**2)Шаблон проектирования** (*паттерн*) — повторяемая архитектурная конструкция в сфере проектирования программного обеспечения, предлагающая решение проблемы проектирования в рамках некоторого часто возникающего контекста.

Обычно шаблон не является законченным образцом, который может быть прямо преобразован в [код](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0); это лишь пример решения задачи, который можно использовать в различных ситуациях.

**3)Зачем знать паттерны?**

* *Проверенные решения*. Вы тратите меньше времени, используя готовые решения, вместо изобретения велосипеда.
* *Стандартизация кода*. Вы делаете меньше просчетов при разработке, используя типовые решения, так как все возможные проблемы в них уже найдены.
* *Общий программистский словарь*. Вместо того, чтобы долго объяснять коллеге логику работы вашего кода, вы называете паттерн, который хотите использовать, и экономите свое и чужое время.

**4-5)История создания паттернов:**

В [1970-е](https://ru.wikipedia.org/wiki/1970-%D0%B5) годы [архитектор](https://ru.wikipedia.org/wiki/%D0%90%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D0%BE%D1%80) [Кристофер Александер](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B5%D0%BA%D1%81%D0%B0%D0%BD%D0%B4%D0%B5%D1%80,_%D0%9A%D1%80%D0%B8%D1%81%D1%82%D0%BE%D1%84%D0%B5%D1%80) составил набор шаблонов проектирования. В области архитектуры эта идея не получила такого развития, как позже в области программной разработки.

В [1987 году](https://ru.wikipedia.org/wiki/1987_%D0%B3%D0%BE%D0%B4) [Кент Бэк](https://ru.wikipedia.org/wiki/%D0%91%D0%B5%D0%BA,_%D0%9A%D0%B5%D0%BD%D1%82) (Kent Beck) и [Вард Каннингем](https://ru.wikipedia.org/wiki/%D0%9A%D0%B0%D0%BD%D0%BD%D0%B8%D0%BD%D0%B3%D0%B5%D0%BC,_%D0%92%D0%B0%D1%80%D0%B4) (Ward Cunningham) взяли идеи Александера и разработали шаблоны применительно к разработке программного обеспечения для разработки графических оболочек на языке [Smalltalk](https://ru.wikipedia.org/wiki/Smalltalk" \o "Smalltalk).

В [1988 году](https://ru.wikipedia.org/wiki/1988_%D0%B3%D0%BE%D0%B4) [Эрих Гамма](https://ru.wikipedia.org/wiki/%D0%93%D0%B0%D0%BC%D0%BC%D0%B0,_%D0%AD%D1%80%D0%B8%D1%85) (Erich Gamma) начал писать докторскую диссертацию при Цюрихском университете об общей переносимости этой методики на разработку программ.

В 1989—1991 годах Джеймс Коплин (James Coplien) трудился над разработкой идиом для программирования на [C++](https://ru.wikipedia.org/wiki/C%2B%2B) и опубликовал в 1991 году книгу «Advanced C++ Idioms».

В этом же году Эрих Гамма заканчивает свою докторскую диссертацию и переезжает в [США](https://ru.wikipedia.org/wiki/%D0%A1%D0%A8%D0%90), где в сотрудничестве с Ричардом Хелмом (Richard Helm), Ральфом Джонсоном (Ralph Johnson) и [Джоном Влиссидесом](https://ru.wikipedia.org/wiki/%D0%92%D0%BB%D0%B8%D1%81%D0%B8%D0%B4%D0%B8%D1%81,_%D0%94%D0%B6%D0%BE%D0%BD) (John Vlissides) публикует книгу [Design Patterns — Elements of Reusable Object-Oriented Software](https://ru.wikipedia.org/wiki/Design_Patterns). В этой книге описаны 23 шаблона проектирования. Также команда авторов этой книги известна общественности под названием «Банда четырёх» ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Gang of Four*, часто сокращается до *GoF*). Именно эта книга стала причиной роста популярности шаблонов проектирования.

**6) три основные группы паттернов:**

**Поражающие паттерны (или порождающие паттерны) нужны для:**

1. **Упрощения создания объектов**: Они помогают упростить процесс создания экземпляров классов, особенно когда нужно учитывать разные параметры или условия.
2. **Улучшения читаемости кода**: Использование паттернов делает код более понятным и структурированным, что облегчает его сопровождение.
3. **Снижения связанности**: Порождающие паттерны помогают уменьшить зависимость между классами, позволяя изменять код без необходимости менять все его части.
4. **Упрощения тестирования**: С помощью паттернов можно легче создавать подмены и моки, что облегчает тестирование компонентов.
5. **Гибкости и расширяемости**: Они позволяют легко добавлять новые виды объектов без изменения существующего кода, что делает систему более адаптивной к изменениям.

