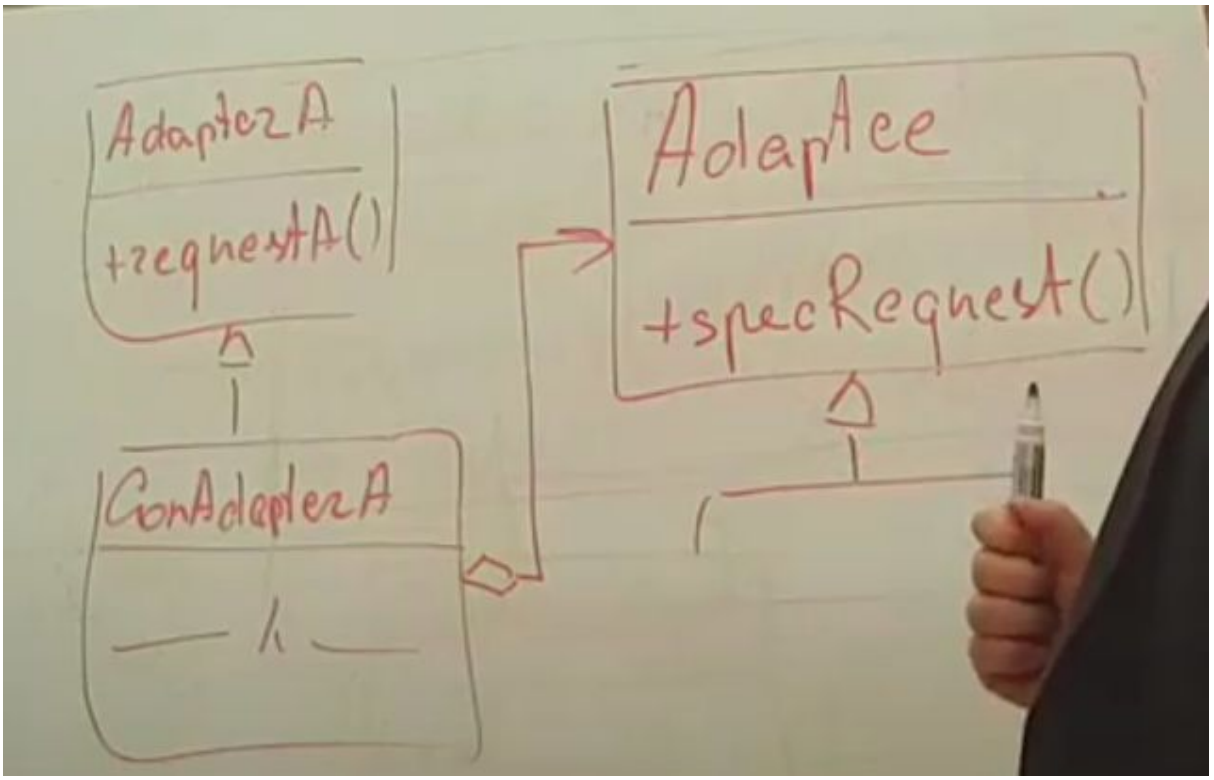
2. Структурные паттерны.

Предлагают структурные решения (декомпозиция классов) с использованием схем наследования, включения…

Проблема: Нам нужно создавать объекты, и эти объекты в разных местах программы мы используем по разному (объект выступает в нескольких ролях) (Получаем избыточный интерфейс) Один объект имеет несколько ответственностей. Используем паттерн:

Паттерн Адаптер.

Идея - подмена интерфейса. У класса был один интерфейс, мы этот интерфейс подменить другим интерфейсом, чтобы в зависимости от ситуации использовать этот объект по разному. Или же есть готовый класс и мы хотим этот класс встроить в нашу систему (Нужно адаптировать (подменить) интерфейс, который был у исходного класса.

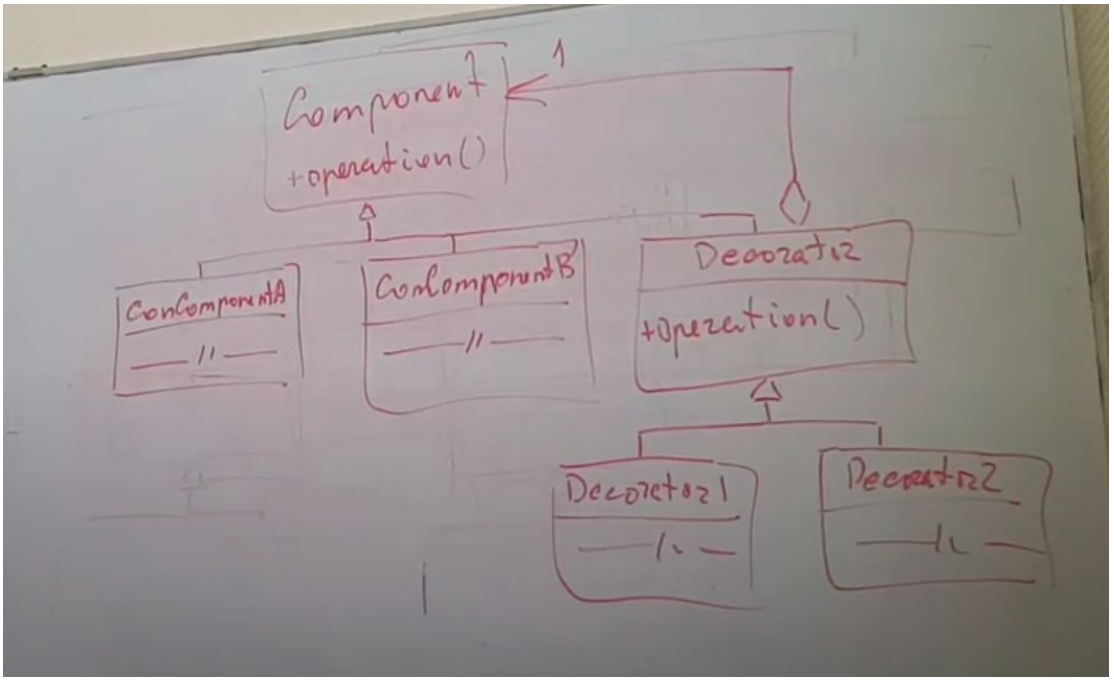


Плюсы: Позволяет нам класс с любым интерфейсом использовать в нашей программе. Позволяет не создавать нам классы с несколькими ответственностями. (Позволяет добавлять функционал, на основе имеющегося.)

Декоратор.

Позволяет добавлять какой-то функционал.
Хотим что-то добавить в объект А и В (одно и тоже) => Делаем производный класс, добавляя новое (Не расширяем интерфейс, а подменяем существующие методы.) Добавление выносим в отдельный класс декоратор. Создаем класс, который обладает таким же интерфейсом, но и еще имеет ссылку на базовый класс (Component) и от него мы уже можем порождать конкретные декораторы.
Плюсы: Избавляемся от большого кол-ва классов. Мы декорировать можем во время выполнения работы (программы) Сокращаем иерархию классов и динамически можем изменять поведение объектов, декорирую наши компоненты. И избавляемся от дублирования кода (Этот код уходит в конкретные декораторы).

