**Билеты ООП (ИС y25, 3 семестр)**

В билетах не все паттерны, но лучше выучить все (особенно абстрактную фабрику)

Паттерны поведения:

1. Паттерн состояние. В чем идея? Какое ключевое отличие от паттерна стратегия?
2. Паттерн стратегия. В чем идея? Какое ключевое отличие от паттерна состояние?
3. Паттерн шаблонный метод. В чем идея? В чем разница между паттернам фабричный метод и шаблонный метод?
4. Паттерн посредник. В чем идея? Когда следует применять?
5. Паттерн цепочка обязанностей. В чем идея? Когда следует применять?
6. Паттерн наблюдатель. В чем идея? Когда следует применять?
7. Паттерн посетитель. В чем идея? Какие требования предъявляются к классам, которые можно посещать?
8. Паттерн команда. В чем идея? Как его можно комбинировать с партнёром снимок?
9. Паттерн снимок. В чем идея? Как его можно комбинировать с партнёром команда?

Порождающие паттерны:

1. Паттерн фабричный метод. В чем идея? В чем разница между паттернам фабричный метод и шаблонный метод?
2. Паттерн одиночка. В чем идея? В чем опасность использования?

Структурные паттерны:

1. Паттерн легковес. В чем идея? Когда его следует применять?
2. Паттерн заместитель. В чем идея? Какие виды (типы) заместителей бывают? Для решения каких задач применяются?
3. Паттерн декоратор. В чем идея? Когда следует применять?
4. Паттерн адаптер. В чем идея? Когда следует применять?
5. Паттерн мост. В чём идея? Когда следует применять?
6. Паттерн фасад. В чем идея? Когда следует применять?
7. **В чем идея инкапсуляции? Абстракция, интерфейс и реализация. Средства защиты и доступа.**

**Инкапсуляция** – каждый класс (блок системы) рассматривается как черный ящик. Пользователю известен интерфейс(список методов и свойств), но не как он реализован внутри. Еще инкапсуляция про объединение всех свойств, действий, в один объект (капсула)

**Защиту** обеспечивает сокрытые данных с помощью модификаторов доступа, полиморфизма ( переопределение методов из родительского класса)

**Абстракция** – набор значимых характеристик объекта, которые отличают его от остальных. Значимые характеристики выделяют с точки зрения задачи, которую решает конкретная архитектура.

**Интерфейс** - абстрактный класс, у которого ни один метод не реализован, представляет контракт взаимодействия для все классов, которые наследуют интерфейс

**Реализация** - отношение между классами, при котором один класс определяет обязательство (интерфейс, абстрактный класс, просто класс), а другой гарантирует его выполнение (наследник).

1. **В чем идея наследования? Простое, многоуровневое, множественное наследование.**

**Наследование** - возможность порождать один класс(класс-наследник) от другого с сохранением всех свойств и методов класса-предка(базовый класс), добавляя при необходимости новые свойства и методы. Выражает собой очень сильную связь между двумя классами.

Виды наследования:

* **Простое –** «одиночное» наследование, производный класс имеет только одного родителя
* **Многоуровневое –** наследование от производного класса: человек -> учитель -> учитель математики
* **Множественное –** класс имеет несколько родительских классов

1. **В чем идея полиморфизма? Раннее и позднее (динамическое) связывание.**

**Полиморфизм** — это возможность использовать классы – наследники в контексте, который был предназначен для базового класса.

**Идея полиморфизма** заключается в способности объектов производных классов по-своему обрабатывать запрос, свойственный всем этим производным классам.

**Раннее связывание (ad-hoc) –** способность объекта класса обрабатывать в одних и тех же методах разные типы данных на этапе компиляции. Реализуется перегрузкой методов:

**Изображение выглядит как текст

Автоматически созданное описаниеПараметрический полиморфизм** - перегрузка метода заменяется его шаблонированием, но для разных типов поведение будет одинаковым.

**Полиморфизм подтипов** (динамический полиморфизм, сильное связывание) - полиморфизм в его классическом виде. Возможность объектов с одинаковым интерфейсом реализовывать методы по своему. (переопределение методов из базового класса)

1. **Принцип “информационного эксперта”. В чем идея? С каким принципом SOLID коррелирует?**

Базовый принцип разделения ответственности. Ответственность на себя должен брать тот, кто обладает максимумом информации для исполнения.