**Структурные паттерны нужны для:**

1. **Упрощения организации классов и объектов**: Они помогают создавать сложные структуры из простых компонентов.
2. **Уменьшения связанности**: Паттерны способствуют снижению зависимости между классами, что упрощает изменения и сопровождение кода.
3. **Повышения повторного использования**: Позволяют использовать существующие классы и объекты в новых контекстах без дублирования кода.
4. **Улучшения читаемости**: Делают архитектуру приложения более понятной и структурированной.
5. **Расширяемости**: Облегчают добавление новых функциональностей без изменения существующего кода

**Поведенческие паттерны нужны для:**

1. **Управления взаимодействием между объектами**: Они помогают определить, как объекты должны взаимодействовать друг с другом.
2. **Упрощения сложной логики**: Позволяют разбивать сложные поведения на более простые и управляемые части.
3. **Увеличения гибкости**: Облегчают изменение поведения объектов без изменения их кода.
4. **Повышения повторного использования**: Предоставляют возможности для использования одних и тех же функций в разных контекстах.
5. **Улучшения читаемости**: Делают код более понятным, описывая поведение через паттерны, а не через сложные условные конструкции.

**7) SOLID**

 (сокр. от [англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) **single responsibility**, **open–closed**, **Liskov substitution**, **interface segregation** и **dependency inversion**) пять принципов, названных [Робертом Мартином](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%80%D1%82%D0%B8%D0%BD,_%D0%A0%D0%BE%D0%B1%D0%B5%D1%80%D1%82_(%D0%B8%D0%BD%D0%B6%D0%B5%D0%BD%D0%B5%D1%80))[[1]](https://ru.wikipedia.org/wiki/SOLID_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)#cite_note-ub-old-web-solid-1)[[2]](https://ru.wikipedia.org/wiki/SOLID_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)#cite_note-ub-solid-2) в начале [2000-х](https://ru.wikipedia.org/wiki/2000-%D0%B5_%D0%B3%D0%BE%D0%B4%D1%8B)[[3]](https://ru.wikipedia.org/wiki/SOLID_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)#cite_note-metz-presentation-2009-3), которые означали 5 основных принципов объектно-ориентированных [проектирования](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) и [программирования](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5). Избавиться от «признаков плохого проекта»[[4]](https://ru.wikipedia.org/wiki/SOLID_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)#cite_note-:0-4) помогают следующие принципы SOLID:

* **SRP**(single responsibility principle)- Принцип единственной ответственности
* **OCP** (open-closed principle)- Принцип открытости/закрытости
* **LSP**((Liskov substitution principle)- Принцип подстановки Лисков
* **ISP**(interface segregation principle)- Принцип разделения интерфейса
* **DIP**(dependency inversion principle)- Принцип инверсии зависимостей

**8) SRP(single responsibility principle)- Принцип единственной ответственности**

У класса должна быть только одна причина для изменения. Или, как подметил Р. Мартин в книге «Чистая архитектура. Искусство разработки программного обеспечения», когда описывал эволюцию данного определения, окончательная версия принципа единственной ответственности выглядит так: «Модуль должен отвечать за одного и только за одного актора». Актор(актер)-сущность, с которым взаимодействует система.

**9) OCP (open-closed principle)- Принцип открытости/закрытости**

Изменение текущего поведения класса повлияет на все остальные компоненты, использующие этот класс. Этот принцип направлен на расширение функциональности класса без изменения его текущего поведения. Это сделано для того, чтобы избежать появления ошибок везде, где используется класс.

**10) LSP((Liskov substitution principle)- Принцип подстановки Лисков**

Когда дочерний класс не может выполнять действия, которые заложены в родительском классе, это может вызвать ошибки.

Если у вас есть класс и вы создаете из него другой класс, то он становится родительским, а новый класс - дочерним. Дочерний класс должен уметь делать все, что может делать родительский класс. Этот процесс называется наследованием.

Дочерний класс должен уметь обрабатывать те же запросы и возвращать тот же результат, что и родительский класс, либо он может возвращать результат того же типа.

На рисунке видно, что родительский класс приносит кофе. При этом дочерний класс может принести капучино, потому что это один из видов кофе. Но его не поймут, если вместо кофе он принесет воду.