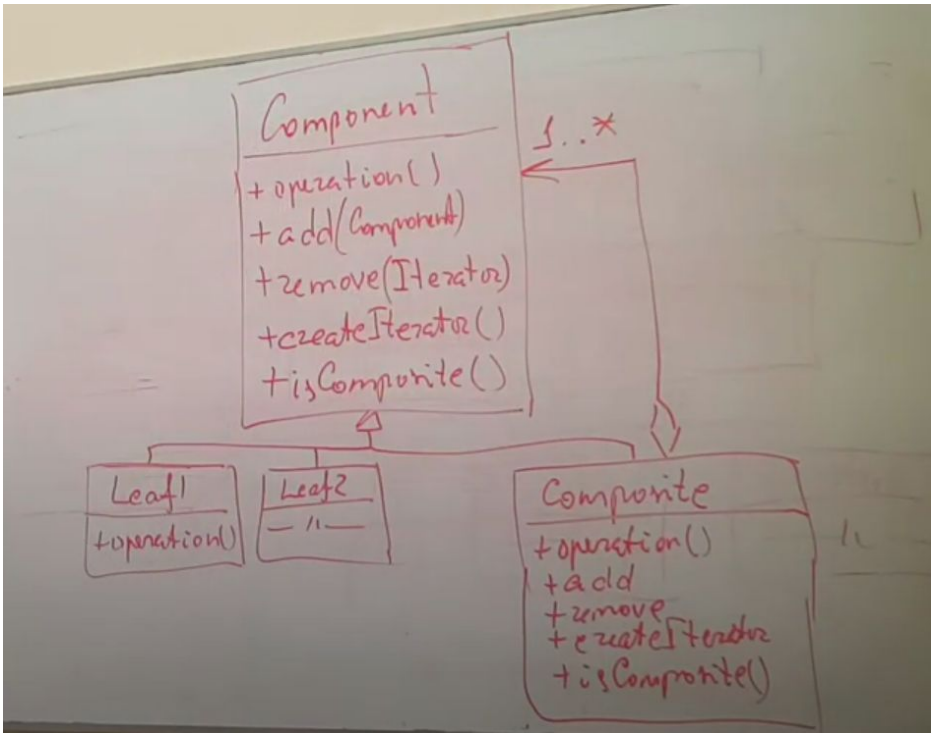
Проблемы: Сделали сложную обертку, нам убрать какую-то обертку проблематично. Заново приходится создавать компонент с обертками (Не можем просто вычеркнуть)



Компоновщик (Composite) (12 пример) (9 лекция 4 часть 13.30)

Часто мы работает с многими объектами однотипно (выполняем одни и те же операции). Возникают ситуации, когда нужно выполнять над группой объектов вот эти действия совместно.

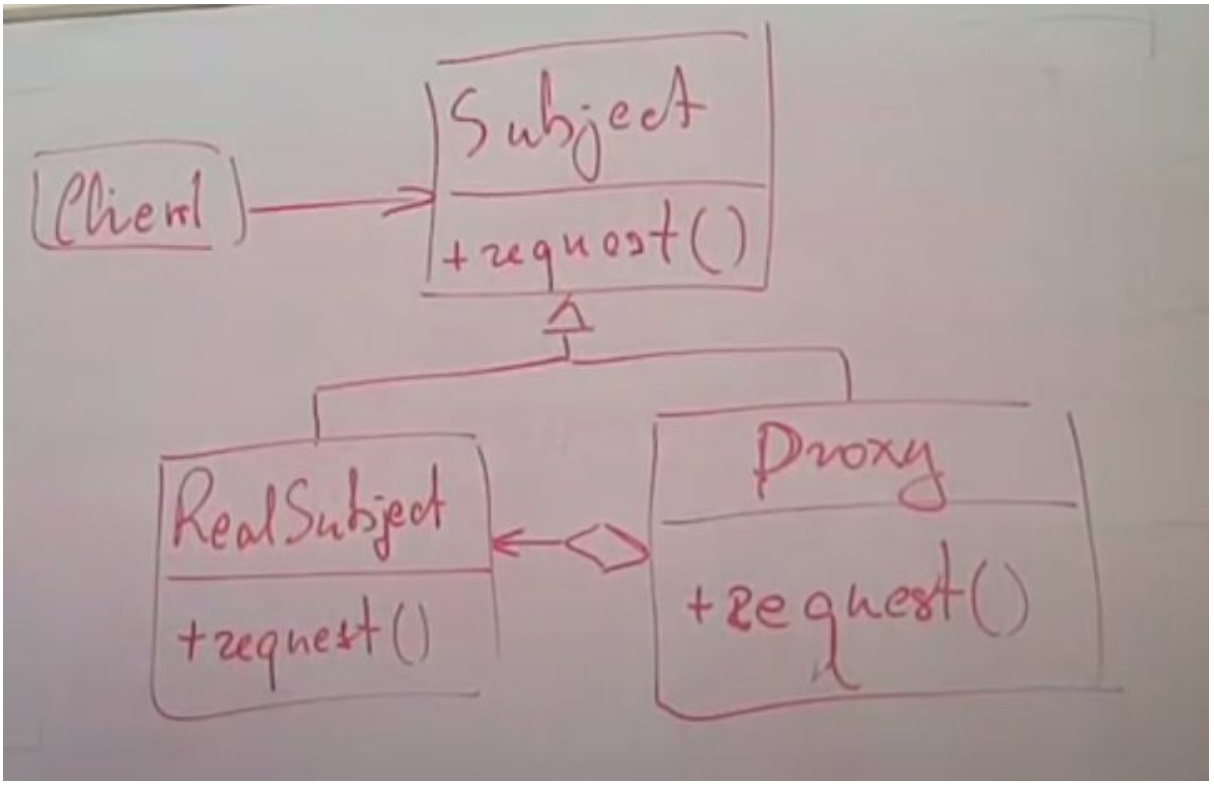
Компоновщик компонует объекты в древовидную структуру. В которой мы над всей иерархией объектов можем выполнять такие же действия, как и над простым элементом.



Плюсы: во время выполнения формируем сложные объект состоящий из подобъектов (компонентов). Выполнять над этой сборкой общие действия (Выполняется на этапе выполнения) Не нужно думать с кем мы работаем.

Заместитель (Proxy)

Позволяет нам работать не с реальным объектом, а с другим объектом, который подменяет реальный. Использовать можно: 1. изменять доступ (разные категории пользователей (защита)) 2. Может контролировать запросы. Proxy может сохранять предыдущий ответ.

