Паттерны

Что такое «шаблон проектирования»?

Шаблон (паттерн) проектирования (design pattern) — это проверенное и готовое к использованию решение. Это не класс и не библиотека, которую можно подключить к проекту, это нечто большее - он не зависит от языка программирования, не является законченным образцом, который может быть прямо преобразован в код и может быть реализован по-разному в разных языках программирования.

Плюсы использования шаблонов:

- снижение сложности разработки за счёт готовых абстракций для решения целого класса проблем.
- облегчение коммуникации между разработчиками, позволяя ссылаться на известные шаблоны.
- унификация деталей решений: модулей и элементов проекта.
- возможность отыскав удачное решение, пользоваться им снова и снова.
- помощь в выборе выбрать наиболее подходящего варианта проектирования.

Минусы:

- слепое следование некоторому выбранному шаблону может привести к усложнению программы.
- желание попробовать некоторый шаблон в деле без особых на то оснований.

Назовите основные характеристики шаблонов.

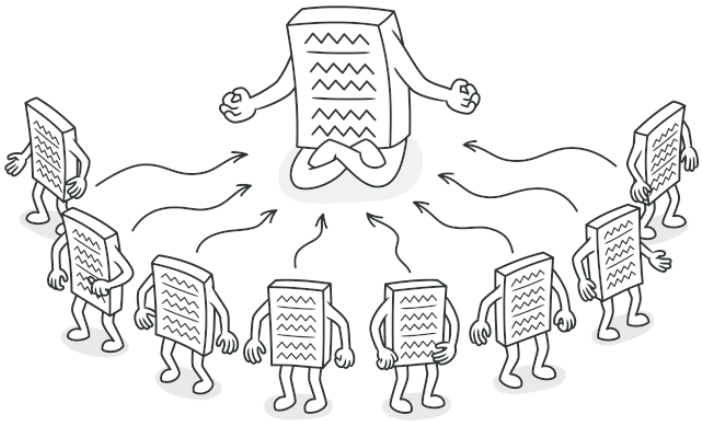
- **Имя** - все шаблоны имеют уникальное имя, служащее для их идентификации;
- **Назначение** назначение данного шаблона;
- **Задача** - задача, которую шаблон позволяет решить;
- **Способ решения** - способ, предлагаемый в шаблоне для решения задачи в том контексте, где этот шаблон был найден;
- **Участники** - сущности, принимающие участие в решении задачи;
- **Следствия** - последствия от использования шаблона как результат действий, выполняемых в шаблоне;
- **Реализация** - возможный вариант реализации шаблона.

Назовите три основные группы паттернов.

- **Порождающие** (Creational) **паттерны** беспокоятся о гибком создании объектов без внесения в программу лишних зависимостей.
- **Структурные** (Structural) **паттерны** показывают различные способы построения связей между объектами.
- **Поведенческие** (Behavioral) **паттерны** заботятся об эффективной коммуникации между объектами.

Расскажите про паттерн Одиночка (Singleton).

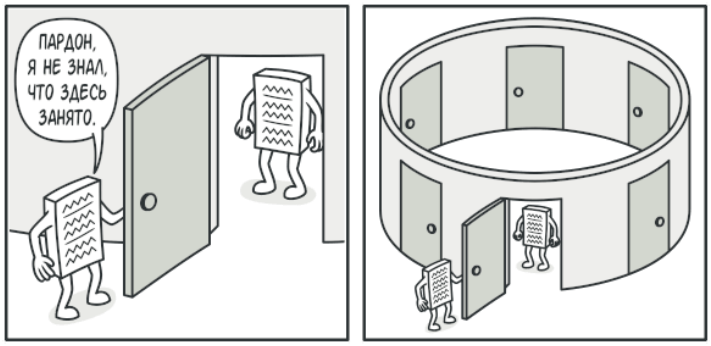
Одиночка — это порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.



Проблема

Одиночка решает сразу две проблемы, нарушая *принцип единственной ответственности* класса.

1. **Гарантирует наличие единственного экземпляра класса**. Чаще всего это полезно для доступа к какому-то общему ресурсу, например, базе данных.
   Представьте, что вы создали объект, а через некоторое время пробуете создать ещё один. В этом случае хотелось бы получить старый объект, вместо создания нового.
   Такое поведение невозможно реализовать с помощью обычного конструктора, так как конструктор класса **всегда** возвращает новый объект.



   Клиенты могут не подозревать, что работают с одним и тем же объектом.
2. **Предоставляет глобальную точку доступа**. Это не просто глобальная переменная, через которую можно достучаться к определённому объекту. Глобальные переменные не защищены от записи, поэтому любой код может подменять их значения без вашего ведома.
   Но есть и другой нюанс. Неплохо бы хранить в одном месте и код, который решает проблему №1, а также иметь к нему простой и доступный интерфейс.

Интересно, что в наше время паттерн стал настолько известен, что теперь люди называют «одиночками» даже те классы, которые решают лишь одну из проблем, перечисленных выше.

Решение

Все реализации одиночки сводятся к тому, чтобы скрыть конструктор по умолчанию и создать публичный статический метод, который и будет контролировать жизненный цикл объекта-одиночки.
Если у вас есть доступ к классу одиночки, значит, будет доступ и к этому статическому методу. Из какой точки кода вы бы его ни вызвали, он всегда будет отдавать один и тот же объект.

Расскажите про паттерн Строитель (Builder).

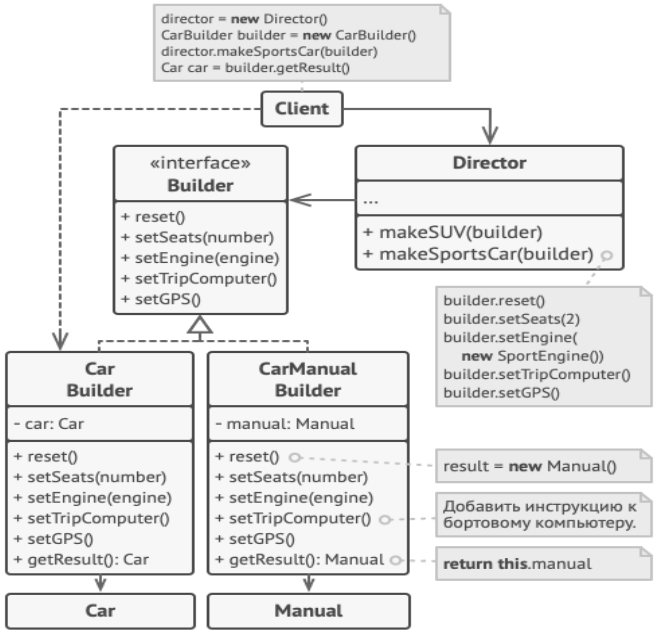
https://refactoring.guru/ru/design-patterns/builder

Создает сложные объекты пошагово. Можно использовать один и тот же код для разных представлений объекта.

Проблема: избавляет от сложных телескопических конструкторов (не все параметры нужны большую часть времени).

КАК: конструирование объекта выносится в другие классы-строители. Чтобы создать объект, нужно поочерёдно вызывать только нужные методы строителя. Можно создать несколько классов-строителей, выполняющих одни и те же шаги по-разному. Можно сделать класс-директор, который будет знать, как делать определенный тип объектов, а класс-строитель выполнять.