Если дочерний класс не соответствует указанным требованиям, это означает, что он полностью изменен и нарушает принцип Барбары Лисков.

Этот принцип направлен на обеспечение согласованности родительского и дочернего классов, чтобы они могли использоваться одинаково и без каких-либо ошибок.

**11) ISP(interface segregation principle)- Принцип разделения интерфейса**

Когда класс используется для выполнения действий, которые не являются полезными, это расточительно и может привести к неожиданным ошибкам, так как класс может не уметь выполнять эти действия.

Класс должен выполнять только те действия, которые необходимы для выполнения его роли. Любое другое действие должно быть полностью удалено или перенесено в другое место, если в будущем оно может быть использовано другим классом. Этот принцип направлен на разделение набора действий на более мелкие наборы, чтобы класс выполнял только **тот** набор действий, который необходим для его работы.

Этот принцип направлен на разделение набора действий на более мелкие наборы, чтобы класс выполнял только **тот** набор действий, который необходим для его работы.

**12-13) DIP(dependency inversion principle)- Принцип инверсии зависимостей**

*Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.*

*Абстракции не должны зависеть от деталей. Детали должны зависеть от абтракций.*

Во-первых, давайте опишем термины, используемые в формулировке принципа более простыми словами.

**Модуль верхнего уровня** - класс, который выполняет действие с помощью инструмента.  
**Модуль нижнего уровня** - инструмент, необходимый для выполнения действия.  
**Абстракция** - интерфейс, который соединяет два класса (модуля).  
**Детали** - то, как работает инструмент.

Этот принцип говорит, что класс не следует объединять с инструментом, который он использует для выполнения действия. Скорее, он должен быть объединен с интерфейсом, который позволит инструменту подключаться к классу.

Это также говорит о том, что и класс, и интерфейс не должны знать, как работает инструмент. Однако инструмент должен соответствовать спецификации интерфейса.  
Этот принцип направлен на уменьшение зависимости класса высокого уровня от класса низкого уровня путем введения интерфейса.

**15-16)Применение порождающих паттернов проектирования**

Порождающие предназначены для создания экземпляра объекта или группы связанных объектов.К ним относятся такие паттерны как:

* Abstract Factory — Абстрактная фабрика
* Builder — Строитель- это порождающий паттерн проектирования, который позволяет создавать сложные объекты пошагово. Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений объектов.
* Factory Method — Фабричный метод- это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.
* Prototype — Прототип- это порождающий паттерн проектирования, который позволяет копировать объекты, не вдаваясь в подробности их реализации.
* Singleton — Одиночка- это порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

**17) Абстрактная фабрика**

Абстрактная фабрика -это порождающий паттерн проектирования, который позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.

Представим, что вы пишем симулятор мебельного магазина. Ваш код содержит:

1. Семейство зависимых продуктов. Скажем, Кресло + Диван + Столик.
2. Несколько вариаций этого семейства. Например, продукты Кресло, Диван и Столик представлены в трёх разных стилях: Ар-деко, Викторианском и Модерне.

Нам нужен такой способ создавать объекты продуктов, чтобы они сочетались с другими продуктами того же семейства. Это важно, так как клиенты расстраиваются, если получают несочетающуюся мебель.

Кроме того, мы не хотим вносить изменения в существующий код при добавлении новых продуктов или семейcтв в программу. Поставщики часто обновляют свои каталоги, и мы бы не хотели менять уже написанный код каждый раз при получении новых моделей мебели.

Для начала паттерн Абстрактная фабрика предлагает выделить общие интерфейсы для отдельных продуктов, составляющих семейства. Так, все вариации кресел получат общий интерфейс Кресло, все диваны реализуют интерфейс Диван и так далее.

Далее мы создаём абстрактную фабрику — общий интерфейс, который содержит методы создания всех продуктов семейства (например, создатьКресло, создатьДиван и создатьСтолик). Эти операции должны возвращать абстрактные типы продуктов, представленные интерфейсами, которые мы выделили ранее — Кресла, Диваны и Столики.

## **18)  Преимущества и недостатки**

* Гарантирует сочетаемость создаваемых продуктов.
* Избавляет клиентский код от привязки к конкретным классам продуктов.
* Выделяет код производства продуктов в одно место, упрощая поддержку кода.
* Упрощает добавление новых продуктов в программу.
* Реализует принцип открытости/закрытости.