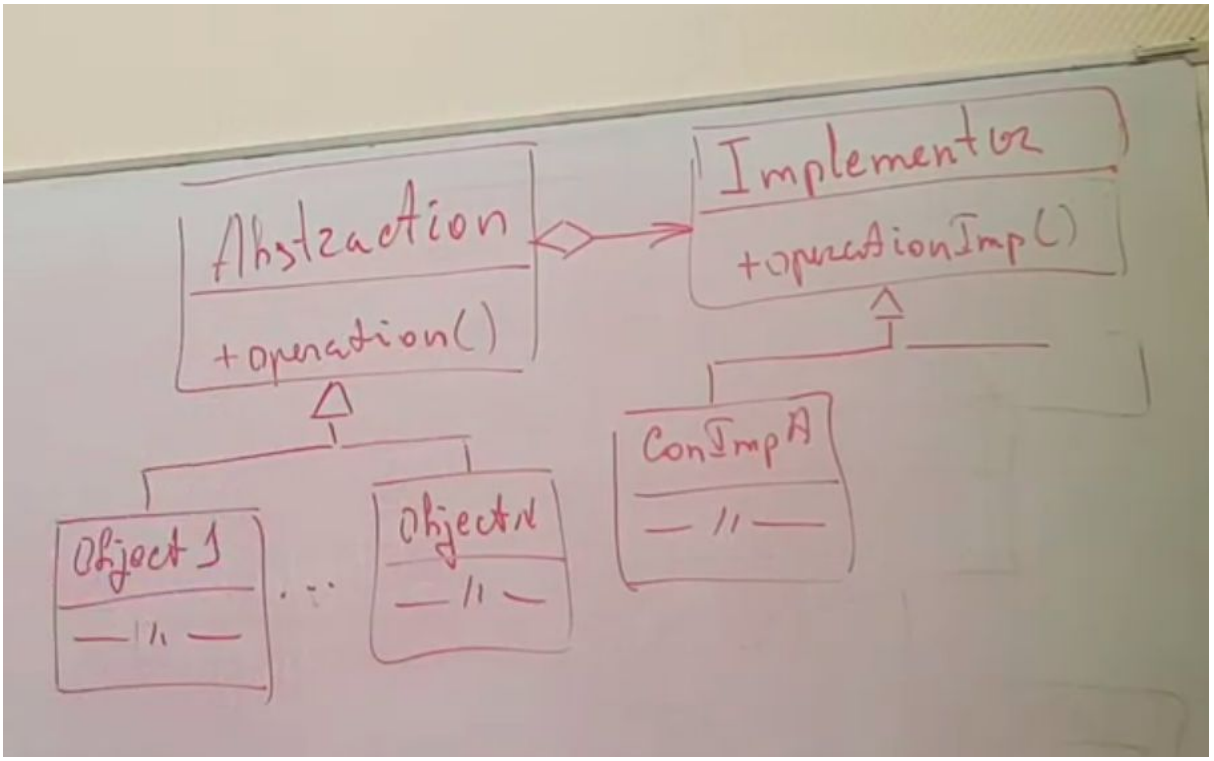


Плюсы: Можно контролировать какой-либо объект незаметно от клиента. Может работать когда объекта нет. Может отвечать за жизненный цикл объекта (может создавать, удалять)
Минус: Может увеличиться время к доступу объекта (т.к. доступ идет через прокси) Прокси должен хранить историю обращения (влияет на память).

Проблема: большая иерархия. и возможны несколько внутренних реализаций для объекта. Разные классы. разные ветви. Стоит задача во время работы поменять реализацию. Мигрировать от одного класса к другому. Дублирование кода, когда разрастается иерархия классов.
Избавляемся:

Паттерн Мост (Bridge)

Предложено: разделить понятие самого объекта, его сущности и его реализацию в отдельные иерархии. Мы сократим количество классов. И мы сделаем систему более гибкой. Во время работы сможем менять реализацию. Сможем частично уйти от повторного кода.
Отделяет сущность от реализации. Можем независимо менять логику. Наращивать реализацию.

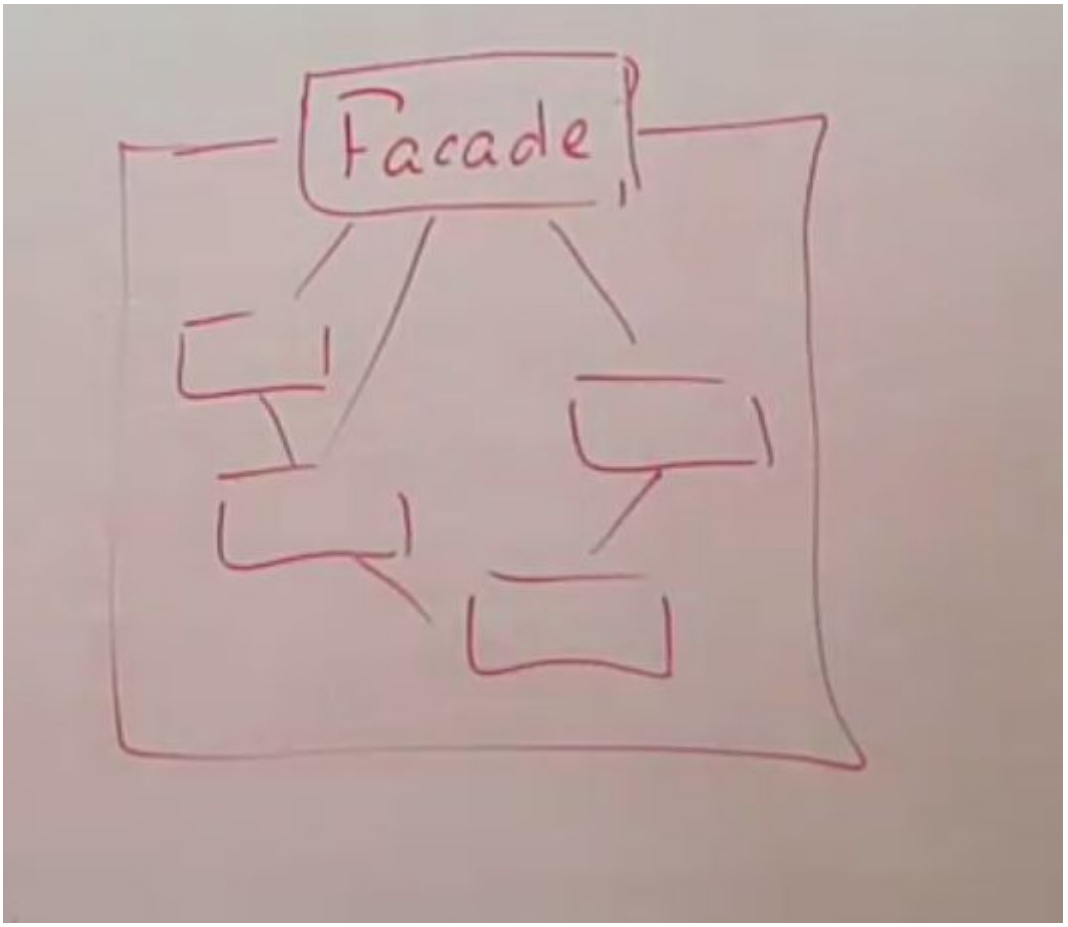


Используем: Когда во время выполнения нужно менять реализацию и когда большая иерархия и по разным ветвям идут одинаковые реализации. Независимо изменяем логику и реализацию.