Позволяет создавать продукты пошагово.	Усложняет код программы из-за введения дополнительных классов.
Позволяет использовать один и тот же код для создания различных продуктов.	Клиент будет привязан к конкретным классам строителей, так как в интерфейсе директора может не быть метода получения результата.
Изолирует сложный код сборки продукта от его основной бизнес-логики.	



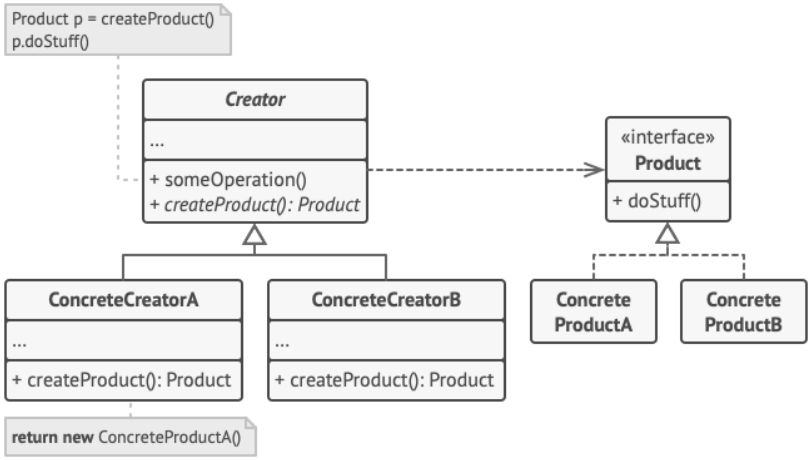
Расскажите про паттерн Фабричный метод (Factory Method).

https://refactoring.guru/ru/design-patterns/factory-method

Определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.

Проблема: Когда заранее неизвестны типы и зависимости объектов, с которыми должен работать ваш код.

КАК: Фабричный метод отделяет код производства продуктов от остального кода, который эти продукты использует. Благодаря этому, код производства можно расширять, не трогая основной. Так, чтобы добавить поддержку нового продукта, вам нужно создать новый подкласс и определить в нём фабричный метод, возвращая оттуда экземпляр нового продукта.



UI-фреймворк - User interface

Расскажите про паттерн Абстрактная фабрика (Abstract Factory).

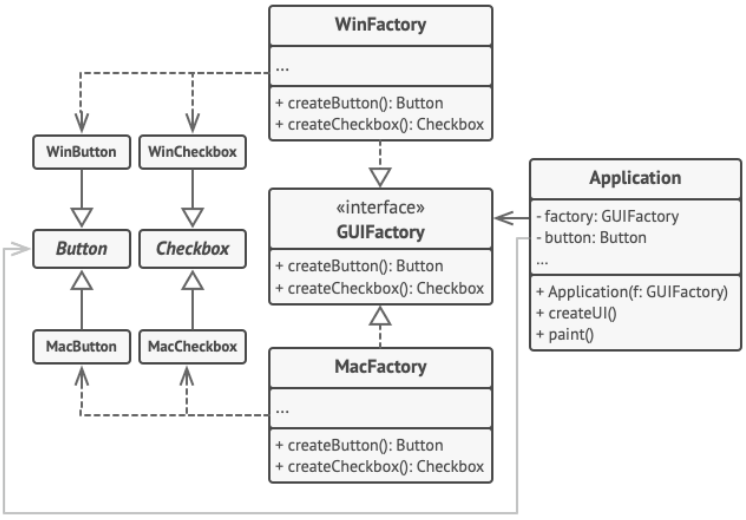
https://refactoring.guru/ru/design-patterns/abstract-factory

позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов

Применение: Когда бизнес-логика программы должна работать с разными видами связанных друг с другом продуктов, не завися от конкретных классов продуктов.

скрывает от клиентского кода подробности того, как и какие конкретно объекты будут созданы. Но при этом клиентский код может работать со всеми типами создаваемых продуктов, поскольку их общий интерфейс был заранее определён.

Когда в программе уже используется Фабричный метод, но очередные изменения предполагают введение новых типов продуктов - Если класс имеет слишком много фабричных методов, они способны затуманить его основную функцию. Поэтому имеет смысл вынести всю логику создания продуктов в отдельную иерархию классов, применив абстрактную фабрику.



Гарантирует сочетаемость создаваемых продуктов.	Усложняет код программы из-за введения множества дополнительных классов.
Избавляет клиентский код от привязки к конкретным классам продуктов.	Требует наличия всех типов продуктов в каждой вариации
Выделяет код производства продуктов в одно место, упрощая поддержку кода.	
Упрощает добавление новых продуктов в программу.	

Реализует принцип открытости/закрытости	

Расскажите про паттерн Прототип (Prototype).

https://refactoring.guru/ru/design-patterns/prototype

позволяет копировать объекты, не вдаваясь в подробности их реализации

Применимость: Когда ваш код не должен зависеть от классов копируемых объектов,

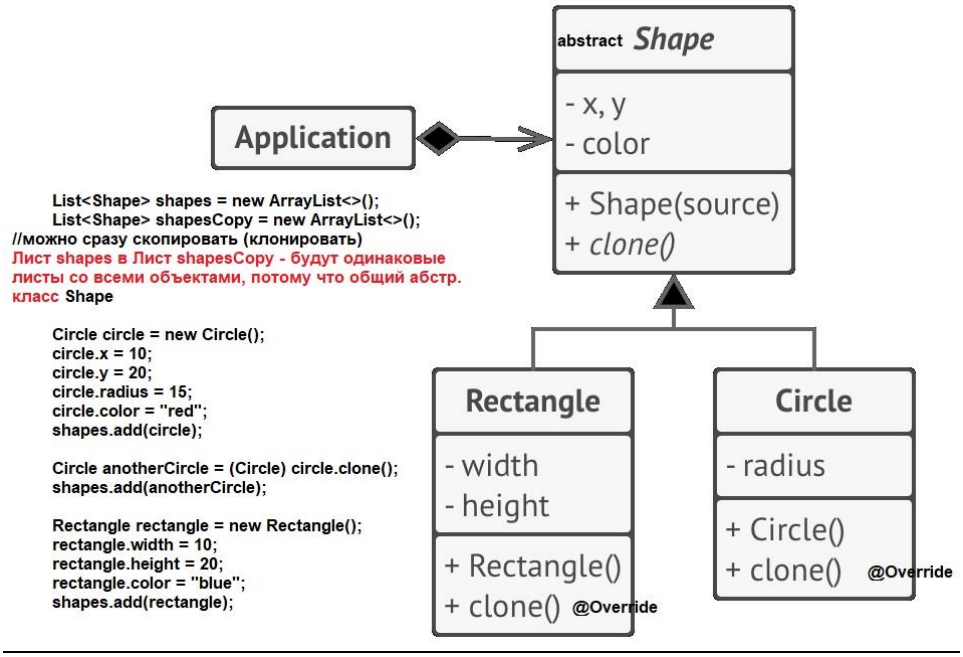
Паттерн прототип предоставляет клиенту общий интерфейс для работы со всеми прототипами. Клиенту не нужно зависеть от всех классов копируемых объектов, а только от интерфейса клонирования.

Когда вы имеете уйму подклассов, которые отличаются начальными значениями полей.

Паттерн прототип предлагает использовать набор прототипов, вместо создания подклассов для описания популярных конфигураций объектов.

