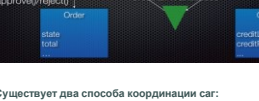


Паттерны	
Что такое «шаблон проектирования»?	Проверенное и готовое к использованию логическое решение, которое может быть реализовано по-разному в разных языках программирования. Плюсы: снижение сложности разработки за счёт готовых абстракций облегчение коммуникации между разработчиками Минусы: слепое следование некоторому 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 данного шаблона; Задача, которую шаблон позволяет решить; Способ решения, предлагаемый в шаблоне для решения задачи в том контексте, где этот шаблон был найден; Участиями - сущности, принимающие участие в решении задачи; Следствия от использования шаблона как результат действий, выполняемых в шаблоне; Реализация - возможный вариант реализации шаблона.
Назовите три основные группы паттернов.	Порождающие - отвечают за удобное и безопасное создание новых объектов или даже целых семейств объектов без внесения в программу лишних зависимостей. Структурные - отвечают за построение удобных в поддержке иерархий классов Поведенческие - занимаются об эффективной коммуникации между объектами. Основные - основные строительные блоки, используемые для построения других шаблонов. Например, интерфейс.
Расскажите про паттерн Одиночка (Singleton).	Порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа. Конструктор помечается как private, а для создания нового объекта Singleton использует специальный метод getInstance(). Он либо создаёт объект, либо отдаёт существующий объект, если он уже был создан. private static Singleton instance; public static Singleton getInstance() { if (instance == null) { instance = new Singleton(); } return instance; } +: можно не создавать множество объектов для ресурсоемких задач, а пользоваться одним -: нарушает принцип одной ответственности, так как его могут использовать множество объектов
	Почему считается антипаттерном? -Нельзя тестировать с помощью mock, но можно использовать powerMock. -Нарушает принцип единой ответственности -Нарушает Open/Close принцип, его нельзя расширить Можно ли его синхронизировать без synchronized у метода? -Можно сделать его Enum (enum). Это статический final класс с константами. JVM загрузит final и static классы на этапе компиляции, а значит несколько потоков не могут создать несколько инстансов. -С помощью double checked locking (lazy). Synchronized внутри метода: private static volatile Singleton instance; public static Singleton getInstance() { Singleton localInstance = instance; if (localInstance == null) { synchronized (Singleton.class) { localInstance = instance; if (localInstance == null) { // second check instance = localInstance = new Singleton(); } } } return localInstance; }
Расскажите про паттерн Строитель (Builder).	Порождающий паттерн, который позволяет создавать сложные объекты пошагово. Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений одного объекта. Паттерн предлагает вынести конструирование объекта за пределы его собственного класса, поручив это дело отдельным объектам, называемым строителями. Процесс конструирования объекта разбить на отдельные шаги (например, построить Стены, поставить Двери). Чтобы создать объект, вам нужно поочередно вызывать методы строителя. Причём не нужно запускать все шаги, а только те, что нужны для производства объекта определённой конфигурации. Можно пойти дальше и выделить вызовы методов строителя в отдельный класс, называемый директором. В этом случае директор будет задавать порядок шагов строительства, а строитель — выполнять их. +: Позволяет использовать один и тот же код для создания различных объектов. Изолирует сложный код сборки объектов от его основной бизнес-логики. -: Усложняет код программы из-за введения дополнительных классов.
Расскажите про паттерн Фабричный метод (Factory Method).	Порождающий шаблон проектирования, в котором предоставляет интерфейс для создания объектов в родительском классе, но позволяет подклассам изменять тип создаваемых объектов. Подклассы имплементируют общий интерфейс с методом для создания объектов. Переопределённый метод в каждом наследнике возвращает нужный вариант объекта. Объекты всё равно будут создаваться при помощи new, но делать это будет фабричный метод. Таким образом можно переопределить фабричный метод в подклассах, чтобы изменить тип создаваемого продукта. Чтобы эта система заработала, все возвращаемые объекты должны иметь общий интерфейс. Подклассы смогут производить объекты различных классов, следуя одному и тому же интерфейсу. +: Выделяет код производства объектов в одно место, упрощая поддержку кода. Реализует принцип открытости/закрытости. -: Может привести к созданию больших параллельных иерархий классов, так как для каждого класса продукта надо создать свой подкласс создателя. Пример: у нас есть интерфейс "Разработчик" и его реализация в виде классов "Разработчик" но под каждую реализацию нужно создать производителя для этого создаем еще один интерфейс ютубщик будет иметь всего один фабричный метод.