Информация должна обрабатываться там, где она содержится.

Например, есть магазин, делается заказ, цена должна подсчитываться не в клиенте, а заказе

[Пример](https://habr.com/ru/company/otus/blog/491636/)

Корреляция идет с принципом OCP из SOLID - информационный эксперт способствует увеличению инкапсуляции и стимулирует возможность переиспользовать компоненты, что помогает системе быть “открытой к расширению, и закрытой для модификации”

1. **В чем заключается смысл принципов низкой связности и высокого зацепления (Low coupling & High cohesion)**

**Low coupling** (низкая связность) заключается во взаимной независимости классов друг от друга.

Класс со «слабым» зацеплением:

- Имеет слабую зависимость от других классов;

- Не зависит от внешних изменений (изменение в одном классе оказывает слабое влияние на другие классы);

- Прост для повторного использования

**High cohesion** (высокая зацепленность) заключается в необходимости соблюдения однородности бизнес-логики в каждом конкретном классе (к примеру, банк не должен делегировать обязанности по созданию кредитного счета отделению; отделение - это интерфейс банка для конечного пользователя)

Класс должен стараться выполнять как можно меньше не специфичных для него задач, и иметь вполне определенную область применения

Эти принципы взаимно дополняют друг друга и в качественном коде используются вместе

1. **В чем заключается смысл принципа подстановки Лисков? В каком случае и как нарушается принцип в проблеме квадрата-прямоугольника?**

Класс-наследник и базовый класс должны быть взаимозаменяемы, клиент не знает какой из классов использует.

Условия для выполнения принципа подстановки Лисков:

* 1. Условия не должны быть усилены в наследнике
  2. Они также не должны быть ослаблены
  3. Инвариант базового класс сохранен в наследнике

Проблема квадрата и прямоугольника происходит при наследовании квадрата от прямоугольника по принципу «квадрат это прямоугольник». Это так, однако с точки зрения математики, а с точки зрения поведения это разные объекты.

То есть иерархии нужно строить основываясь не на том, чем фактически является объект, а от поведений объекта.

1. **SOLID**

**S** – single responsibility. У одного класса только одна ответственность, и все его свойства, методы, должны быть направлены на выполнение этой ответственности

**O** – open closed principle. Система должна быть открыта для расширения, но закрыта для модификации. Расширение системы через дописывание кода: полиморфизм (новые имплементации интерфейса), а не через изменение старого

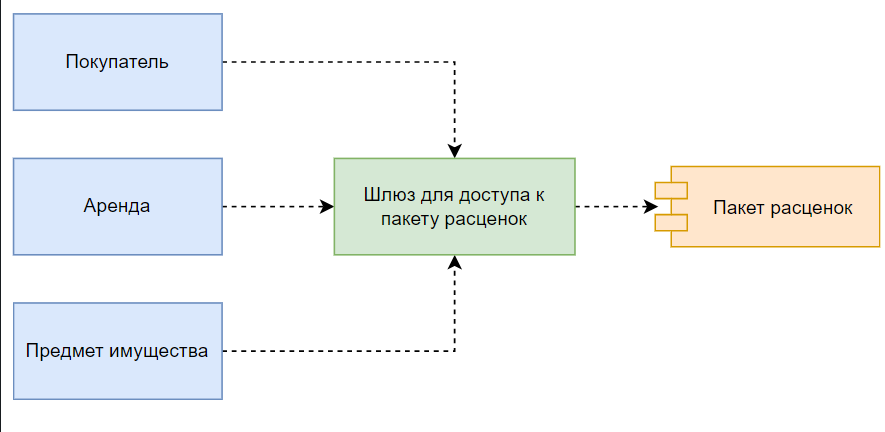
**L** – Liskov substitution (подстановка лисков). Класс-наследник и базовый должны быть взаимозаменяемы

**I** – interface segregation (разделение интерфейса). Не нужно делать один большой интерфейс для всего, лучше много, но под конкретную свою задачу. Нарушение ведет к зависимости от изменений всех методов интерфейса.