Вместо порождения объектов из подклассов, вы будете копировать существующие объекты-прототипы, в которых уже настроено внутреннее состояние. Это позволит избежать взрывного роста количества классов в программе и уменьшить её сложность

Позволяет клонировать объекты, не привязываясь к их конкретным классам.	Сложно клонировать составные объекты, имеющие ссылки на другие объекты.
Меньше повторяющегося кода инициализации объектов.	
Ускоряет создание объектов.	
Альтернатива созданию подклассов для конструирования сложных объектов.	



Расскажите про паттерн Адаптер (Adapter). (структурный)

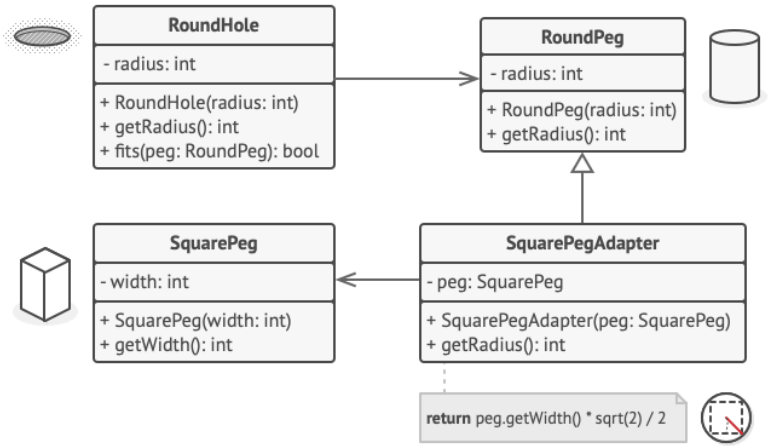
https://refactoring.guru/ru/design-patterns/adapter

Позволяет подружить несовместимые объекты.

Адаптер выступает прослойкой между двумя объектами, превращая вызовы одного в вызовы понятные другому.

1. Адаптер имеет интерфейс, который совместим с одним из объектов.
2. Поэтому этот объект может свободно вызывать методы адаптера.

3. Адаптер получает эти вызовы и перенаправляет их второму объекту, но уже в том формате и последовательности, которые понятны второму объекту.



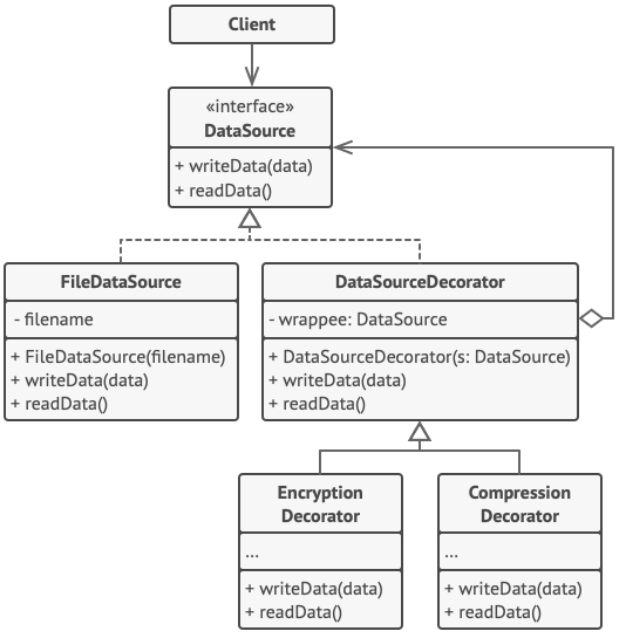
Когда: Когда вы хотите использовать сторонний класс, но его интерфейс не соответствует остальному коду приложения.
Когда вам нужно использовать несколько существующих подклассов, но в них не хватает какой-то общей функциональности, причём расширить суперкласс вы не можете

Отделяет и скрывает от клиента подробности преобразования различных интерфейсов.	Усложняет код программы из-за введения дополнительных классов.

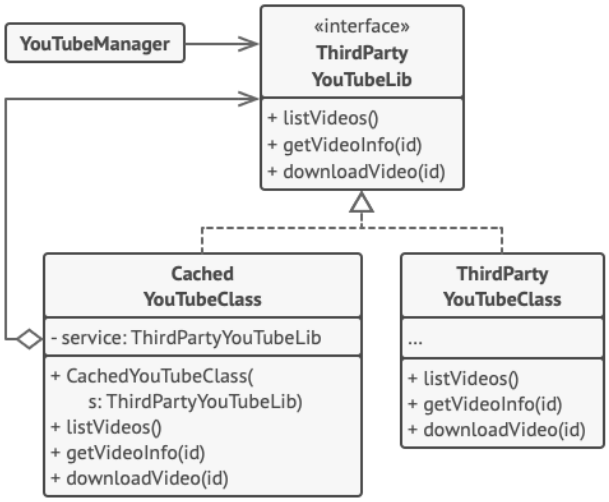
Расскажите про паттерн Декоратор (Decorator) он же ОБЁРТКА (структурный).

https://refactoring.guru/ru/design-patterns/decorator
позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».
Помещаете целевой объект в другой объект-обёртку, который запускает базовое поведение объекта, а затем добавляет к результату что-то своё. Оба объекта имеют общий интерфейс, поэтому для пользователя нет никакой разницы, с каким объектом работать — чистым или обёрнутым. Вы можете использовать несколько разных обёрток одновременно — объединённое поведение всех обёрток сразу.
Применимость:
-Когда вам нужно добавлять обязанности объектам на лету, незаметно для кода, который их использует. Объекты помещают в обёртки, имеющие дополнительные поведения.
-Когда нельзя расширить обязанности объекта с помощью наследования. ключевое слово final, которое может заблокировать наследование класса. Расширить такие классы можно только с помощью Декоратора.

Большая гибкость, чем у наследования.	Трудно конфигурировать многократно обёрнутые объекты.
Позволяет добавлять обязанности на лету.	Обилие крошечных классов
Можно добавлять несколько новых обязанностей сразу.	
Позволяет иметь несколько мелких объектов вместо одного объекта на все случаи жизни.	



Расскажите про паттерн Заместитель (Proxy). структурный
https://refactoring.guru/ru/design-patterns/proxy
Примеры:



- Ленивая инициализация (виртуальный прокси). Когда у вас есть тяжёлый объект, грузящий данные из файловой системы или базы данных. Вместо того, чтобы грузить данные сразу после старта программы, можно сэкономить ресурсы и создать объект тогда, когда он действительно понадобится.
- Локальный запуск сервиса (удалённый прокси). Когда настоящий сервисный объект находится на удалённом сервере. В этом случае заместитель транслирует запросы клиента в вызовы по сети в протоколе, понятном удалённому сервису.
- Логирование запросов (логирующий прокси). Когда требуется хранить историю обращений к сервисному объекту. Заместитель может сохранять историю обращения клиента к сервисному объекту.
- Кеширование объектов («умная» ссылка). Когда нужно кешировать результаты запросов клиентов и управлять их жизненным циклом. Заместитель может подсчитывать количество ссылок на сервисный объект, которые были отданы клиенту и остаются активными. Когда все ссылки освобождаются, можно будет освободить и сам сервисный объект (например, закрыть подключение к базе данных). Кроме того, Заместитель может отслеживать, не менял ли клиент

сервисный объект. Это позволит использовать объекты повторно и здóрово экономить ресурсы, особенно если речь идёт о больших прожорливых сервисах.