<p>Какие паттерны используются в Spring Framework?</p> <p>Singleton - Bean scopes Factory - Bean factories classes Prototype - Bean scopes Adapter - Spring Web and Spring MVC Proxy - Spring Aspect Oriented Programming support Template Method - JdbcTemplate, HibernateTemplate etc Front Controller - Spring MVC DispatcherServlet DAO - Spring Data Access Object support Dependency Injection</p>					
<p>Какие паттерны используются в Hibernate?</p> <p>Domain Model – объектная модель предметной области, включающая в себя как поведение так и данные. 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</p>					
<p>Шаблоны GRASP: Low Coupling (низкая связанность) и High Cohesion (высокая сплоченность)</p> <p>Low Coupling - части системы, которые изменяются вместе, должны находиться близко друг к другу. Необходимо распределить ответственности между классами так, чтобы обеспечить минимальную связанность. High Cohesion - если возвести Low Coupling в абсолют, то можно прийти к тому, чтобы разместить всю функциональность в одном единственном классе. Классы должны содержать связанную бизнес-логику. В таком случае связей не будет вообще, но что-то тут явно не так, ведь в этот класс попадет совершенно несвязанная между собой бизнес-логика. Принцип High Cohesion говорит следующее: части системы, которые изменяются параллельно, должны иметь как можно меньше зависимостей друг на друга.</p>					
<p>Расскажите про паттерн Saga</p> <p>Saga — это механизм, обеспечивающий согласованность данных в микросервисах без применения распределенных транзакций.</p> <p>Для каждой системной команды, которой надо обновлять данные в нескольких сервисах, создается некоторая saga. Saga представляет из себя некоторый «чек-лист», состоящий из последовательных локальных ACID-транзакций, каждая из которых обновляет данные в одном сервисе. Для обработки сбоев применяется компенсирующая транзакция. Такие транзакции выполняются в случае сбоя на всех сервисах, на которых локальные транзакции выполнялись успешно.</p> <p>Типов транзакций в saga четыре: Компенсирующая — отменяет изменение, сделанное локальной транзакцией. Компенсируемая — это транзакция, которую необходимо компенсировать (отменить) в случае, если последующие транзакции завершаются неудачей. Поворотная — транзакция, определяющая успешность всей saga. Если она выполняется успешно, то saga гарантировано дойдет до конца. Повторяемая — идет после поворотной и гарантировано завершается успехом.</p>	<p>Option #1: Choreography-based coordination using events</p>  <p>Существует два способа координации sag: Хореография (Choreography) — каждая транзакция публикует события, которые запускают транзакции в других сервисах. Оркестровка (Orchestration) — оркестратор говорит участникам, какие транзакции должны быть запущены.</p>				
<p>Пример нарушения</p>					
<p>Самым ярким примером нарушения этого принципа, с моей точки зрения, является циклическая зависимость (да-да, то, за что обычно отрываю руки на code review):</p>					
<p>Пример нарушения</p> <p>Давайте рассмотрим класс, представляющий из себя данные с какого-либо счетчика:</p> <pre> public class A { private int a; private B b; public A(int a) { this.a = a; this.b = new B(this); } } public class B { private A a; public B(A a) { this.a = a; } } </pre>	<pre> @AllArgsConstructor public class Data { private TemperatureData temperatureData; private TimeData timeData; public Data(int time, int temperature) { this.temperatureData = new TemperatureData(temperature); this.timeData = new TimeData(time); } // тут логика по работе как со временем, так и с температурой } @AllArgsConstructor public class TimeData { private int time; private int calculateTimeDifference(int time) { return this.time - time; } } @AllArgsConstructor public class TemperatureData { private int temperature; private int calculateTemperatureDifference(int temperature) { return this.temperature - temperature; } } </pre>				