**D** – dependency inversion. Модули более высокого уровня не должны зависеть от модулей ниже. Детали зависят от абстракций

1. **Написала все в 7 вопросе чтобы было в цвета флага лгбт**
2. **Gateway и Mapper. В чём идея? В чём ключевые различия?**

**Gateway** (шлюз) – объекты могут обращаться не напрямую к внешнему ресурсу (бд, xml, какой-то api, интерфейсы которых неудобны), а к шлюзу, который будет посредником между неудобным интерфейсом и другими объектами. Так же если решат сменить неудобный объект, благодаря gateway его будет легко сменить



**Mapper** – отделяет друг от друга разные части системы, которые должны быть независимы. Они не должны знать о mapper (прозрачный посредник).

А еще он посредник между внутренними компонентами ( внешние это бд например )

**Паттерны**

Тут расписаны не все паттерны из билетов, но для нормального понимания работы все равно придется читать [metanit](https://metanit.com/sharp/patterns/) и [refactoring guru](https://refactoringguru.cn/refactoring) (с впн), билеты можно использовать только как краткий конспект

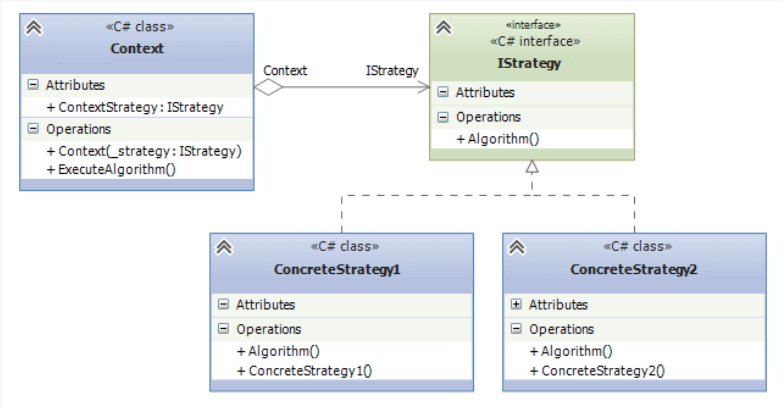
**Стратегия**

Проблема: нужно обеспечить взаимозаменяемость алгоритма в клиенте, добавление новых алгоритмов не должно затрагивать класс, где он должен использоваться (ocp)

Стратегия инкапсулирует разные поведения(алгоритмы) с возможностью подмены.

IStrategy - интерфейс для алгоритма

Сontext – класс клиент, который использует алгоритм



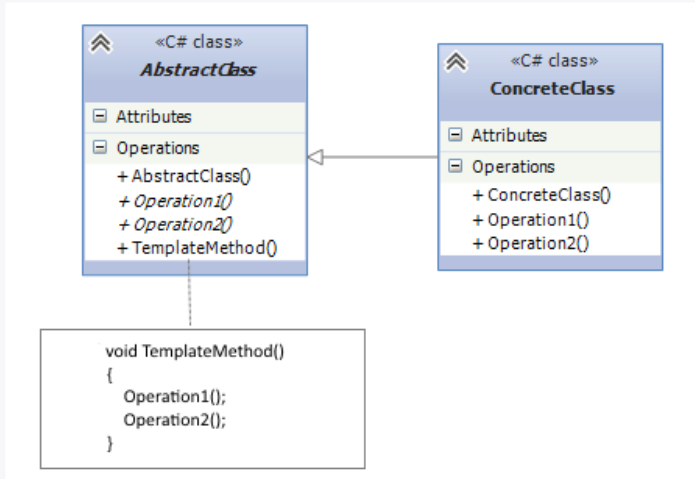
Отличие от состояния: оба паттерна используют композицию чтобы менять состояние объекта, делегируя работу вложенному объекту. Но в Стратегии объекты не знают друг о друге, они никак не связаны. В состоянии конкретные реализации могут друг друга переключать.