Недостатки:

* Усложняет код программы из-за введения множества дополнительных классов.
* Требует наличия всех типов продуктов в каждой вариации.

На фотографии показана структура абстрактной фабрики.

**19-21)Применение структурных паттернов**

Структурные паттерны проектирования помогают организовать классы и объекты для формирования больших и сложных систем. Они упрощают проектирование, делая его более гибким и эффективным.

Адаптер — это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе.

Мост — это структурный паттерн проектирования, который разделяет один или несколько классов на две отдельные иерархии — абстракцию и реализацию, позволяя изменять их независимо друг от друга.

Компоновщик — это структурный паттерн проектирования, который позволяет сгруппировать множество объектов в древовидную структуру, а затем работать с ней так, как будто это единичный объект.

Декоратор — это структурный паттерн проектирования, который позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».

Фасад — это структурный паттерн проектирования, который предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку.

Легковес — это структурный паттерн проектирования, который позволяет вместить бóльшее количество объектов в отведённую оперативную память. Легковес экономит память, разделяя общее состояние объектов между собой, вместо хранения одинаковых данных в каждом объекте.

Заместитель — это структурный паттерн проектирования, который позволяет подставлять вместо реальных объектов специальные объекты-заменители. Эти объекты перехватывают вызовы к оригинальному объекту, позволяя сделать что-то *до* или *после* передачи вызова оригиналу.

**22)Рассмотрим структурные паттерны на примере паттерна Адаптер**

**Адаптер** — это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе..

Представьте, что мы делаем приложение для торговли на бирже. Наша приложение скачивает биржевые котировки из нескольких источников в XML, а затем рисует красивые графики.

В какой-то момент мы решили улучшить приложение, применив стороннюю библиотеку аналитики. Но вот беда — библиотека поддерживает только формат данных JSON, несовместимый с вашим приложением.

Мы смогли бы переписать библиотеку, чтобы та поддерживала формат XML. Но, во-первых, это может нарушить работу существующего кода, который уже зависит от библиотеки. А во-вторых, у нас может просто не быть доступа к её исходному коду.

Мы можем создать *адаптер*. Это объект-переводчик, который трансформирует интерфейс или данные одного объекта в такой вид, чтобы он стал понятен другому объекту.

При этом адаптер оборачивает один из объектов, так что другой объект даже не знает о наличии первого. Например, мы можем обернуть объект, работающий в метрах, адаптером, который бы конвертировал данные в футы.

Адаптеры могут не только переводить данные из одного формата в другой, но и помогать объектам с разными интерфейсами работать сообща. Это работает так:

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.

Иногда возможно создать даже *двухсторонний адаптер*, который работал бы в обе стороны.

## Аналогия из жизни

Когда вы в первый раз летите за границу, вас может ждать сюрприз при попытке зарядить ноутбук. Стандарты розеток в разных странах отличаются. Ваша европейская зарядка будет бесполезна в США без специального адаптера, позволяющего подключиться к розетке другого типа.

**23) Преимущества паттерна адаптер:**

Отделяет и скрывает от клиента подробности преобразования различных интерфейсов.

Недостатки:

Усложняет код программы из-за введения дополнительных классов.

**24-26) Применение поведенческих паттернов  
Поведенческие паттерны проектирования помогают определить, как объекты взаимодействуют и распределяют ответственность друг между другом. Они улучшают коммуникацию между классами и упрощают реализацию сложного поведения.**

**Цепочка обязанностей** — это поведенческий паттерн проектирования, который 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.

**Команда** — это поведенческий паттерн проектирования, который превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций.

**Итератор** — это поведенческий паттерн проектирования, который даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.

**Посредник** — это поведенческий паттерн проектирования, который позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник.

**Снимок** — это поведенческий паттерн проектирования, который позволяет сохранять и восстанавливать прошлые состояния объектов, не раскрывая подробностей их реализации.

**Наблюдатель** — это поведенческий паттерн проектирования, который создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.

**Состояние** — это поведенческий паттерн проектирования, который позволяет объектам менять поведение в зависимости от своего состояния. Извне создаётся впечатление, что изменился класс объекта.

**Стратегия** — это поведенческий паттерн проектирования, который определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы.

**Шаблонный метод** — это поведенческий паттерн проектирования, который определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Паттерн позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.

**Посетитель** — это поведенческий паттерн проектирования, который позволяет добавлять в программу новые операции, не изменяя классы объектов, над которыми эти операции могут выполняться.

