**Паттерны** – это повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого частного контекста. Существует 22 классических паттерна проектирования. Они делятся на три группы:

* **Порождающие** применяются для гибкого создания объектов без внесения в код лишних зависимостей.
* **Структурные** показывают различные способы построения связей между объектами.
* **Поведенческие** описывают эффективную коммуникацию между объектами.

Если алгоритм – это четкая последовательность действий для получения конкретного результата, то паттерн – это общее высокоуровневое описание задачи (концепция).

**Паттерн *Singleton* (одиночка)** – гарантирует создание только единственного экземпляра класса. Подходит для конфигурации.

**Жадная одиночка** – захватывает ресурсы сразу при старте:

*public class Configuration {*

*private String prop1;*

*private final static Configuration instance = new Configuration();*

*private Configuration() {*

*// read properties from file }*

*public static Configuration getInstance() {*

*return instance; }*

**Ленивая (*Lazy*) одиночка** – захватывает ресурсы при первом обращении (небезопасно в многопоточном приложении):

*public class Configuration {*

*private String prop1;*

*private final static Configuration instance = null;*

*private Configuration() {*

*// read properties from file }*

*public static Configuration getInstance() {*

*if (instance == null) { instance = new Configuration(); }*

*return instance; }*

**Ленивая (*Lazy*) одиночка через *Lombok*** – захватывает ресурсы при первом обращении (безопасно в многопоточном приложении):

*public class Configuration {*

*private String prop1;*

*@Getter(lazy = true)*

*private final static Configuration instance = new Configuration();*

*private Configuration() {*

*// read properties from file }*

**Паттерн *Builder*** (**строитель**) – порождающий паттерн проектирования, который применяется для удобного построения сложного объекта. Его применение имеет следующие преимущества:

* Можно создавать объекты поэтапно.
* Можно использовать один и тот же код для создания различных объектов.
* Код сборки изолируется от основной логики программы.

Можно сгенерировать конструкторы *Builder* через *idea* и вручную дописывать валидацию, либо использовать аннотации *Lombok*.

Аннотация ***@Builder*** на классе позволяет использовать одноименный паттерн.

Аннотация ***@Builder.Default*** на поле переменной позволяет присвоить значение по умолчанию.

Внешне использование паттерна *Builder* похоже на построение стрима.

*Car firstCar = Car.builder() // вызов класса-строителя*

*.color("white") // установка цвета*

*.power(200) // установка мощности*

*.height(1560) // установка высоты*

*.length(4280) // установка длины*

*.weight(1450) // установка ширины*

*.build(); // создание класса Car*

Использование на классе аннотации с флагом ***@Builder(toBuilder = true)*** позволяет использовать на его неизменяемом (***immuttable***) экземпляре метод toBuilder(), который вернет билдер с уже проинициализированными полями на основе исходного объекта. После этого в них можно поменять значения и сконструировать объект.

*final Point point1 = Point.builder().x(0).y(0).build(); // неизменяемый экземпляр*

*final Point point2 = point1.toBuilder().y(3).build(); // метод toBuilder()*

**Паттерн *Factory*** (**фабрика**) – выдает те или иные реализации объектов на основании условий.

*interface Human {*

*void scream(); }*

*class Suzi implements Human {*

*@Override*

*public void scream() { System.out.println(“Suzi scream”); } }*

*class French implements Human {*

*@Override*

*public void scream() { System.out.println(“French scream”); } }*

*interface HumanFactory {*

*Human getHuman(String requirements); }*

*class FrenchHumanFactory implements HumanFactory {*

*public Human getHuman(String requirements) {*

*if (requirements.contains(“french”)) return new French();*

*if (requirements.contains(“girl”)) return new Suzi();*

*return null; } }*

**Паттерн *Chain of Responsibility* (цепочка обязанностей)** – это поведенческий паттерн проектирования, который позволяет передавать запросы последовательно по цепочке обработчиков. Каждый обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи.