**Шаблонный метод**

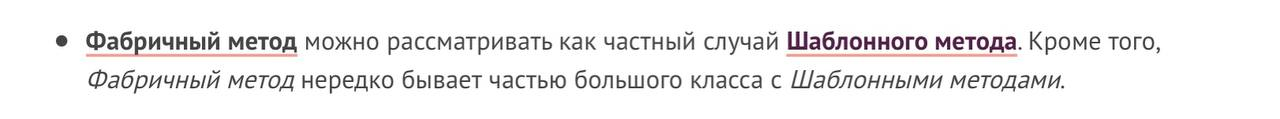
Проблема: есть несколько классов, в алгоритме которых используется одна и та же идея, но некоторые шаги отличаются.

* Когда планируется, что в будущем подклассы должны будут **переопределять различные этапы** алгоритма без изменения его структуры
* Когда в классах, реализующим схожий алгоритм, происходит **дублирование кода**. Вынесение общего кода в шаблонный метод уменьшит его дублирование в подклассах.

Шаблонный метод – каркас, определяющий основу алгоритма, разбивает его на несколько шагов. При это он позволяет наследникам переопределять некоторые шаги алгоритма, но структура остается прежней.

 **AbstractClass**: определяет шаблонный метод **TemplateMethod**(), который реализует алгоритм. Алгоритм состоит из последовательности вызовов других методов, часть из которых, может быть абстрактными и должны быть переопределены в классах-наследниках. При этом сам метод **TemplateMethod**(), представляющий структуру алгоритма, *переопределяться не должен.*

**ConcreteClass**: подкласс, который может переопределять различные методы родительского класса.



**Состояние**

Проблема: В коде методов объекта используются многочисленные условные конструкции, выбор которых зависит от текущего состояния объекта. Такое тяжело поддерживать, потому что могут добавляться новые состояния

Объект может находится в одном из нескольких состояний, набор которых (а также переходов между ними) конечен

Паттерн предлагает создать отдельные классы для каждого состояния (stateA and stateB) и вынести туда поведения, соотв. этим состояниям.

Изображение выглядит как стол

Автоматически созданное описание

**Main (context)** - объект, поведение которого должно динамически изменяться в соответствии с состоянием. Выполнение же конкретных действий делегируется объекту состояния. Хранит ссылку на состояние, в котором щас находится.

State – объявляет действия-методы, которые должны реализовать конкретные состояния. Можно одним из полей сделать Context, чтобы передавать его другим состояниям

Состояние — это надстройка над стратегией. От стратегии отличается тем, что клиент(контекст) и сами состояния знают друг о друг и могут инициировать переход между ними.

**Посредник**

Проблема: Разные классы должны взаимодействовать друг с другом, но прописывать эту логику взаимодействия внутри класса не вариант (создается тесная связь)

* Когда имеется множество взаимосвязаных объектов, связи между которыми сложны и запутаны
* Когда необходимо повторно использовать объект, однако повторное использование затруднено в силу сильных связей с другими объектами.

Паттерн Посредник (Mediator) обеспечивает взаимодействие множества объектов без необходимости ссылаться друг на друга, а с помощью объекта-посредника, который будет перенаправлять запросы куда надо. Тем самым достигается слабосвязанность взаимодействующих объектов, они зависят только от посредника

**Компоненты** – разнородные объекты, каждый хранит ссылку на посредника, но работает только через интерфейс Imediator.

**Посредник** – определяет интерфейс для обмена информацией между компонентами. Метод notify оповещает компоненты об изменениях.

**Конкретный посредник –** определяет взаимодействие компонент между собой.

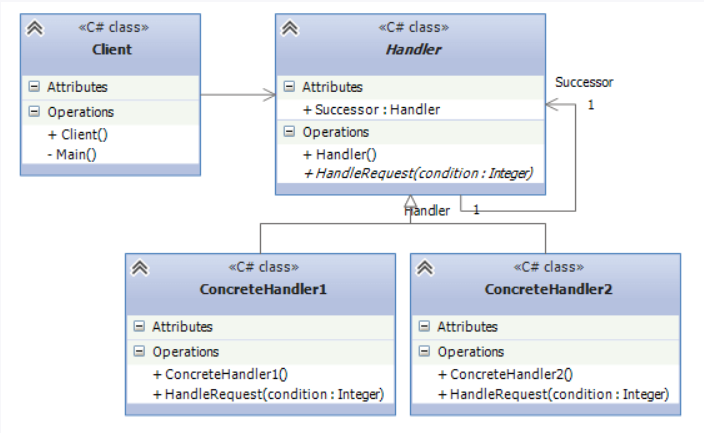
**Цепочка обязанностей**

Проблема: ряд действий нужно выполнять последовательно: обработка запроса

Все обработчики образуют цепочку, каждый обработчик имеет ссылку на следующий. Запрос перемещается по ней.