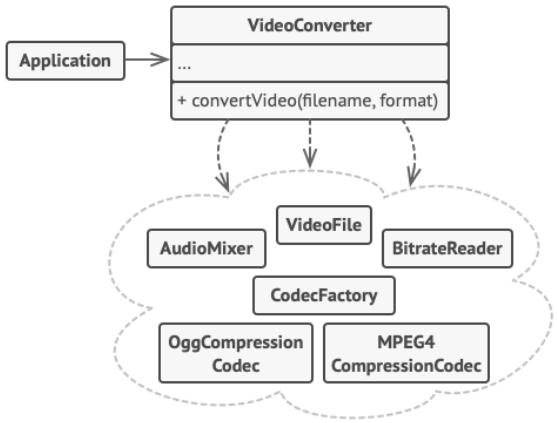
Позволяет контролировать сервисный объект незаметно для клиента.	Усложняет код программы из-за введения дополнительных классов.
Может работать, даже если сервисный объект ещё не создан.	Увеличивает время отклика от сервиса.
Может контролировать жизненный цикл служебного объекта.	

Паттерн Фасад структурный

https://refactoring.guru/ru/design-patterns/facade

предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку
- Когда вам нужно представить простой или урезанный интерфейс к сложной подсистеме.
- Когда вы хотите разложить подсистему на отдельные слои. Используйте фасады для определения точек входа на каждый уровень подсистемы.

Вместо непосредственной работы с дюжиной классов (сторонней библиотеки), фасад предоставляет коду приложения единственный метод для конвертации видео, который сам заботится о том, чтобы правильно сконфигурировать нужные объекты фреймворка и получить требуемый результат



Изолирует клиентов от компонентов сложной подсистемы.	Фасад рискует стать божественным объектом*, привязанным ко всем классам программы . * - антипаттерн объектно-ориентированного программирования, описывающий объект, который хранит в себе «слишком много» или делает «слишком много»

Расскажите про паттерн Итератор (Iterator) поведенческий.

https://refactoring.guru/ru/design-patterns/iterator

позволяет последовательно обходить сложную коллекцию, без раскрытия деталей её реализации. клиент может обходить разные коллекции одним и тем же способом, используя единый интерфейс итераторов.

Когда: у вас есть сложная структура данных, и вы хотите скрыть от клиента детали её реализации (из-за сложности или вопросов безопасности).

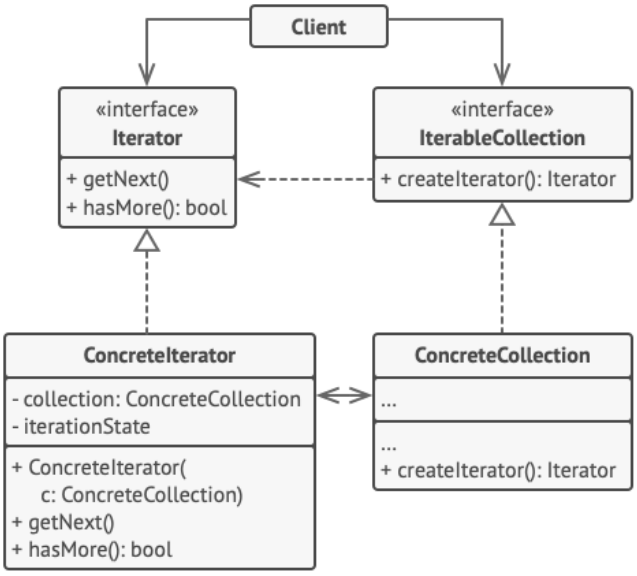
Итератор предоставляет клиенту всего несколько простых методов перебора элементов коллекции. Это не только упрощает доступ к коллекции, но и защищает её данные от неосторожных или злоумышленных действий.

Когда: вам нужно иметь несколько вариантов обхода одной и той же структуры данных.

Нетривиальные алгоритмы обхода структуры данных могут иметь довольно объёмный код. Этот код будет захламлять всё вокруг — будь то сам класс коллекции или часть бизнес-логики программы. Применив итератор, вы можете выделить код обхода структуры данных в собственный класс, упростив поддержку остального кода.

Когда: вам хочется иметь единый интерфейс обхода различных структур данных.

Итератор позволяет вынести реализации различных вариантов обхода в подклассы. Это позволит легко взаимозаменять объекты итераторов, в зависимости от того, с какой структурой данных приходится работать.



Упрощает классы хранения данных.	Не оправдан, если можно обойтись простым циклом.
Позволяет реализовать различные способы обхода структуры данных.	
Позволяет одновременно перемещаться по структуре данных в разные стороны.	

Расскажите про паттерн Шаблонный метод (Template Method) поведенческий.

https://refactoring.guru/ru/design-patterns/template-method

Определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Паттерн позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.

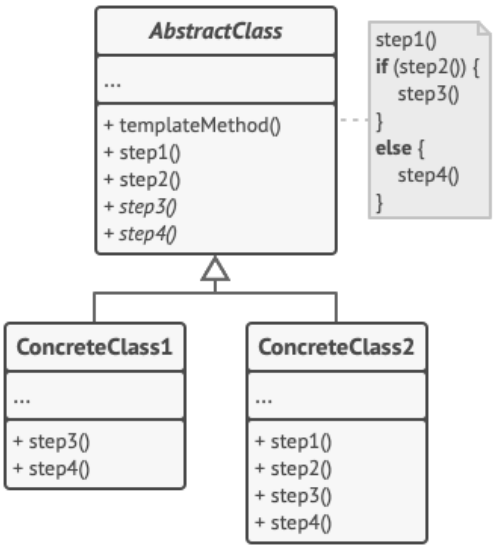
- Когда подклассы должны расширять базовый алгоритм, не меняя его структуры.

Шаблонный метод позволяет подклассам расширять определённые шаги алгоритма через наследование, не меняя при этом структуру алгоритмов, объявленную в базовом классе.

- Когда у вас есть несколько классов, делающих одно и то же с незначительными отличиями. Если вы редактируете один класс, то приходится вносить такие же правки и в остальные классы.

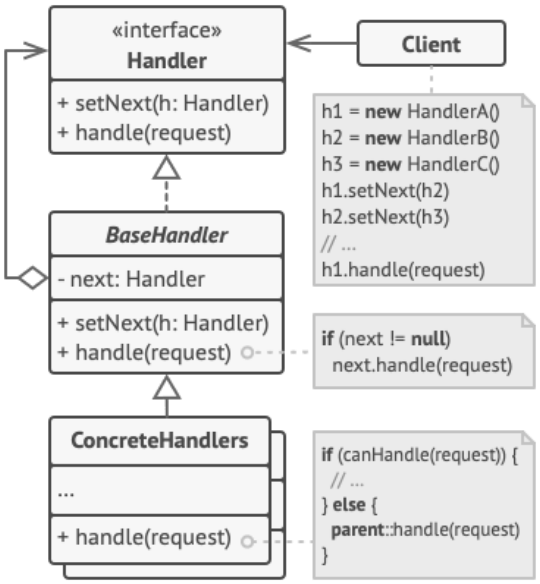
Паттерн шаблонный метод предлагает создать для похожих классов общий суперкласс и оформить в нём главный алгоритм в виде шагов. Отличающиеся шаги можно переопределить в подклассах.