паттерн фасад.

Имеем группу объектов с жесткими связями и чтобы извне не работать с каждым объектом по отдельности мы можем все эти объекты объединить в один класс - фасад, который будет представлять интерфейс со все этим объединением. Нам не нужно извне работать с мелкими объектами. Более

того фасад может следить за целостностью этой системы.

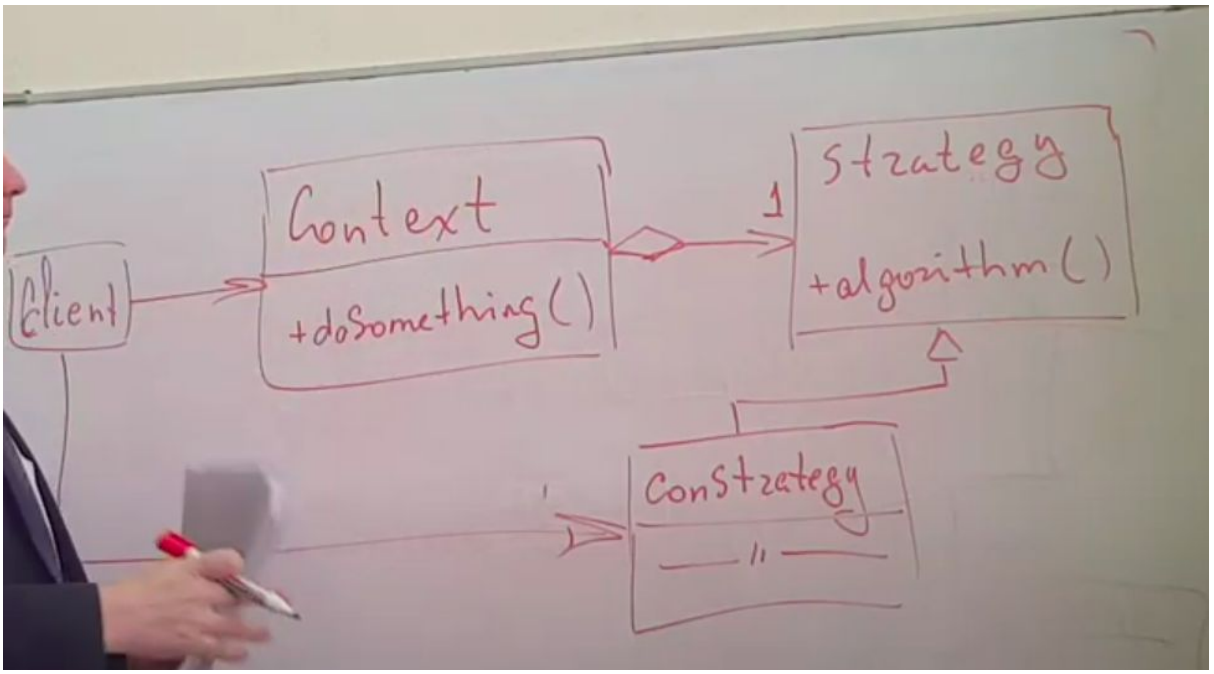


Уменьшается кол-во связей, за счет таких объединений. Клиенту не нужно уметь работать с мелкими объектами ему достаточно уметь работать с фасадом.

Шаблоны И Паттерны Поведения

Стратегия (Strategy).

Нам во время выполнения нужно менять реализацию какого-либо метода.
То, что изменяется - выносится в отдельный класс.
Определяет семейство всевозможных алгоритмов.



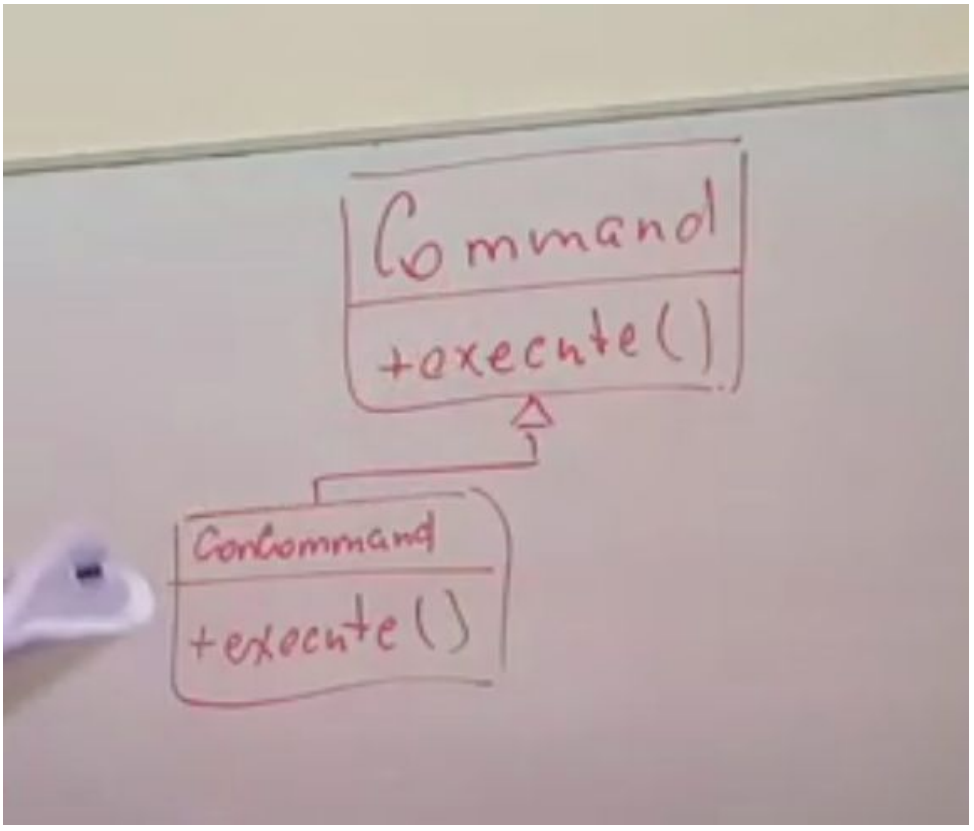
Обработка запросов.