Есть 2 сценария:

* 1. Запрос идет по цепочке, если какой-то из обработчиков не может обработать запрос(=проверка неуспешна), то запрос не идет дальше
  2. Запрос идет по цепочке, пока не найдется обработчик, который сможет обработать его

 **Handler** - определяет интерфейс для обработки запроса. Также может определять ссылку на следующий обработчик запроса

**ConcreteHandler** - конкретные обработчики, . При невозможности обработки (или наоборот, при успехе) и наличия ссылки на следующий обработчик, передают запрос этому обработчику

**Client**: отправляет запрос объекту Handler

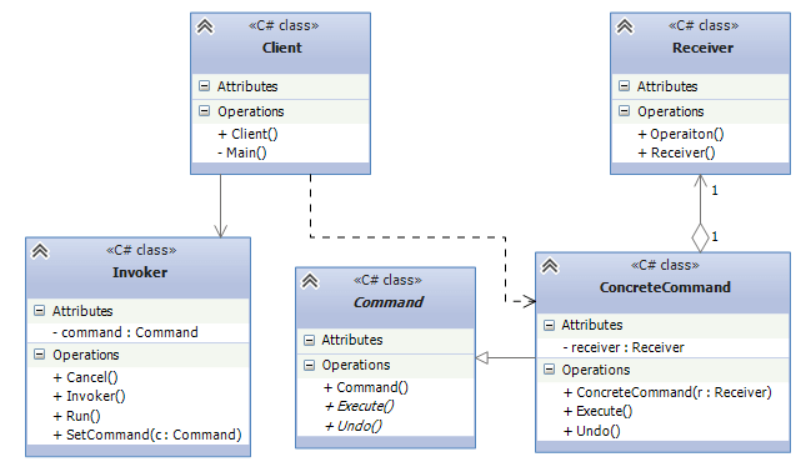
**Плюсы**:

* Ослабление связанности между объектами. Отправителю и получателю запроса ничего не известно друг о друге. Клиенту неизвестна цепочка объектов, какие именно объекты составляют ее, как запрос в ней передается.
* В цепочку с легкостью можно добавлять новые обработчики, которые реализуют общий интерфейс.

**Команда**

Проблема: вызов действия возможен из разных классов (герой движется вправо и на клавишу А и на стрелку), нужно передавать действие, которые могут вставать в очередь, и в ответ на действие может быть реакция

Паттерн предлагает инкапсулировать команду в отдельный класс



**Command**: интерфейс, представляющий команду. Обычно определяет метод Execute() для выполнения действия, а также нередко включает метод Undo(), реализация которого должна заключаться в отмене действия команды

**ConcreteCommand**: конкретная реализация команды, реализует метод Execute(), в котором вызывается определенный метод, определенный в классе Receiver

**Receiver**: получатель команды. Определяет действия, которые должны выполняться в результате запроса

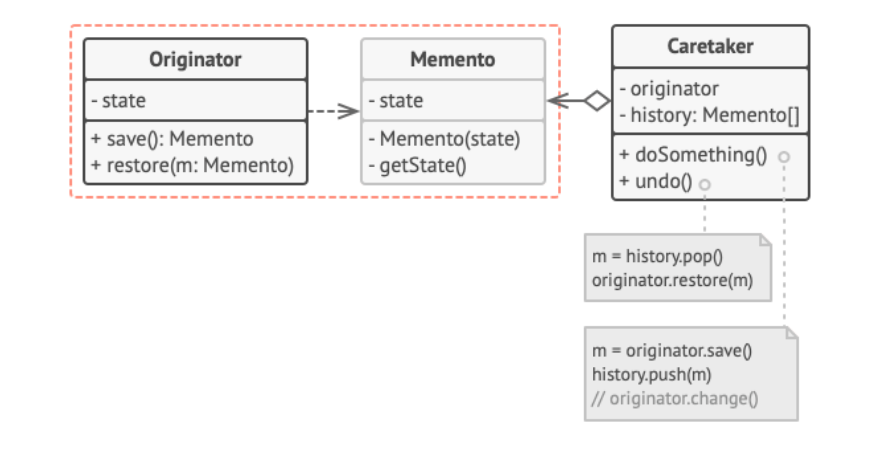
**Invoker**: инициатор команды - вызывает команду для выполнения определенного запроса

