Оглавление

[ЭКЗАМЕНАЦИОННЫЙ БИЛЕТ №1 2](#_Toc160569965)

[ЭКЗАМЕНАЦИОННЫЙ БИЛЕТ №2 4](#_Toc160569966)

[ЭКЗАМЕНАЦИОННЫЙ БИЛЕТ №3 6](#_Toc160569967)

[ЭКЗАМЕНАЦИОННЫЙ БИЛЕТ №4 8](#_Toc160569968)

[ЭКЗАМЕНАЦИОННЫЙ БИЛЕТ №5 10](#_Toc160569969)

[ЭКЗАМЕНАЦИОННЫЙ БИЛЕТ №6 12](#_Toc160569970)

[ЭКЗАМЕНАЦИОННЫЙ БИЛЕТ №7 14](#_Toc160569971)

[ЭКЗАМЕНАЦИОННЫЙ БИЛЕТ №8 16](#_Toc160569972)

[ЭКЗАМЕНАЦИОННЫЙ БИЛЕТ №9 18](#_Toc160569973)

[ЭКЗАМЕНАЦИОННЫЙ БИЛЕТ №10 20](#_Toc160569974)

[ЭКЗАМЕНАЦИОННЫЙ БИЛЕТ №11 22](#_Toc160569975)

[ЭКЗАМЕНАЦИОННЫЙ БИЛЕТ №12 24](#_Toc160569976)

[ЭКЗАМЕНАЦИОННЫЙ БИЛЕТ №13 26](#_Toc160569977)

[ЭКЗАМЕНАЦИОННЫЙ БИЛЕТ №14 28](#_Toc160569978)

[ЭКЗАМЕНАЦИОННЫЙ БИЛЕТ №15 30](#_Toc160569979)

# ЭКЗАМЕНАЦИОННЫЙ БИЛЕТ №1

Вопрос №1. Что такое жизненный цикл программного обеспечения и какие фазы включает?

Жизненный цикл программного обеспечения (ЖЦ ПО) - это последовательность фаз и процессов, через которые проходит программное обеспечение от начальной концепции до его вывода из эксплуатации. ЖЦ ПО включает следующие основные фазы:

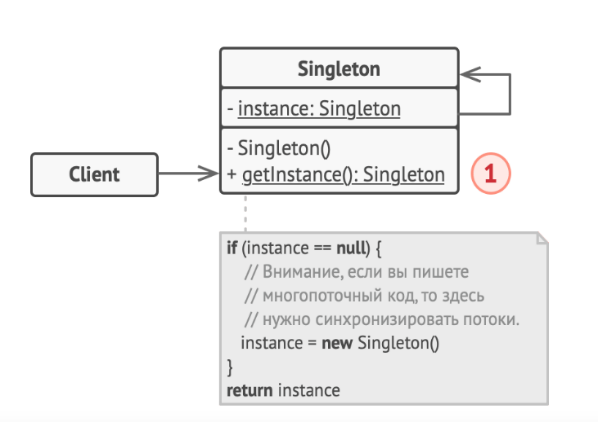
* **Анализ и планирование:** В этой фазе определяются требования к программному обеспечению, проводится анализ рисков, определяются бизнес-цели и планируется весь процесс разработки.
* **Проектирование:** В этой фазе создается архитектура программного обеспечения, определяются компоненты системы, их взаимодействие и интерфейсы. Проектирование также включает выбор технологий и инструментов разработки.
* **Разработка:** В этой фазе программное обеспечение фактически создается. Разработчики пишут код, выполняют тестирование и интеграцию компонентов, создают документацию и проводят отладку.
* **Тестирование:** В этой фазе проводятся различные виды тестирования, включая модульное, интеграционное и системное тестирование. Цель - обнаружить и исправить ошибки и проверить соответствие требованиям.
* **Внедрение и сопровождение**: После успешного тестирования программное обеспечение готово к внедрению в реальную среду. Затем начинается сопровождение, которое включает обновления, исправление ошибок и поддержку в течение всего жизненного цикла ПО.

Вопрос №2. Каковы основные принципы разработки программного обеспечения?