Это позволит убрать дублирование кода в нескольких классах с похожим поведением, но отличающихся в деталях.



Облегчает повторное использование кода.	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

Расскажите про паттерн Цепочка обязанностей (Chain of Responsibility). поведенческий
https://refactoring.guru/ru/design-patterns/chain-of-responsibility
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
- Когда: программа должна обрабатывать разнообразные запросы несколькими способами, но заранее неизвестно, какие конкретно запросы будут приходить и какие обработчики для них понадобятся. С помощью Цепочки обязанностей вы можете связать потенциальных обработчиков в одну цепь и при получении запроса поочерёдно спрашивать каждого из них, не хочет ли он обработать запрос.
- Когда важно, чтобы обработчики выполнялись один за другим в строгом порядке.
- Когда набор объектов, способных обработать запрос, должен задаваться динамически.
В любой момент вы можете вмешаться в существующую цепочку и переназначить связи так, чтобы убрать или добавить новое звено.



Уменьшает зависимость между клиентом и обработчиками.	Запрос может остаться никем не обработанным.
Реализует принцип единственной обязанности.	
Реализует принцип открытости/закрытости.	

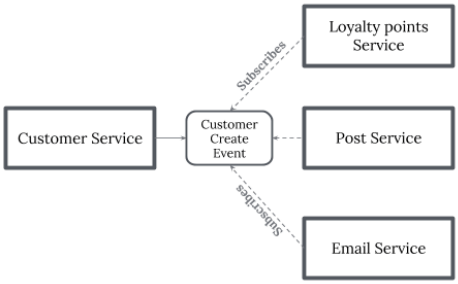
Паттерн Saga

https://bool.dev/blog/detail/saga-pattern-i-raspredelennye-tranzaktsii

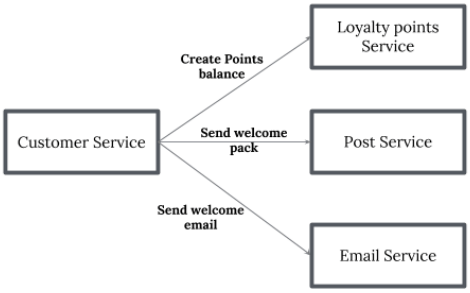
- это решение для реализации бизнес-транзакций, охватывающих несколько микросервисов. По сути, это последовательность транзакций. В SAGA каждая служба, выполняющая транзакцию, публикует событие. Последующий сервис, который выполняет следующую транзакцию в цепочках, будет запускаться по выходу предыдущей транзакции, а затем продолжится до последней транзакции в цепочках. В случае сбоя одной из транзакций в цепочках SAGA выполнит ряд резервных действий, чтобы отменить влияние всех предыдущих транзакций.

Когда каждый сервис имеет свою собственную базу данных и бизнес-транзакция охватывает несколько сервисов, как мы обеспечиваем согласованность данных между сервисами. Каждый запрос имеет компенсационный запрос, который выполняется при сбое запроса. Это может быть реализовано двумя способами:

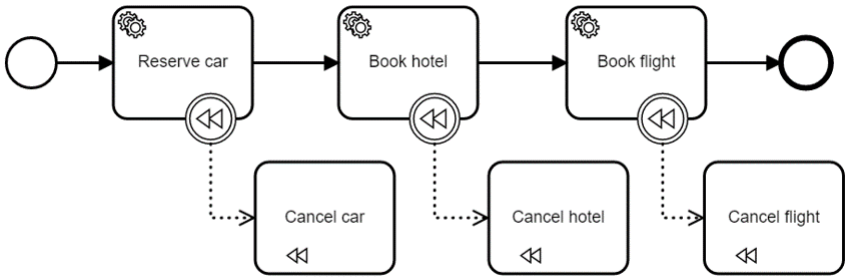
- Choreography (Хореография) — при отсутствии центра координации, каждый сервис создает и слушает события другого сервиса и решает, следует ли предпринять действие или нет. Хореография это способ указать, как две или более сторон; ни один из которых не имеет никакого контроля над процессами других сторон, или, возможно, какой-либо видимости этих процессов - не может координировать свои действия и процессы для обмена информацией и ценностями. Используйте хореографию, когда требуется координация между областями контроля / видимости. Вы можете думать о хореографии в простом сценарии как о сетевом протоколе. Это диктует приемлемые образцы запросов и ответов между сторонами.



- Orchestration (Оркестрация) — оркестратор (объект) берет на себя ответственность за принятие сагой решений и последовательность бизнес-логики. когда у вас есть контроль над всеми участниками процесса. когда все они находятся в одной области контроля, и вы можете контролировать поток действий. Это, конечно, чаще всего, когда вы указываете бизнес-процесс, который будет выполняться внутри одной организации, которую вы контролируете.



Sagas исходят из осознания того, что особо долгоживущие транзакции, а также далеко распределенные транзакции по границам местоположения и / или доверия не могут быть легко обработаны с использованием классической модели ACID с Two-Phase Commit принципом.



Какие паттерны используются в Spring Framework?

Singleton - Creating beans with default scope.
Factory - Bean Factory classes
Prototype - Bean scopes
Adapter - Spring Web and Spring MVC
Proxy - Spring Aspect Oriented Programming support
транзакции в спринге + прокси (на собесах)
Template Method - JdbcTemplate, HibernateTemplate etc
Front Controller - Spring MVC DispatcherServlet
Один контроллер обрабатывает все запросы к веб-сайту.

Паттерн Front Controller объединяет всю обработку запросов, пропуская запросы через единственный объект-обработчик. В Spring в качестве Front Controller выступает DispatcherServlet, все действия проходят через него. Как правило в приложении задаётся только один DispatcherServlet с маппингом “/”, который перехватывает все запросы. Это и есть реализация паттерна Front Controller

Data Access Object - Spring DAO support

DAO – предоставляет абстрактный интерфейс к любому типу хранилища данных. То есть клиент не работает с хранилищем данных напрямую. Скрывает детали реализации доступа к источникам данных, и это позволяет изменять источник данных без влияния на бизнес логику.

Dependency Injection and Aspect Oriented Programming

Внедрение зависимостей — это шаблон проектирования для реализации IoC. Соединение объектов с другими объектами или «внедрение» объектов в другие объекты выполняется контейнером IoC, а не самими объектами.

Какие паттерны используются в Hibernate?

https://bool.dev/blog/detail/domain-model-domain-logic-patterns-poeaa

Domain Model – объектная модель предметной области, включающая в себя как поведение, так и данные. Цель проектирования предметной области – определение бизнес-объектов, которые представляют реальные сущности предметной области.

Реализация модели предметной области означает пополнение приложения целым слоем объектов, описывающих различные стороны определенной области бизнеса. Одни объекты призваны имитировать элементы данных, которыми оперируют в этой области, а другие должны формализовать те или иные бизнес-правила. Функции тесно сочетаются с данными, которыми они манипулируют.

В модели предметной области смешиваются данные и функции, допускаются многозначные атрибуты, создаются сложные сети ассоциаций и используются связи наследования. В сфере корпоративных программных приложений можно выделить две разновидности моделей предметной области. "Простая" во многом походит на схему базы данных. Представление бизнес-логики и содержит, как правило, по одному объекту домена в расчете на каждую таблицу. "Сложная" модель может отличаться от структуры базы данных и содержать иерархии наследования, стратегии и иные типовые решения, а также сложные сети мелких взаимосвязанных объектов.