Паттерн команда. (6 часть видео)

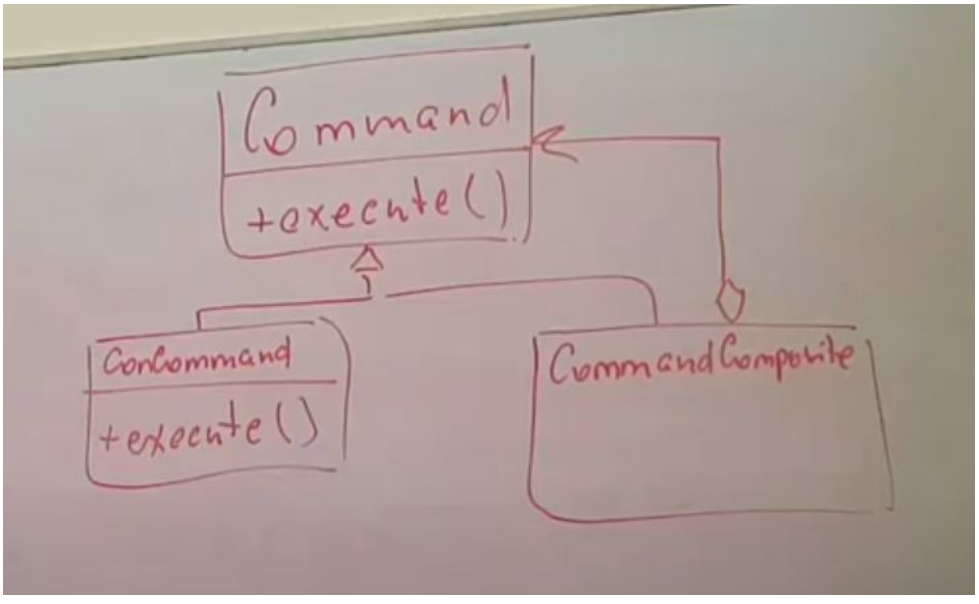
Идея: Обернуть каждый запрос в отдельный класс.

Запрос (команда) идет в виде объекта.

Имеем базовый класс команду. И имеем конкретные команды (в зависимости от того, что нам нужно). Команда несет данные и она может непосредственно нести что нужно сделать (Указатель на метод объекта, который нужно выполнить).



Плюсы: появление команды уменьшает зависимость между объектами (подсистемы могут общаться между собой командами (одна передает другой команду) не нужно держать связи между объектами, чтобы их вызвать) Сформировав команду мы можем выполнить ее через какое-то время (можем формировать очередь).
Если добавть этой команде композит, то мы можем формировать сложные команды из нескольких.

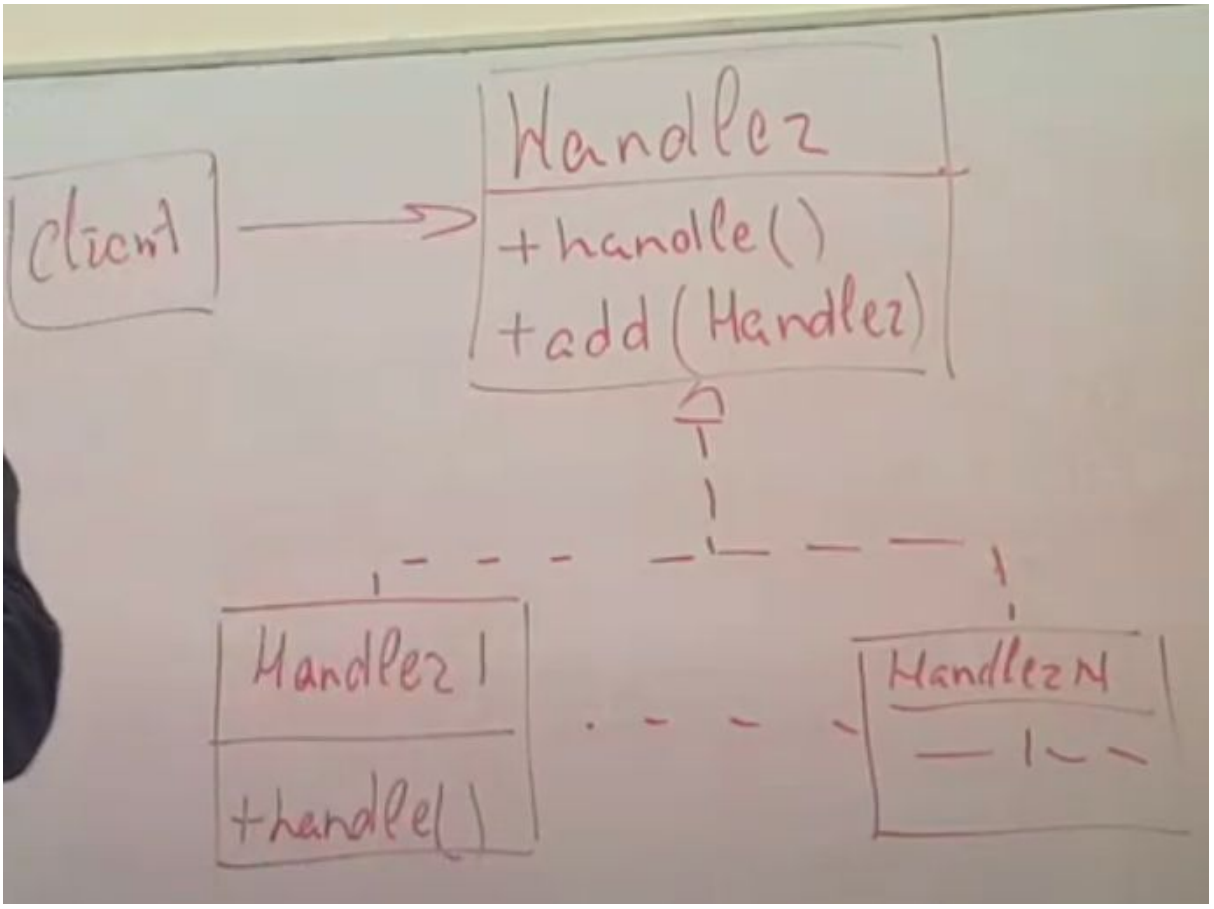


Цепочка обязанностей (Chain of Responsibility). (6 часть 8 минута)

Часто задача сводится к тому, что запрос должны обрабатывать несколько объектов.