# **20)Рассмотрим поведенческие паттерны на примере паттерна “Цепочке обязанностей”**

С каждой новой «фичей» код проверок, выглядящий как большой клубок условных операторов, всё больше и больше раздувался. При изменении одного правила приходилось трогать код всех проверок. А для того, чтобы применить проверки к другим ресурсам, пришлось продублировать их код в других классах.

Поддерживать такой код стало не только очень хлопотно, но и затратно. И вот в один прекрасный день вы получаете задачу рефакторинга...

Как и многие другие поведенческие паттерны, Цепочка обязанностей базируется на том, чтобы превратить отдельные поведения в объекты. В нашем случае каждая проверка переедет в отдельный класс с единственным методом выполнения. Данные запроса, над которым происходит проверка, будут передаваться в метод как аргументы.

Но есть подход, при котором обработчики прерывают цепь только когда они могут обработать запрос. В этом случае запрос движется по цепи, пока не найдётся обработчик, который может его обработать. Очень часто такой подход используется для передачи событий, создаваемых классами графического интерфейса в результате взаимодействия с пользователем.

Например, когда пользователь кликает по кнопке, программа выстраивает цепочку из объекта этой кнопки, всех её родительских элементов и общего окна приложения на конце. Событие клика передаётся по этой цепи до тех пор, пока не найдётся объект, способный его обработать. Этот пример примечателен ещё и тем, что цепочку всегда можно выделить из древовидной структуры объектов, в которую обычно и свёрнуты элементы пользовательского интерфейса.

## Аналогия из жизни

Мы купили новую видеокарту. Она автоматически определилась и заработала под Windows, но в вашей любимой Ubuntu «завести» её не удалось. Со слабой надеждой вы звоните в службу поддержки.

Первым мы слышим голос автоответчика, предлагающий выбор из десятка стандартных решений. Ни один из вариантов не подходит, и робот соединяет вас с живым оператором.

Увы, но рядовой оператор поддержки умеет общаться только заученными фразами и давать шаблонные ответы. После очередного предложения «выключить и включить компьютер» мы просим связать вас с настоящими инженерами.

Оператор перебрасывает звонок дежурному инженеру, изнывающему от скуки в своей каморке. Уж он-то знает, как нам помочь! Инженер рассказывает нам, где скачать подходящие драйвера и как настроить их под Ubuntu. Запрос удовлетворён. Мы кладём трубку.

**21) Преимущества паттерна цепочка обязанностей:**

* Уменьшает зависимость между клиентом и обработчиками.
* Реализует принцип единственной обязанности.
* Реализует принцип открытости/закрытости.

Недостатки:

* Запрос может остаться никем не обработанным.

На фотографии структура Цепочки обязанностей

**29) «Низкоуровневые» шаблоны**

«Низкоуровневые» шаблоны, учитывающие специфику конкретного языка программирования, называются идиомами. Это хорошие решения проектирования, характерные для конкретного языка или программной платформы, и потому не универсальные.

**Идиома** программирования — устойчивый способ выражения некоторой составной конструкции в одном или нескольких языках программирования. Идиома является шаблоном решения задачи, записи алгоритма или структуры данных путём комбинирования встроенных элементов языка.

**23) Примеры простых идиом:**

* **Инкремент**. В языках типа [BASIC](https://ru.wikipedia.org/wiki/BASIC) идиома для единичного приращения значения переменной выглядит так: i=i+1;
* **Обмен значениями (swap):**

**Вопросы:**

1)1 2)2 3)2 4)1 5)3 6)1 7)4 8)2 9)3

### Заключение

### (Текст слайда)

### Дополнительно:

### 1. **Упрощение разработки**

Паттерны помогают разработчикам избегать повторения ошибок, предлагая готовые решения, которые уже были протестированы и доказали свою эффективность.

### 2. **Улучшение читаемости и поддержки кода**

Использование общепринятых паттернов делает код более понятным для других разработчиков, что упрощает его поддержку и развитие.

### 3. **Гибкость и масштабируемость**

Паттерны позволяют создавать более гибкие архитектуры, которые легче адаптировать к изменениям требований или добавлению новых функций.

### 4. **Командная работа**

Общие знания о паттернах проектирования в команде способствуют лучшему пониманию архитектуры проекта, что облегчает совместную работу и коммуникацию между разработчиками.

### 5. **Не универсальное решение**

Хотя паттерны проектирования очень полезны, их следует применять с осторожностью. Неправильное или чрезмерное использование может привести к усложнению системы и снижению производительности.