Client: клиент - создает команду и устанавливает ее получателя с помощью метода SetCommand()

**Снимок (Хранитель)**

Проблема: нужно сделать копию объекта (например нужна поддержка отменяемости действий, тогда сначала нужно сделать копию объекта, а уже потом применять изменения). Можно просто копировать его поля(при этом их придется делать публичными, что небезопастно), но если класс расширится, но в класс-копировальщик придется тоже дописывать код.

Все это нарушает инкапсуляцию, паттерн поручает создание копии объекта самому объекту (Originator)

**Originator -** создатель делает снимки своего состояния, может воспроизвести прошлое состояние, если в него подать снимок.

**Memento –** снимок, содержит состояние создателя, желательно иммутабельный. Это вложенный класс создателя, поэтому он имеет доступ к его полям.

**Caretaker –** опекун знает когда нужно делать снимок создателя, и когда его нужно восстановить. Может хранить историю состояний в виде стека снимков. Опекун не имеет доступа к полям создателя, он может манипулировать только снимками.

**Легковес**

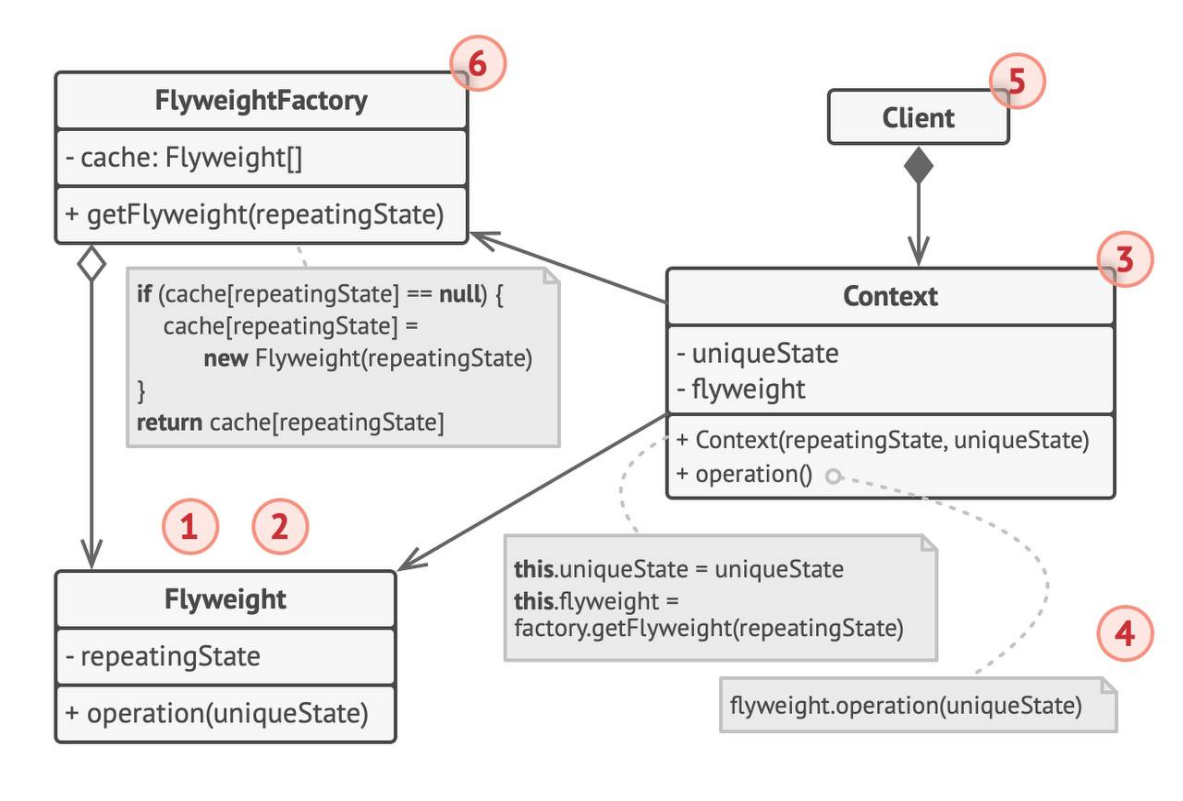
Проблема: используется большое количество однообразных объектов, которые используют много оперативы

Паттерн предлагает разделить информацию об объекте на изменяемую и неизменяемую часть (внешнее и внутреннее состояние).