Идея: создать цепочку обработчиков. Реализации цепочки может быть разная. (Пример 20 Рекуррентный вариант)

Позволяет передавать запрос по цепочке обработчиков. Каждый обработчик сам решает обработать или нет (может передать дальше по цепочке)



Handler - определяет общий интерфейс и задает механизм передачи запросов. В каждый Handler1, Handler2 содержат код обработки запросов. Используем, когда один и тот же запрос может выполняться разными способами. Когда у нас есть четкая последовательность в обработчиках(передавать по цепочке, пока он не обработается). На этапе выполнения мы формируем эту цепочку (решаем, какие объекты будут входить и их последовательность)

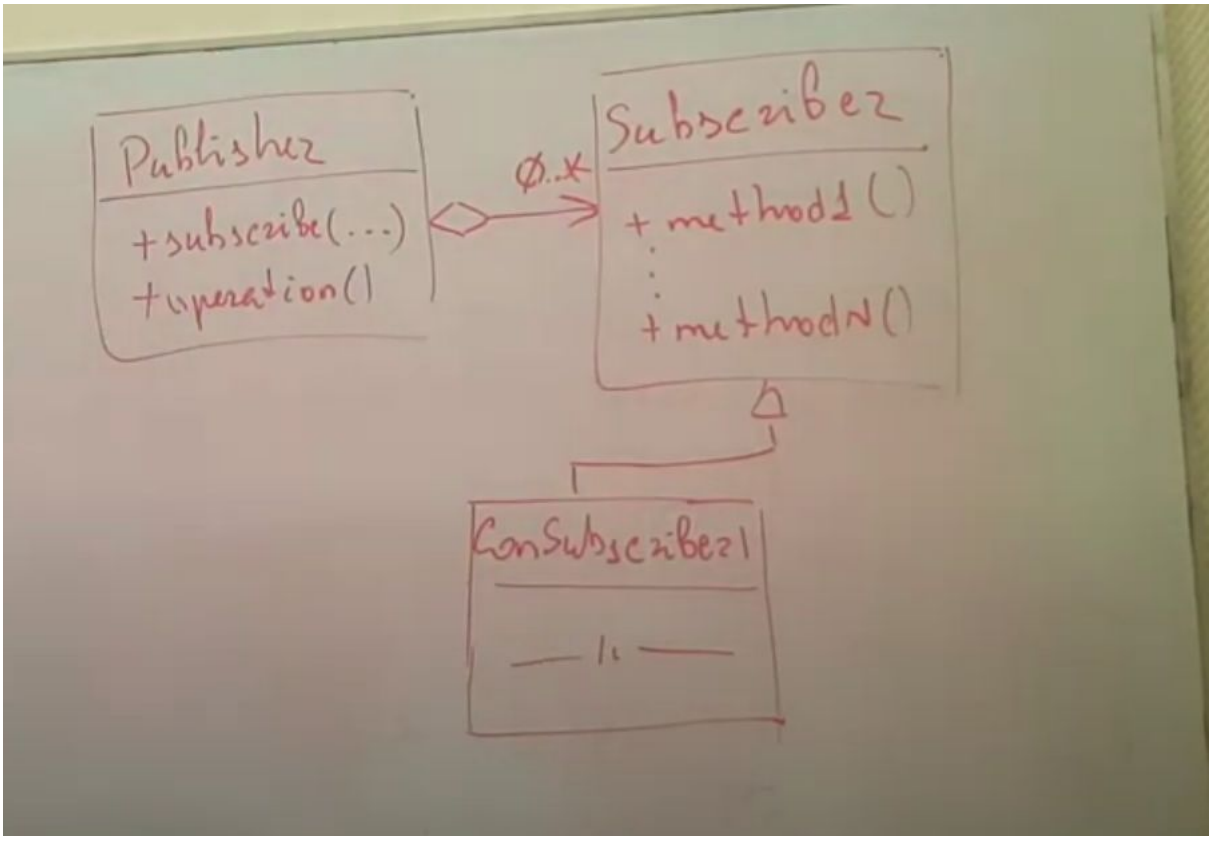
Паттерн подписчик издатель

Очень часто нам нужно запрос какой-либо передать не одному объекту, а многим и это должно выполняться на этапе выполнения.

Задается механизм - тот, кто хочет принять сообщение подписывается у издателя. Если подписался, то получает, если нет - то не получает.

Получаем механизм, когда группа объектов реагирует на изменение одного. Он своих подписчиков оповещает, когда происходит изменение, вызывая их методы.

Подписчик подписывается у издателя. И издатель держит список объектов, которые на него подписались.



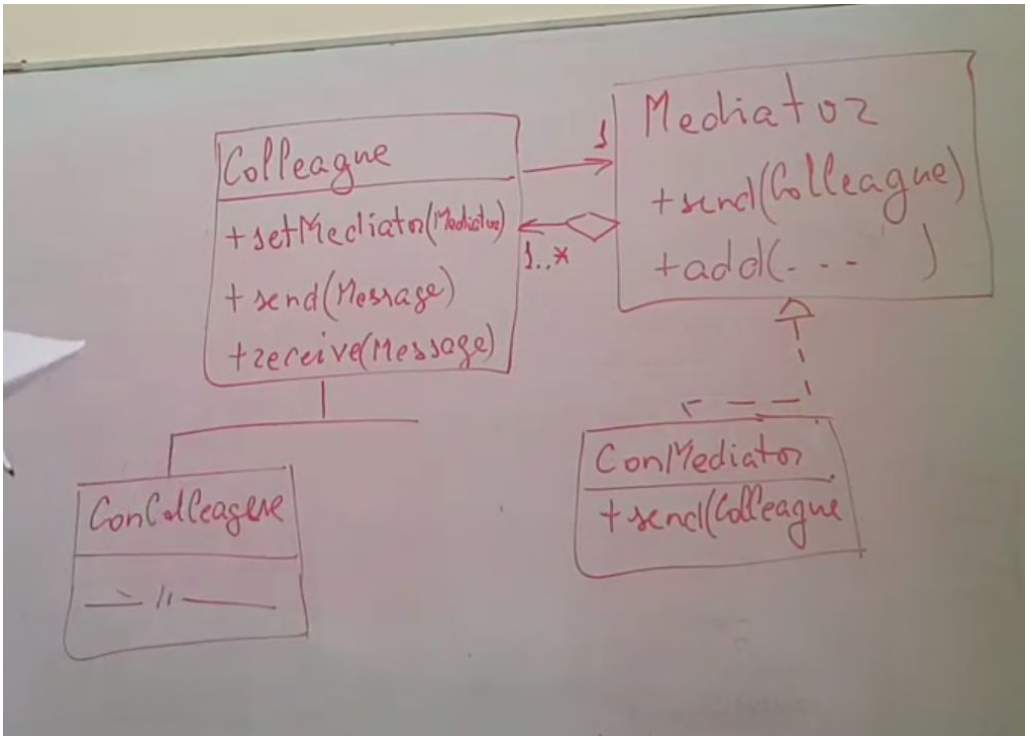
Удобно: Не делаем зависимыми издателей от подписчиков. Можем как подписаться так и отписаться (нужно делать итератор для того, чтобы отписаться).

Минус: Издатель должен держать список объектов, которые на него подписались. И нет порядка в оповещении подписчиков.

Паттерн посредник (mediator) 6 часть 30 минута.