* **Принцип модульности**: Разделение программного обеспечения на независимые модули, которые выполняют конкретные функции. Модульность позволяет упростить разработку, тестирование и сопровождение, а также повторное использование кода.
* **Принцип единственной ответственности**): Каждый модуль или компонент программного обеспечения должен быть ответственен за выполнение одной конкретной задачи. Это облегчает понимание и изменение кода, а также повышает его гибкость.
* **Принцип открытости/закрытости**: Программное обеспечение должно быть открыто для расширения новыми функциями, но закрыто для изменения существующего кода. Это достигается путем использования абстракций, интерфейсов и наследования.
* **Принцип подстановки Барбары Лисков**: Объекты программного обеспечения должны быть заменяемыми экземплярами своих базовых типов. Это означает, что код, использующий базовый тип, должен работать корректно и с подклассами этого типа.
* **Принцип инверсии зависимостей**: Зависимости между модулями должны быть организованы таким образом, чтобы высокоуровневые модули не зависели от низкоуровневых модулей. Вместо этого, они должны зависеть от абстракций. Это способствует гибкости и повторному использованию кода.
* **Принцип единого стиля кодирования**: Согласованность стиля кодирования в рамках проекта или организации помогает обеспечить читаемость и понимание кода всем членам команды разработчиков. Это включает форматирование, именование переменных и функций, комментарии и документирование кода.

**Практическое задание:**

Реализуйте шаблон проектирования Singleton “Одиночка” на любом удобном языке (С++/Java/C#).



public class Singleton {

private static Singleton instance;

// Приватный конструктор, чтобы предотвратить создание экземпляров класса извне

private Singleton() {

}

// Метод для получения экземпляра класса

public static Singleton getInstance() {

if (instance == null) {

// Создаем экземпляр только при первом вызове

instance = new Singleton();

}

return instance;

}

// Другие методы класса

public void someMethod() {

// Реализация метода

}

}

В данном примере класс Singleton имеет приватное статическое поле instance, которое хранит единственный экземпляр класса. Конструктор класса Singleton объявлен как приватный, что предотвращает создание экземпляров класса извне.

Метод getInstance() проверяет, существует ли уже экземпляр класса. Если экземпляр еще не создан, то он создается при первом вызове метода getInstance(), и затем возвращается. При последующих вызовах метода getInstance() уже созданный экземпляр класса возвращается непосредственно.

Теперь, чтобы получить экземпляр класса Singleton, можно вызвать метод getInstance() следующим образом:

java

Copy

Singleton singleton = Singleton.getInstance();

Таким образом, мы можем удостовериться, что в программе существует только один экземпляр класса Singleton.

Обратите внимание, что данная реализация шаблона Singleton не является потокобезопасной. Если вы использовать эту реализацию в многопоточной среде, возможны проблемы с созданием нескольких экземпляров класса. Для обеспечения потокобезопасности рекомендуется использовать двойную проверку на null или синхронизированный блок.

# ЭКЗАМЕНАЦИОННЫЙ БИЛЕТ №2

Вопрос №1. Какие классы алгоритмов существуют, и какие задачи являются неразрешимыми?

* **Алгоритмы сортировки**: Эти алгоритмы используются для упорядочивания элементов в заданном наборе данных. Некоторые из наиболее известных алгоритмов сортировки включают сортировку пузырьком, сортировку вставками, сортировку слиянием и быструю сортировку.
* **Алгоритмы поиска:** Эти алгоритмы используются для нахождения элемента или определенного условия в наборе данных. Примеры алгоритмов поиска включают линейный поиск, бинарный поиск и алгоритмы поиска в графах, такие как поиск в ширину и поиск в глубину.
* **Алгоритмы графов:** Эти алгоритмы используются для работы с графами, которые представляют связи между объектами. Примеры алгоритмов графов включают алгоритм Дейкстры для нахождения кратчайшего пути и алгоритм Флойда-Уоршелла для нахождения всех кратчайших путей в графе.
* **Алгоритмы динамического программирования:** Эти алгоритмы используются для решения задач, которые можно разбить на подзадачи, включающие в себя переиспользование промежуточных результатов. Примеры алгоритмов динамического программирования включают рюкзаковую задачу и нахождение наибольшей общей подпоследовательности.

**Касательно неразрешимых задач,** это задачи, для которых не существует алгоритма, способного решить их для всех возможных входных данных. То есть, неразрешимые задачи не могут быть решены алгоритмически.одной из самых известных является проблема останова (Halting Problem). Она заключается в определении, можно ли для произвольной программы и входных данных точно предсказать, остановится ли программа или будет работать вечно. Алан Тьюринг доказал, что проблема останова является неразрешимой для общего случая.