Data Mapper – слой мапперов (Mappers), который передает данные между объектами и базой данных, сохраняя их независимыми друг от друга и себя.

Proxy — применяется для ленивой загрузки.

Factory — используется в SessionFactory

Алгоритмы и структуры данных

Что такое Big O? Как происходит оценка асимптотической сложности алгоритмов?

«О» большое - математическое обозначение для сравнения асимптотического поведения (асимптотики) функций.

Под асимптотикой понимается характер изменения функции при стремлении ее аргумента к определённой точке.

Фраза «сложность алгоритма есть O(f(n)) означает, что с увеличением параметра n, характеризующего количество входной информации алгоритма, время работы алгоритма будет возрастать не быстрее, чем некоторая константа, умноженная на f(n).

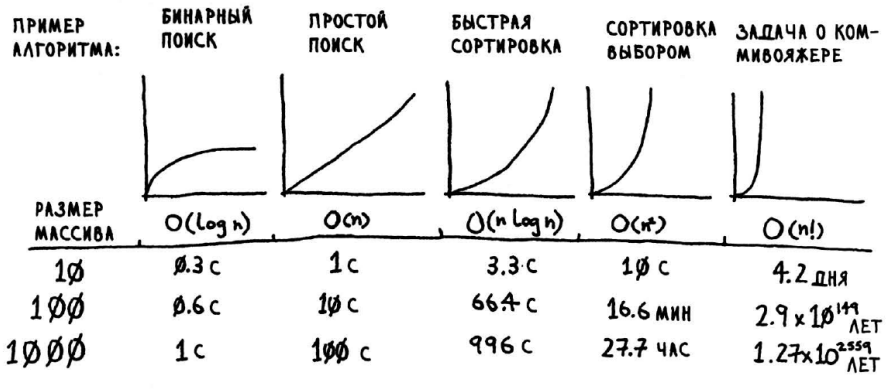
«О» большое описывает, насколько быстро работает алгоритм. Время выполнения «О» большое имеет вид O(n). Постойте, но где же секунды? А их здесь нет - «О» большое не сообщает скорость в секундах, а позволяет сравнить количество операций. Оно указывает, насколько быстро возрастает время выполнения алгоритма. Такая запись O(n) - сообщает количество операций, которые придется выполнить алгоритму.

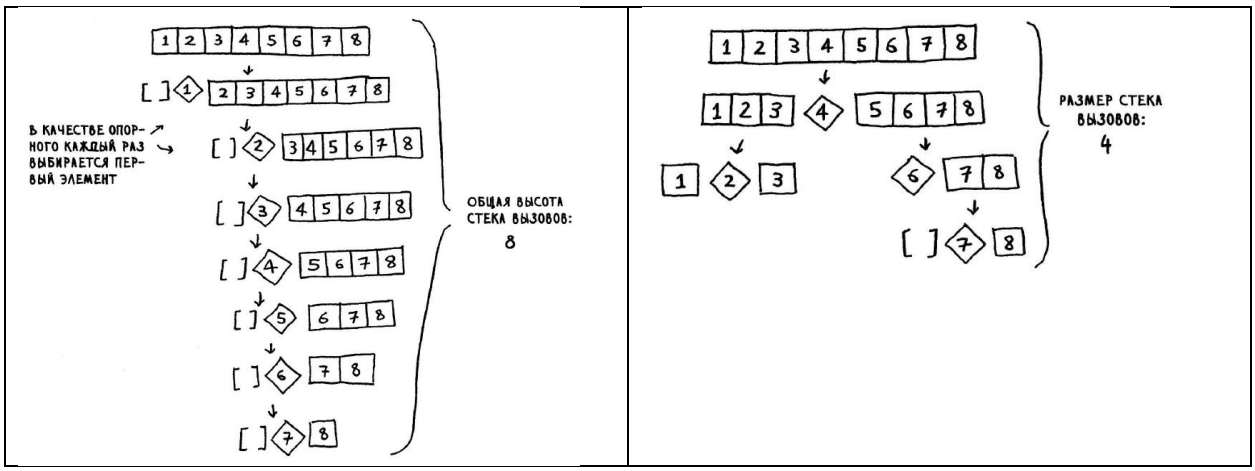
«О-большое» определяет время выполнения в худшем случае.

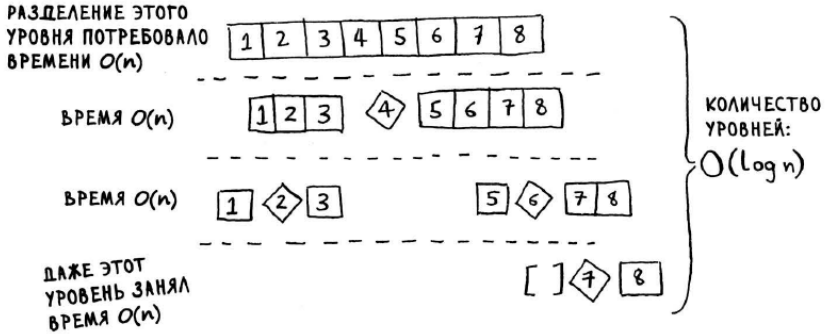
https://drive.google.com/open?id=1ySStPNuokeJ2PXM4nThuLNLG7SxHy3Oz

Ниже перечислены пять разновидностей «О-большого», которые будут встречаться вам особенно часто, в порядке убывания скорости выполнения:

- $O(\log n)$, или *логарифмическое время*. Пример: бинарный поиск.
- $O(n)$, или *линейное время*. Пример: простой поиск.
- $O(n * \log n)$. Пример: эффективные алгоритмы сортировки (быстрая сортировка — но об этом в главе 4).
- $O(n^2)$. Пример: медленные алгоритмы сортировки (сортировка выбором — см. главу 2).
- $O(n!)$. Пример: очень медленные алгоритмы (задача о коммивояжере — о ней будет рассказано в следующем разделе).
- Скорость алгоритмов измеряется не в секундах, а в темпе роста количества операций.
- По сути формула описывает, насколько быстро возрастает время выполнения алгоритма с увеличением размера входных данных.
- Время выполнения алгоритмов выражается как «О-большое».
- Время выполнения $O(\log n)$ быстрее $O(n)$, а с увеличением размера списка, в котором ищется значение, оно становится *намного* быстрее.







В этом примере существуют $O(\log n)$ (с технической точки зрения правильнее сказать «высота стека вызовов равна $O(\log n)$») уровней. А так как каждый уровень занимает время $O(n)$, то весь алгоритм займет время $O(n) * O(\log n) = O(n \log n)$. Это сценарий лучшего случая.

В худшем случае существуют $O(n)$ уровней, поэтому алгоритм займет время $O(n) * O(n) = O(n^2)$.

А теперь сюрприз: лучший случай также является средним. Если вы всегда будете выбирать опорным элементом случайный элемент в массиве, быстрая сортировка в среднем завершится за время $O(n \log n)$. Это один из самых быстрых существующих алгоритмов сортировки, который заодно является хорошим примером стратегии «разделяй и властвуй».