**Изменяемая** часть (внешняя) – индивидуальные признаки, которые у каждого свои (координаты частицы в игре, паспортные данные сотрудника, положение символа на странице)

**Неизменяемая** часть (внутренняя) – признаки, которые объединяют многих (цвет частицы, компания сотрудника, код символа из таблицы кодировки)

При создании легковеса внешнее состояние выносится. В легковесе остается только внутреннее состояние. То есть в примере с символами легковес будет хранить код символа.

**Context** – объект, который мы поделили на внешнее (unique state) и внутреннее состояние (flyweight)

**Flyweight** – содержит внутреннюю часть, которая повторяется во множестве объектов. Operation – поведение объекта – можно оставить в легковесе и передавать данные контекста, можно наоборот.

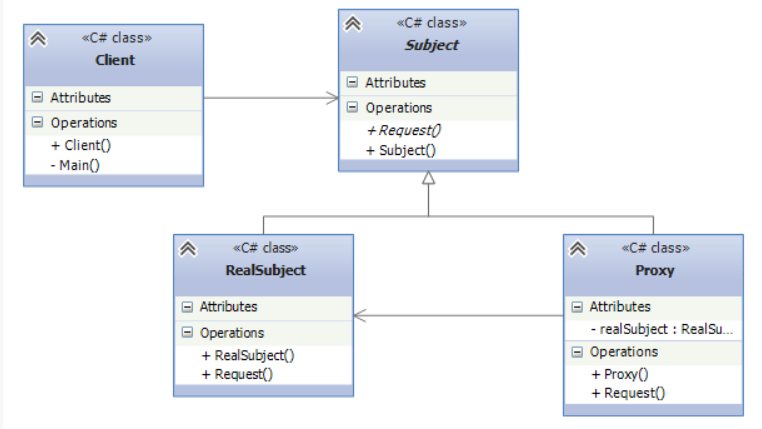
**FlyweightFactory –** фабрика управляет созданием и повторным использованием легковесов. Фабрика получает запрос на получение легковеса, создает новый, если его нет, и возвращает.   
**Client** – вычисляет и хранит контекст.

Плюсы: экономит оперативу

Минусы: процессорное время расходуется на поиск контекста, Усложнение кода программы

**Заместитель (прокси)**

Виды прокси:

* **Виртуальный** прокси (ленивая инициализация): идут запросы к тяжелому объекту, который грузит данные из фс или бд. Предлагается создать прокси, который будет принимать дополнительную информацию о тяжелом объекте. Только когда реально он понадобится, объект будет создан.
* **Защищающий** прокси (защита доступа): Прокси будет перенаправлять запросы к реальному объекту, только после проверки запроса на права доступа.
* **Удаленный** прокси (локальный запуск сервера): при взаимодействии по сети (распределенные системы) кодирование запросы и его аргументов для работы в другом адресном пространстве. Клиент работает с прокси, как будто тот находится в своем адресном пространстве.
* **Умная ссылка** (кэширование объектов): ведется подсчет ссылок на тяжелый объект, если все ссылки уже неактивны, то тяжелый объект можно свернуть. Можно отслеживать изменение клиентом в тяжелом объекте

Клиент обращается к прокси с запросом, если условия внутри прокси выполнены, то он перенаправляет запрос

**Многослойная архитектура**

1. **Какие слои присутствуют в классической 3х уровневой архитектуре. Дайте краткое описание и назначение каждого слоя**

**Presentation** layer

Взаимодействие с пользователем, получение от него данных (ввод), простая валидация, отрисовка интерфейса.