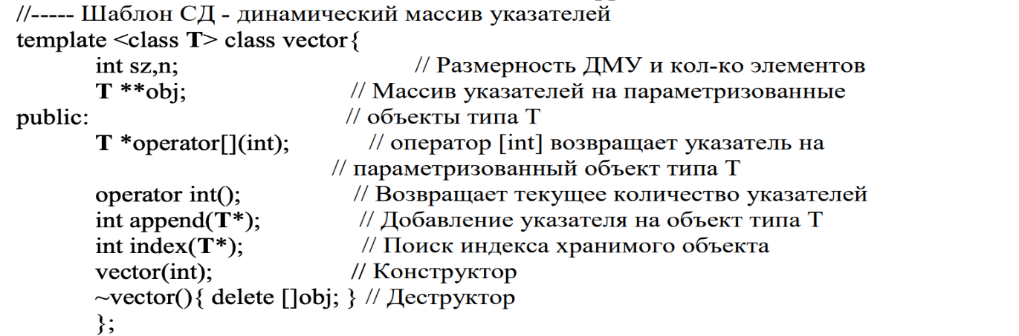
Вопрос №2. Что такое виртуальные классы

**Виртуальные классы** - это механизм, используемый в некоторых объектно-ориентированных языках программирования для реализации полиморфизма. Виртуальный класс - это базовый класс, который определяет интерфейс (методы) и общую функциональность для производных классов. Производные классы могут переопределить методы виртуального класса и предоставить свою реализацию

* В языке программирования C++, например, виртуальные классы объявляются с использованием ключевого слова "virtual". Когда производный класс наследует виртуальный класс, он может предоставить свою собственную реализацию для виртуальных методов, а также иметь свои собственные дополнительные методы и поля
* Использование виртуальных классов позволяет обрабатывать объекты производных классов через указатель на базовый класс, что способствует полиморфизму и гибкости в программировании.

**Практическое задание:**

Реализуйте метод «int append(T\*)» из шаблона класса vector



#include <iostream>

#include <memory>

template<typename T>

class vector {

private:

T\* data;

size\_t capacity;

size\_t size;

public:

vector() : data(nullptr), capacity(0), size(0) {}

~vector() {

delete[] data;

}

int append(T\* item) {

if (size >= capacity) {

// Увеличиваем емкость в два раза при необходимости

size\_t newCapacity = (capacity == 0) ? 1 : capacity \* 2;

T\* newData = new T[newCapacity];

// Копируем существующие элементы в новую область памяти

for (size\_t i = 0; i < size; i++) {

newData[i] = data[i];

}

// Освобождаем старую память

delete[] data;

// Перенаправляем указатель на новую область памяти

data = newData;

capacity = newCapacity;

}

// Добавляем новый элемент в конец вектора

data[size++] = \*item;

return size - 1; // Возвращаем индекс добавленного элемента

}

};

int main() {

vector<int> vec;

int item1 = 10;

int item2 = 20;

int index1 = vec.append(&item1);

int index2 = vec.append(&item2);

std::cout << "Index of item1: " << index1 << std::endl;

std::cout << "Index of item2: " << index2 << std::endl;

return 0;

}

В данном примере реализован шаблон класса vector, который содержит метод append(). Метод append() принимает указатель T\* на элемент типа T и добавляет его в конец вектора. Если размер вектора size достигает его текущей емкости capacity, то вектор увеличивает свою емкость в два раза. После этого, существующие элементы копируются в новую область памяти большей емкости, а старая память освобождается. Затем новый элемент добавляется в конец вектора, и его индекс возвращается в качестве результата.

В функции main() создается экземпляр класса vector<int> и добавляются два элемента типа int. Затем выводятся индексы добавленных элементов. Обратите внимание, что данный пример использует динамическое выделение памяти с помощью оператора new[]. Для обеспечения безопасности и управления памятью рекомендуется использовать современные контейнеры и умные указатели, такие как std::vector и std::shared\_ptr из стандартной библиотеки C++.

# ЭКЗАМЕНАЦИОННЫЙ БИЛЕТ №3

Вопрос №1. Что такое системы контроля версий, и какие виды систем контроля версий существуют?