Сначала создадим базовый класс или интерфейс, цепочки:

*public abstract class Searcher {*

*protected Searcher next;*

*public Searcher setNext(Searcher next) {*

*this.next = next;*

*return next; }*

*public List<Booking> findAll(Long userId, State state, Pageable pageable,*

*LocalDateTime dateTime, BookingRepository bookingRepository) {*

*if (shouldSearch(state)) {*

*return findBooking(userId, state, pageable, dateTime, bookingRepository);*

*} else if (next != null) {*

*return next.findAll(userId, state, pageable, dateTime, bookingRepository);*

*} else {*

*throw new BookingException("State not found."); } }*

*public abstract Boolean shouldSearch(State state);*

*public abstract List<Booking> findBooking(Long userId, State state, Pageable pageable,*

*LocalDateTime dateTime, BookingRepository bookingRepository); }*

Потом создадим элементы цепочки:

*public class SearcherByBookerIdAndStateAll extends Searcher {*

*@Override*

*public Boolean shouldSearch(State state) {*

*return state.equals(State.ALL); }*

*@Override*

*public List<Booking> findAll(Long userId, State state, Pageable pageable,*

*LocalDateTime dateTime, BookingRepository bookingRepository) {*

*return bookingRepository.findByBookerIdOrderByStartDesc(userId, pageable).toList(); } }*

Собираем элементы цепочки в единую последовательность вызовов:

*public class ChainSearcherByBooker {*

*Searcher searcherAll = new SearcherByBookerIdAndStateAll();*

*Searcher searcherCurrent = searcherAll.setNext(new SearcherByBookerIdAndStateCurrent());*

*Searcher searcherPast = searcherCurrent.setNext(new SearcherByBookerIdAndStatePast());*

*Searcher searcherFuture = searcherPast.setNext(new SearcherByBookerIdAndStateFuture());*

*Searcher searcherWaiting = searcherFuture.setNext(new SearcherByBookerIdAndStateWaiting());*

*Searcher searcherRejected = searcherWaiting.setNext(new SearcherByBookerIdAndStateRejected());*

*public List<Booking> search(Long userId, State state, Pageable pageable,*

*LocalDateTime dateTime, BookingRepository bookingRepository) {*

*return searcherAll.findAll(userId, state, pageable, dateTime, bookingRepository); } }*

**Паттерн *Strategy* (стратегия)** – это поведенческий паттерн проектирования, который выносит набор алгоритмов в собственные классы и делает их взаимозаменяемыми.

Сначала создадим интерфейс:

*public interface Searcher() {*

*State getType();*

*Page<Booking> search(Long userId, State state, Pageable pageable,*

*LocalDateTime dateTime, BookingRepository bookingRepository); }*

Потом создадим элементы, реализующие его:

*public class SearcherByBookerIdAndStateAll implements Searcher {*

*@Override*

*public State getType() {*

*return State.ALL;*

*@Override*

*public List<Booking> search(Long userId, State state, Pageable pageable,*

*LocalDateTime dateTime, BookingRepository bookingRepository) {*

*return bookingRepository.findByBookerIdOrderByStartDesc(userId, pageable).toList(); } }*

Соберем созданные элементы в хеш-мапу:

*Searcher searcherAll = new SearcherByBookerIdAndStateAll();*

*Searcher searcherCurrent = new SearcherByBookerIdAndStateCurrent();*

*Searcher searcherPast = new SearcherByBookerIdAndStatePast();*

*Searcher searcherFuture = new SearcherByBookerIdAndStateFuture();*

*Searcher searcherWaiting = new SearcherByBookerIdAndStateWaiting();*

*Searcher searcherRejected = new SearcherByBookerIdAndStateRejected();*

*private final Map<State, Searcher> searcherByBooker = new HashMap()<>;*

*searcherByBooker.put(searcherAll.getType(), searcherAll);*

*searcherByBooker.put(searcherCurrent.getType(), searcherCurrent);*

*searcherByBooker.put(searcherPast.getType(), searcherPast);*

*searcherByBooker.put(searcherFuture.getType(), searcherFuture);*

*searcherByBooker.put(searcherWaiting.getType(), searcherWaiting);*

*searcherByBooker.put(searcherRejected.getType(), searcherRejected);*

*// в зависимости от state будет вызываться нужная реализация*

*Page<booking> bookings = searcherByBooker.get(state)*

*.search(userId, state, pageable, dateTime, bookingRepository);*