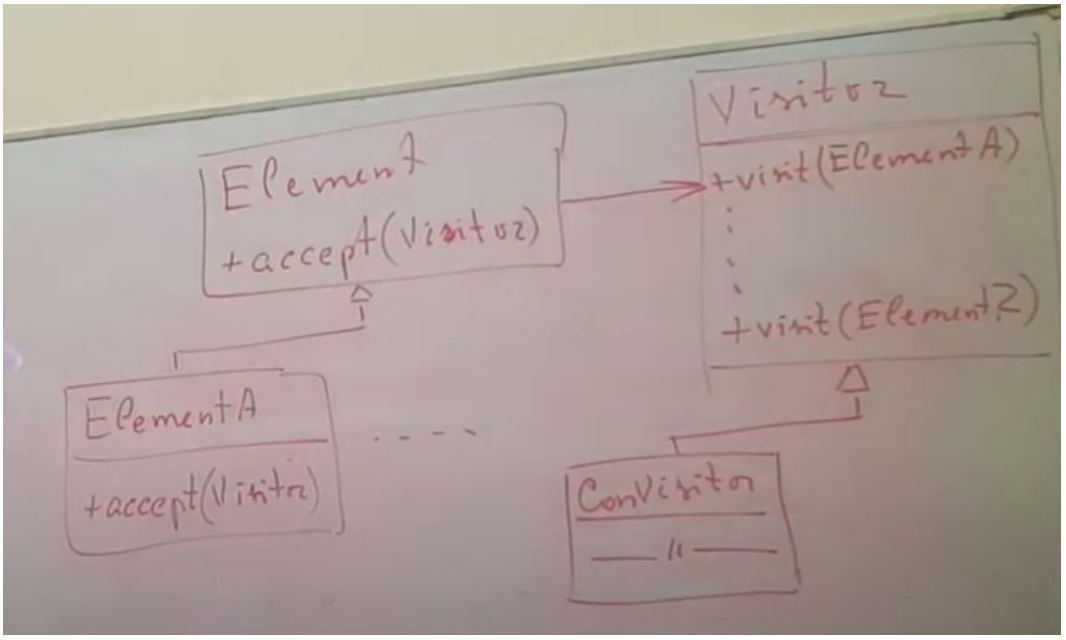
Позволяет уменьшить кол-во связей между объектами. Благодаря перемещению этих связей в посредника.

У объектов будет связь только с посредниками (между собой связи нет). Они могут бросить сообщение (send) или принять (receive) сообщение. Каждый из этих объектов (коллега) устанавливает с каким медиатором он работает. Но медиатор должен со всеми им работать, поэтому он содержит указатели на все эти объекты.



Паттерн Посетитель (Visitor) (22)

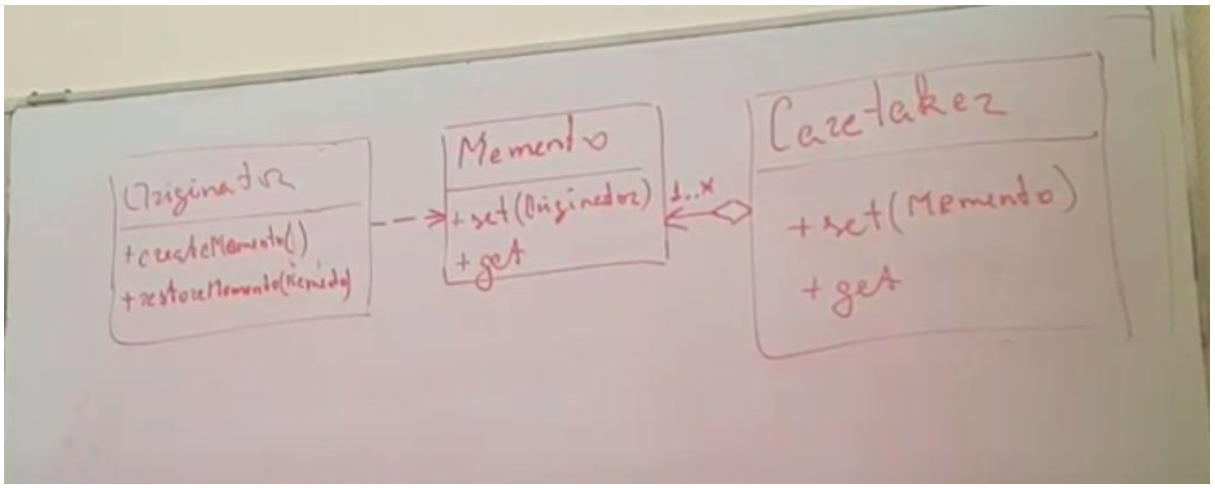
Проблема: изменение интерфейса объектов. Позволяет во время выполнения подменить или расширить функционал (адаптер решает изначально, а visitor позволяет сделать это на этапе выполнения).



Позволяет добавить ко всем объектам, и точно также добавить функционал не ко всем объектам. (Добавляем в базовый класс)
Плюс: в один класс сводим методы, относящиеся к одному функционалу.
Минусы: 1. Расширяется иерархия. Добавляются новые классы. Проблема связей на уровне базовых классов. 2. Может меняться иерархия (крайне редко)

Паттерн опекун. Memento (26) (8 часть 13 минута)

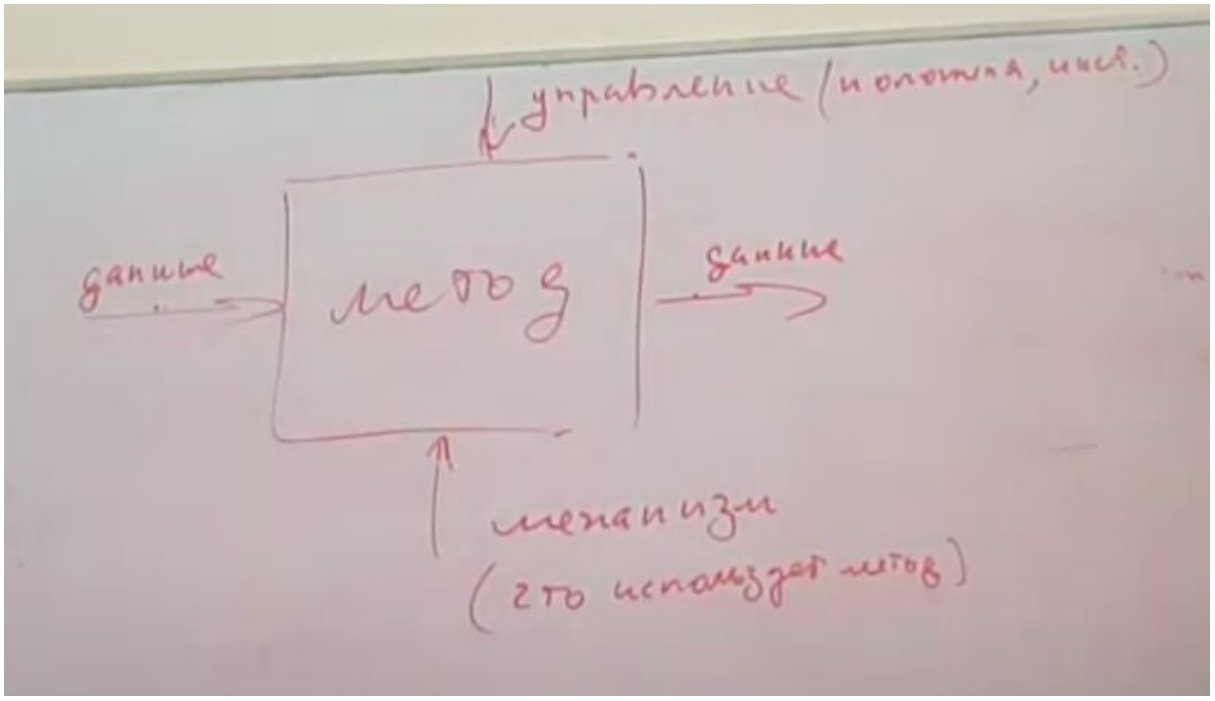
Когда мы выполняем какие-то операции, объект изменяется, но не всегда нас устраивают те изменения, которые произошли и мы хотим вернуться к предыдущему состоянию нашего объекта (сделать откат). Можно сохранять состояния. Но возлагать это на сам объект не нужно (будет слишком тяжелый)
Идея: выделить эту обязанность другому объекту, задача которого хранить предыдущие состояния объекта. И если нужно откатываться.