	Хеш-таблицы (средний случай)	Хеш-таблицы (худший случай)	Массивы	Связанные списки
Поиск	O(1)	O(n)	O(1)	O(n)
Вставка	O(1)	O(n)	O(n)	O(1)
Удаление	O(1)	O(n)	O(n)	O(1)

Что такое рекурсия? Сравните преимущества и недостатки итеративных и рекурсивных алгоритмов. С примерами.

Рекурсия - способ отображения какого-либо процесса внутри самого этого процесса, то есть ситуация, когда процесс является частью самого себя.
Для того чтобы понять рекурсию, надо сначала понять рекурсию.

Рекурсия состоит из базового случая и шага рекурсии. Базовый случай представляет собой самую простую задачу, которая решается за одну итерацию, например, if(n == 0) return 1.

В базовом случае обязательно присутствует условие выхода из рекурсии;
Смысл рекурсии в движении от исходной задачи к базовому случаю, пошагово уменьшая размер исходной задачи на каждом шаге рекурсии.

После того, как будет найден базовый случай, срабатывает условие выхода из рекурсии, и стек рекурсивных вызовов разворачивается в обратном порядке, пересчитывая результат исходной задачи, который основан на результате, найденном в базовом случае. Так работает рекурсивное вычисление факториала.

Пример на Java:

```
int factorial(int n) {
if(n == 0) return 1; // в этой строчке базовый случай с условием выхода
else return n * factorial(n-1)
// в этой строчке шаг рекурсии или рекурсивный вызов, который  ищет
базовый случай, пошагово уменьшая размер исходной задачи
}
```

Пример на C из википедии:

```
int factorial (int n) {
  return (n==0) ? 1 : n * factorial(n-1); //такой же if-else как в примере
выше, только в виде тернарного оператора;
}
```

Рекурсия имеет линейную сложность O(n);

Что такое жадные алгоритмы? Приведите пример.

Жадный алгоритм (greedy algorithm) - это алгоритм, который на каждом шагу делает локально наилучший выбор в надежде, что итоговое решение будет оптимальным.

К примеру, алгоритм Дейкстры (без отрицательных ребер) нахождения кратчайшего пути в графе вполне себе жадный, потому что мы на каждом шагу ищем вершину с наименьшим весом, в которой мы еще не бывали, после чего обновляем значения других вершин. При этом можно доказать, что кратчайшие пути, найденные в вершинах, являются оптимальными.

К слову, алгоритм Флойда, который тоже ищет кратчайшие пути в графе (правда, между всеми вершинами), не является примером жадного алгоритма.

Флойд демонстрирует другой метод — метод динамического программирования. Есть область применимости жадных алгоритмов.

Общих рецептов тут нет, но есть довольно мощный инструмент, с помощью которого в большинстве случаев можно определить, даст ли жадина оптимальное решение. Этот инструмент - матроид.

Матроид — это пара (X, I), где X — конечное множество, называемое носителем матроида, а I — некоторое множество подмножеств X, называемое семейством независимых множеств. При этом должны выполняться следующие условия: - Множество I непусто. Даже если исходное множество X было пусто — X = Ø, то I будет состоять из одного элемента — множества, содержащего пустое. I = {{Ø}} - Любое подмножество любого элемента множества I также будет элементом этого множества. - Если множества A и B принадлежат множеству I, а также известно, что размер A меньше B, то существует какой-нибудь элемент x из B, не принадлежащий A, такое что объединение x и A будет принадлежать множеству I. Это свойство является не совсем тривиальным, но чаще всего наиважшейшим из всех остальных.

Расскажите про пузырьковую сортировку.

Сортировка пузырьком / Bubble sort O(n^2) (^ - коньюкция (знак степени)).
Будем идти по массиву слева направо. Если текущий элемент больше следующего, меняем их местами. Делаем так, пока массив не будет отсортирован. Заметим, что после первой итерации самый большой элемент будет находиться в конце массива, на правильном месте. После двух итераций на правильном месте будут стоять два наибольших элемента, и так далее. Очевидно, не более чем после n итераций массив будет отсортирован. Таким образом, асимптотика в худшем и среднем случае – O(n^2), в лучшем случае – O(n).

```
public class BubbleSort {
 public static void main(String[] args) {
     int[] testData = {10, 4, 43, 5, 4, 67, 12, 0, 99, 19};
     System.out.println(Arrays.toString(bubbleSort(testData)));
 }

 public static int[] bubbleSort(int[] array) {
     boolean sorted = false;
```

```
            int temp;
            while(!sorted) {
                sorted = true;
                for (int i = 0; i < array.length - 1; i++) {
                    if (array[i] > array[i+1]) {
                        temp = array[i];
                        array[i] = array[i+1];
                        array[i+1] = temp;
                        sorted = false;
                    }
                }
            }
            return array;
        }
    }
```

https://proglib.io/p/java-sorting-algorithms

Расскажите про быструю сортировку.

Быстрая сортировка / Quicksort O(n^2)

Выберем некоторый опорный элемент. После этого перекинем все элементы, меньшие его, налево, а большие – направо. Рекурсивно вызовемся от каждой из частей. В итоге получим отсортированный массив, так как каждый элемент меньше опорного стоял раньше каждого большего опорного. Асимптотика: O(n*logn) в среднем и лучшем случае, O(n^2). Наихудшая оценка достигается при неудачном выборе опорного элемента.

```
    public class QuickSort {
        public static void main(String[] args) {
            int[] testData = {8, 0, -3, 5, 6, 9, 8, -4, 2, -99, 43};

            int low = 0;
            int high = testData.length - 1;

            quickSort(testData, low, high);
            System.out.println(Arrays.toString(testData));
        }

        public static void quickSort(int[] array, int low, int high) {
            if (array.length == 0)
                return;//завершить выполнение, если длина массива равна 0

            if (low >= high)
                return;//завершить выполнение если уже нечего делить

            //выбрать опорный элемент
            int middle = low + (high - low) / 2;
            int opora = array[middle];

            //разделить на подмассивы, который больше и меньше опорного
    элемента
            int i = low, j = high;
            while (i <= j) {
                while (array[i] < opora) {
                    i++;
                }

                while (array[j] > opora) {
                    j--;
                }

                if (i <= j) {//меняем местами
                    int temp = array[i];
                    array[i] = array[j];
                    array[j] = temp;
```

```
                i++;
                j--;
            }
        }

        //вызов рекурсии для сортировки левой и правой части
        if (low < j)
            quickSort(array, low, j);

        if (high > i)
            quickSort(array, i, high);
    }
}
```

https://otus.ru/nest/post/788/

Расскажите про сортировку слиянием.