**Business logic** layer

Обработка пользовательского ввода, инфы полученной от presentation layer, алгоритмы, вычисления, взаимодействие с dal слоем, получение от него каких то данных, передача результата всех взаимодействий present layer

**Data access** layer

Хранит **модели**, которые описывают используемые сущности, также здесь размещаются **специфичные классы** для работы с разными технологиями доступа к данным, например, класс контекста данных Entity Framework. Здесь также хранятся **репозитории,** через которые уровень бизнес-логики взаимодействует с базой данных.

1. **Какие обязанности берет на себя слой DAL? С какими слоями и как он связан?**

Хранит модели, описывающие используемые сущности, также здесь размещаются специфичные классы для работы с разными технологиями доступа к данным, например, класс контекста данных Entity Framework. Здесь также хранятся репозитории, через которые уровень бизнес-логики взаимодействует с базой данных.

Для примера, DAL может возвращать объект с его атрибутами вместо строк из таблицы базы данных. Это позволяет создавать модули с более высоким уровнем абстракции.

Слой DAL **не зависит от других уровней**, может получать запросы из слоя **BL**, передавать свои запросы **базе данных** и получать от неё ответы

Носовицкий сказал что dal зависит от бизнес логики, а бизнес логика от него нет.

1. **Паттерны слоя DAL. Перечислить, дать краткое описание каждому, указать ключевые различия**

[Хороший сайт на русском](https://bool.dev/blog/detail/patterny-arkhitektury-istochnikov-dannykh-poeaa)

**Table data gateway**

Для каждой таблицы из бд создается свой класс, он работает со всеми строками таблицы. Прячет бд под crud интерфейс (create, read, update, delete).

Принимает параметры, а в базу данных посылает sql запрос. Сам класс состояния не имеет, просто работает почтальоном беду dal и bll

**Row data gateway**

класс, который является Gateway для одной строки в таблице базы данных. Один экземпляр работает со одной строкой в этой таблице. Создается класс с полями – названиями столбцов таблицы.

Можно сделать объект, и держать логику хранения данных внутри. Но в таком случае у нас наши сущности будут зависеть от базы данных, и их будет не так уж легко тестировать. Тесты будут требовать подключения к базе данных и от того станут медленными.

Для решения этой проблемы, нам нужна прослойка между сущностью и базой данных. Row Data Gateway. То есть шлюз просто хранит в себе строку, и бизнес логике не нужно лезть в бд напрямую.

**Active record**

То же самое что и Row Data, но Active Record **содержит код бизнес-логики.** Логика доступа данных в сущности. Нарушение SRP, но в маленьких проектах можно (и раз в книжке есть, то мы проходим). Пользоваться им не рекомендуется (хуйня)

**Data mapper**

Связывает две условно независимые сущности, перемещает между ними данных. Например, между базой данных и приложением.

Изменение domain model не требует изменения структуры БД и наоборот, что крайне важно при наличии сложных отображений, особенно при использовании уже существующих баз данных

Разница с gateway: mapper работает напрямую с доменной моделью(сущностью), а шлюз получит примитивы на вход (поля)

1. **Какие обязанности берет на себя слой BLL? С какими слоями и как он связан?**

Обрабатывает данные, полученные от presentation layer, алгоритмы, вычисления, обмен с бд, выдача этого всего обратно на уровень представления.

Взаимодействует с dal и presentation

1. **Паттерны слоя BLL. Перечислить, дать краткое описание каждому, указать ключевые различия**

**Transaction script**

Подходит для процедурного программирования. Одна процедура это а-ля одна транзакция.

Каждая процедура обрабатывает один запрос из present layer

Можно еще добавить ооп, а точнее паттерн команда( мне было лень разбираться, поэтому подробное описание просите у анжелики в конспектах)

Плюсы:

-просто

-сочетается с паттернами dal Row data gateway and Table data gateway

-четкие границы ответственности

Минусы:

-дублирование кода, если транзакции выполняют похожие функции

**Table module**

Сопоставление: класс – таблица данных. В отличии от domain model нет идентичности объектов, с которым работаем.

**Domain model**

Выделяются отдельные объекты, соотв. Предметной области

1. Презент слой пишем то же что в первом вопросе
2. Паттерн плагин не расписала, потому что нашла только какие-то абстракции