Используется, когда нужно возвращаться на какие-то стадии.
Плюсы: Самому объекту не нужно хранить свои состояния предыдущие.
Минусы: Опекуном нужно управлять. Много памяти захватываем.

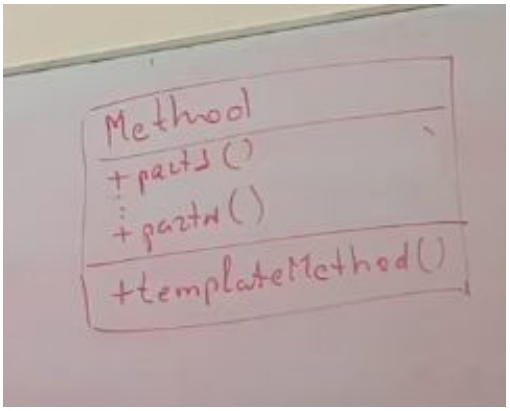
Паттерн Шаблонный метод.

Является скелетом какого-либо метода. Мы любую задачу разбиваем на этапы (метод). Шаги, которые мы выполняем, для того чтобы исходные данные преобразовать в результат, который нам нужен. (**IDEF0**).



Задача, реализовать в себе класс, который в себе будет формализовывать вышеприведенное понятие.

Любой метод мы разбиваем на шаги преобразования. IDEF диаграмма у нас как раз разбита на шаги.
Метода на большое количество этапов не разбивается (не больше 5)
part1...partn - этапы.
templateMethod - шаблонный метод.
Формируем класс, который отвечает за метод.

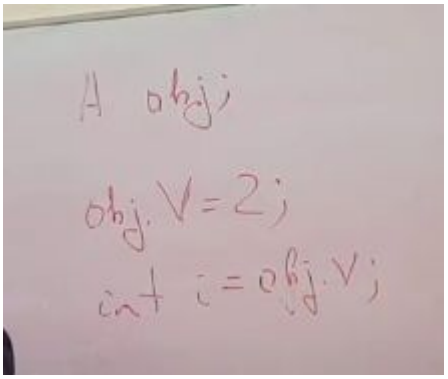


Состояние (23)

Идея: объект может находится в разных состояниях и каждое состояние формировать как класс, которое отвечает за каждое определенное состояние.
(Этот паттерн активно используют, но мы не будем)

Свойство (21)

Очень специфический. Дает возможность формализовать свойство.
В современных языках есть понятие свойство:
К примеру есть класс obj и у него есть свойство V (открытые поля данные). Но поля должны быть закрыты (Ниже приведенный доступ не должен быть доступен)



Этот доступ осуществляется через методы. Один метод устанавливает set, а другой метод возвращает get.
Мы не можем сделать V открытым членом. Если рассматривать V как объект, то для этого объекта мы должны перегрузить оператор присваивания. Нам нужен класс, который имеет перегруженный оператор = и оператор приведения типа. Этот перегруженный оператор должен вызывать метод set, а оператор приведения типа вызывать должен метод get. Удобно создать шаблон свойство. Первый параметр, это тип объекта для которого создается этот шаблон (Т.е. А), а второй это тип V (Т.е. int)

Проблемы: очень часто этот паттерн не любят. Он имеет подводные камни: Проблема возникает в копировании. При некорректном использовании у нас может свойство отделиться от объекта.

Лр:

Решить проблему с замещением одного объекта на другой. С расширением функционала. Добавление нового функционала. Причем чтобы изменений в коде были минимальные.

Мы разбиваем задачу на 2 домена, это домен интерфейса и прикладной домен (Сама наша задача). Интерфейсный домен нас не интересует.

Отрисовку делать на уровне прикладного домена.

Использовать паттерн **фасад**, который от нас будет скрывать (мелкие детали) те объекты, которые мы используем в нашей задаче (это сцена, менеджеры управления сценой, другие вспомогательные объекты)

Проектирование начинаем с физических объектов, которые реально существуют (в физическом мире существует объект, который мы хотим поворачивать)

Мы работаем с каркасной моделью, но в принципе мы должны заложить в архитектуру нашу модель, что в дальнейшем мы можем подменить объект (можем использовать например полигональное представление модели, может появиться понятие материала).

Мы должны рассматривать нашу систему так, чтобы было легкое подмена одного объекта на другой (одного типа на другой тип), развитие системы. У нас есть сцена, на сцене могут быть объекты: геометрическая модель, наблюдатель(камера), источник света. Лучше дать возможность добавления источника света (но не реализовывать), а вот камеру удобно реализовать. Камер и фигур может быть несколько. Мы можем из этих геом. фигур образовывать сборку. Мы можем связывать камеру с геометрической фигурой (даже не с одной, а с несколькими)

Перемещать можем как камеру, так и саму модель. Одну модель или в совокупности несколько моделей.

Сценой нужно управлять, так что удобно реализовывать менеджеры, которые имеют свою обязанность, кто-то управляет объектами на сцене, кто-то загружает сцену… (в зависимости от того, какая задача нужна)

Хотелось бы чтобы касаемо замены было легко подменить одну графическую библиотеку на другую, замена хранилища, замена представления данных (объекта, камеры). Добавление функционала (строим каркасную модель, в дальнейшем нужно убрать невидимые грани)

Один объект выполняет одну обязанность.

Реализовывать так, чтобы можно было добавить новые сущности и расширить функционал. Мы это делали за счет адаптера или паттерна визитер.

Сделать диаграмму. А потом переходить к кодированию.