Сортировка слиянием / Merge sort O(n*logn)

Сортировка, основанная на парадигме «разделяй и властвуй». Разделим массив пополам, рекурсивно отсортируем части, после чего выполним процедуру слияния: поддерживаем два указателя, один на текущий элемент первой части, второй – на текущий элемент второй части. Из этих двух элементов выбираем минимальный, вставляем в ответ и сдвигаем указатель, соответствующий минимуму. Слияние работает за O(n), уровней всего logn, поэтому асимптотика O(n*logn). Эффективно заранее создать временный массив и передать его в качестве аргумента функции. Эта сортировка рекурсивна, как и быстрая, а потому возможен переход на квадратичную при небольшом числе элементов.

```
public class MergeSort {
    public static void main(String[] args) {
        int[] array1 = {8, 0, -3, 5, 6, 9, 8, -4, 2, -99, 43};
        int[] result = mergesort(array1);
        System.out.println(Arrays.toString(result));
    }
    public static int[] mergesort(int[] array1) {
        int[] buffer1 = Arrays.copyOf(array1, array1.length);
        int[] buffer2 = new int[array1.length];
        int[] result = mergesortInner(buffer1, buffer2, 0, array1.length);
        return result;
    }

    /**
     * @param buffer1 Массив для сортировки.
     * @param buffer2 Буфер. Размер должен быть равен размеру buffer1.
     * @param startIndex Начальный индекс в buffer1 для сортировки.
     * @param endIndex Конечный индекс в buffer1 для сортировки.
     * @return
     */
    public static int[] mergesortInner(int[] buffer1, int[] buffer2,
                                       int startIndex, int endIndex) {
        if (startIndex >= endIndex - 1) {
            return buffer1;
        }

        // уже отсортирован.
        int middle = startIndex + (endIndex - startIndex) / 2;
        int[] sorted1 = mergesortInner(buffer1, buffer2, startIndex,
middle);
        int[] sorted2 = mergesortInner(buffer1, buffer2, middle,
endIndex);

        // Слияние
        int index1 = startIndex;
        int index2 = middle;
        int destIndex = startIndex;
        int[] result = sorted1 == buffer1 ? buffer2 : buffer1;
```

```
            while (index1 < middle && index2 < endIndex) {
                result[destIndex++] = sorted1[index1] < sorted2[index2]
                        ? sorted1[index1++] : sorted2[index2++];
            }
            while (index1 < middle) {
                result[destIndex++] = sorted1[index1++];
            }
            while (index2 < endIndex) {
                result[destIndex++] = sorted2[index2++];
            }
            return result;
        }
    }
```

https://urvanov.ru/2017/08/19/%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC-%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B8-%D1%81%D0%BB%D0%B8%D1%8F%D0%BD%D0%B8%D0%B5%D0%BC-%D0%BD%D0%B0-java/

Расскажите про бинарное дерево.

Бинарное дерево - иерархическая структура данных, в которой каждый узел имеет не более двух потомков (детей). Как правило, первый называется родительским узлом, а дети называются левым и правым наследниками. Каждый узел в дереве задаёт поддерево, корнем которого он является. Оба поддерева — левое и правое — являются двоичными деревьями. У всех узлов левого поддерева произвольного узла X значения ключей данных меньше, нежели значение ключа данных самого узла X. У всех узлов правого поддерева произвольного узла X значения ключей данных больше либо равны, нежели значение ключа данных самого узла X.
https://www.youtube.com/watch?v=C9FK1pHLnhI&list=PLQOaTSbfxUtAlipl4136nwb4ISyFk8oI4

Расскажите про красно-черное дерево.

Красно-черное дерево - один из видов самобалансирующихся двоичных деревьев поиска, гарантирующих логарифмический рост высоты дерева от числа узлов и позволяющее быстро выполнять основные операции дерева поиска: добавление, удаление и поиск узла.

Сбалансированность достигается за счёт введения дополнительного атрибута узла дерева — «цвета».

Этот атрибут может принимать одно из двух возможных значений — «чёрный» или «красный».

1. Узел может быть либо красным, либо черным и имеет двух потомков.
2. Корень — как правило чёрный. Это правило слабо влияет на работоспособность модели, так как цвет корня всегда можно изменить с красного на чёрный.
3. Все листья — черные и не содержат данных.
4. Оба потомка каждого красного узла — черные.
5. Любой простой путь от узла-предка до листового узла-потомка содержит одинаковое число черных узлов.

Благодаря этим ограничениям, путь от корня до самого дальнего листа не более чем вдвое длиннее, чем до самого ближнего и дерево примерно сбалансировано.

Операции вставки, удаления и поиска требуют в худшем случае времени, пропорционального длине дерева, что позволяет красно-черным деревьям быть более эффективными в худшем случае, чем обычные двоичные деревья поиска.
https://www.youtube.com/watch?v=-gKLTH0KTF4
https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

Расскажите про линейный и бинарный поиск.

Линейный поиск - O(n);
Бинарный поиск - O(log(n)); сложность ниже, но массив должен быть отсортирован
Поиск в глубину и в ширину в деревьях;

Расскажите про очередь и стек.

Queue - это очередь, которая обычно (но необязательно) строится по принципу FIFO (First-In-First-Out) - соответственно извлечение элемента осуществляется с начала очереди, вставка элемента - в конец очереди.

Хотя этот принцип нарушает, к примеру PriorityQueue, использующая «natural ordering» или переданный Comparator при вставке нового элемента.

Deque (Double Ended Queue) расширяет Queue и согласно документации это линейная коллекция, поддерживающая вставку/извлечение элементов с обоих концов. Помимо этого реализации интерфейса Deque могут строится по принципу FIFO, либо LIFO.

Реализации и Deque, и Queue обычно не переопределяют методы equals() и hashCode(), вместо этого используются унаследованные методы класса Object, основанные на сравнении ссылок.

Heap (куча) используется Java Runtime для выделения памяти под объекты и классы. Создание нового объекта также происходит в куче. Это же является областью работы сборщика мусора. Любой объект, созданный в куче, имеет глобальный доступ и на него могут ссылаться из любой части приложения.

Stack (стек) это область хранения данных также находящееся в общей оперативной памяти (RAM). Всякий раз, когда вызывается метод, в памяти стека создается новый блок, который содержит примитивы и ссылки на другие объекты в методе. Как только метод заканчивает работу, блок также перестает использоваться, тем самым предоставляя доступ для следующего метода. Размер стековой памяти намного меньше объема памяти в куче. Стек в Java работает по схеме LIFO (Последний зашел - Первый вышел)

Различия между Heap и Stack памятью:

- Куча используется всеми частями приложения в то время как стек используется только одним потоком исполнения программы.
- Всякий раз, когда создается объект, он всегда хранится в куче, а в памяти стека содержится лишь ссылка на него.
- Память стека содержит только локальные переменные примитивных типов и ссылки на объекты в куче.
- Объекты в куче доступны с любой точке программы, в то время как стековая память не может быть доступна для других потоков.
- Стековая память существует лишь какое-то время работы программы, а память в куче живет с самого начала до конца работы программы.
- Если память стека полностью занята, то Java Runtime бросает исключение java.lang.StackOverflowError.
- Если заполнена память кучи, то бросается исключение java.lang.OutOfMemoryError: Java Heap Space.
- Размер памяти стека намного меньше памяти в куче.
- Из-за простоты распределения памяти, стековая память работает намного быстрее кучи.

Сравните сложность вставки, удаления, поиска и доступа по индексу в ArrayList и LinkedList.

LinkedList в подавляющем большинстве случаев проигрывает ArrayList, но в оставшемся меньшинстве он вне конкуренции.

Когда использовать LinkedList:

1. Необходимо много данных добавлять в начало списка
2. Удалять с начала (index = 0) списка, т.е. элементы, которые были добавлены первыми.
3. .set в конце списка

Когда использовать ArrayList:

1. .get
2. .set (начало и середина)
3. .add
4. .remove (кроме начала списка)

Сравнение сложности:

	Индекс	Поиск	Вставка	Удаление
ArrayList	O(1)	O(n)	O(n)	O(n)
LinkedList	O(n)	O(n)	O(1)	O(1)

https://habr.com/ru/post/188010/
https://habr.com/ru/post/233797/