**Системы контроля версий (СКВ)** - это инструменты, используемые разработчиками программного обеспечения для управления изменениями в исходном коде и других файловых ресурсах проекта. Они позволяют отслеживать и контролировать изменения, вносимые в файлы, а также управлять версиями проекта, совместной работой нескольких разработчиков и восстановлением предыдущих состояний проекта.

**Существует несколько видов систем контроля версий:**

1. **Локальные системы контроля версий:** Эти системы хранят все версии файлов и историю изменений непосредственно на локальном компьютере разработчика. Примером локальной СКВ является RCS (Revision Control System).
2. **Централизованные системы контроля версий (ЦСКВ):** В ЦСКВ все файлы и история изменений хранятся на центральном сервере, к которому подключаются разработчики. Они могут скачивать последнюю версию проекта, вносить изменения и отправлять их на сервер. Примеры ЦСКВ включают SVN (Subversion) и Perforce.
3. **Распределенные системы контроля версий (РСКВ):** В РСКВ каждый разработчик имеет полную копию репозитория проекта, включая всю историю изменений. Это позволяет разработчикам работать независимо и синхронизировать изменения между собой. Примеры РСКВ включают Git, Mercurial и Bazaar.

Вопрос №2. Для чего нужен итератор

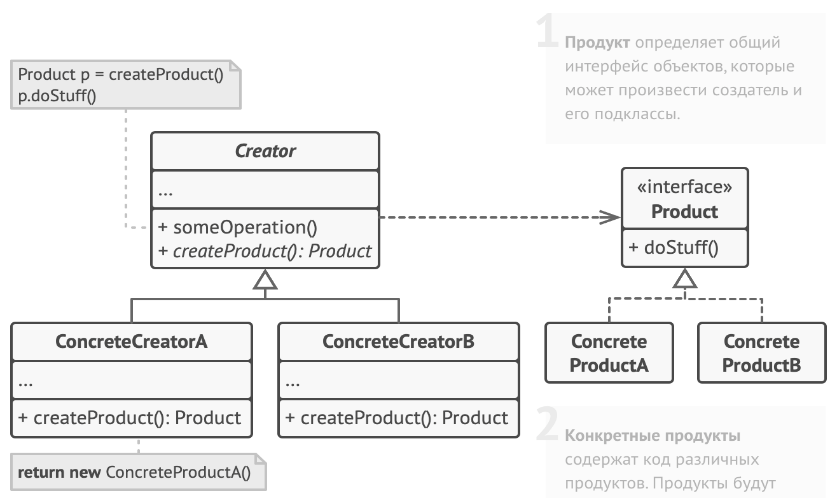
**Итератор** - это объект или механизм, используемый в программировании для последовательного доступа и обхода элементов коллекции или контейнера данных. Итераторы предоставляют удобный способ перебора элементов, скрывая детали реализации структуры данных.

Итераторы позволяют получить доступ к элементам коллекции по одному и обрабатывать их последовательно, не раскрывая внутреннюю структуру коллекции. Они обычно предоставляют методы для перемещения по коллекции, получения текущего элемента и проверки наличия следующего элемента.

**Основная цель использования итераторов** - обеспечить единообразный интерфейс доступа к элементам различных типов коллекций, таких как массивы, списки, деревья и другие. Итераторы позволяют выполнять итерацию по коллекции без необходимости знать ее внутреннюю структуру, что делает код более гибким, модульным и переносимым.

**Практическое задание: ………………**

Реализуйте шаблон проектирования Фабричный метод “Factory method” на любом удобном языке (С++/Java/C#).



#include <iostream>

#include <string>

// Абстрактный класс, представляющий продукт

class Product {

public:

virtual void operation() = 0;

};

// Конкретный класс продукта A

class ConcreteProductA : public Product {

public:

void operation() override {

std::cout << "ConcreteProductA operation" << std::endl;

}

};

// Конкретный класс продукта B

class ConcreteProductB : public Product {

public:

void operation() override {

std::cout << "ConcreteProductB operation" << std::endl;

}

};

// Абстрактный класс, представляющий фабрику

class Factory {

public:

virtual Product\* createProduct() = 0;

};

// Конкретная фабрика для создания продукта A

class ConcreteFactoryA : public Factory {

public:

Product\* createProduct() override {

return new ConcreteProductA();

}

};

// Конкретная фабрика для создания продукта B

class ConcreteFactoryB : public Factory {

public:

Product\* createProduct() override {

return new ConcreteProductB();

}

};

int main() {

Factory\* factoryA = new ConcreteFactoryA();

Product\* productA = factoryA->createProduct();

productA->operation();

Factory\* factoryB = new ConcreteFactoryB();

Product\* productB = factoryB->createProduct();

productB->operation();

delete factoryA;

delete productA;

delete factoryB;

delete productB;

return 0;

}

Он состоит из двух абстрактных классов: Product и Factory, а также их конкретных реализаций: ConcreteProductA, ConcreteProductB, ConcreteFactoryA и ConcreteFactoryB.

Класс Product представляет абстрактный продукт, а его конкретные реализации ConcreteProductA и ConcreteProductB представляют конкретные продукты A и B соответственно. Класс Product имеет виртуальный метод operation(), который должен быть реализован в каждой конкретной реализации продукта.

Класс Factory представляет абстрактную фабрику, а его конкретные реализации ConcreteFactoryA и ConcreteFactoryB представляют конкретные фабрики для создания продуктов A и B соответственно. Класс Factory имеет виртуальный метод createProduct(), который должен быть реализован в каждой конкретной реализации фабрики и возвращает объект типа Product.

В функции main() создаются экземпляры фабрик ConcreteFactoryA и ConcreteFactoryB, затем с их помощью создаются соответствующие продукты ConcreteProductA и ConcreteProductB. Вызывается метод operation() на каждом продукте для выполнения специфических операций.

Обратите внимание, что в данном примере используется динамическое выделение памяти с помощью оператора new. Для обеспечения безопасности и управления памятью рекомендуется использовать современные контейнеры и умные указатели, такие как std::shared\_ptr и std::make\_shared из стандартной библиотеки C++.

# ЭКЗАМЕНАЦИОННЫЙ БИЛЕТ №4

Вопрос №1. Разница между интерфейсом и абстрактным классом

**Интерфейс и абстрактный класс** - это два различных концепта, используемых в объектно-ориентированном программировании для определения контракта и поведения классов. Вот основные различия между ними:

**Интерфейс:**

Интерфейс определяет набор методов и свойств, которые должны быть реализованы классами, которые реализуют данный интерфейс.

* В интерфейсе не может быть определена реализация методов, только их сигнатуры.
* Классы могут реализовывать несколько интерфейсов одновременно.
* Интерфейс может использоваться для достижения полиморфизма, когда различные объекты могут быть обработаны через общий интерфейс.

**Абстрактный класс**:

Абстрактный класс - это класс, который может содержать как абстрактные методы, так и реализацию конкретных методов.

* Абстрактные методы в абстрактном классе не имеют реализации и должны быть переопределены в производных классах.
* Классы могут наследовать только один абстрактный класс, но при этом могут реализовывать интерфейсы.
* Абстрактные классы могут содержать общую реализацию методов, что позволяет избежать дублирования кода в производных классах.

Когда выбрать интерфейс или абстрактный класс зависит от конкретной задачи и дизайна программы. Если вам нужно определить общий контракт для нескольких классов, используйте интерфейс. Если вы хотите предоставить базовую реализацию и определить абстрактные методы, используйте абстрактный класс.

Вопрос №2. Что такое список

**Список** - это структура данных, которая представляет собой упорядоченную коллекцию элементов, в которой каждый элемент имеет свой индекс или позицию. Список позволяет хранить и обрабатывать последовательность значений.

В программировании список обычно реализуется в виде массива или связанного списка. В массиве элементы хранятся в непрерывной области памяти, а доступ к элементам осуществляется по индексам. В связанном списке элементы хранятся в узлах, которые содержат сами значения и ссылки на следующие узлы.

Списки могут быть изменяемыми (mutable) или неизменяемыми (immutable). В изменяемых списках можно добавлять, удалять и изменять элементы после создания списка. В неизменяемых списках элементы не могут быть изменены после создания списка, а операции добавления или удаления элементов создают новый список.

**Практическое задание: ………………**

Сделайте рефакторинг следующей функции методом «Встраивания метода»

**class** **PizzaDelivery** {

// ...

**int** getRating() {

**return** moreThanFiveLateDeliveries() ? 2 : 1;

}

**boolean** moreThanFiveLateDeliveries() {

**return** numberOfLateDeliveries > 5;

}

}

Встраивание метода (Inline Method) - это метод рефакторинга, который заменяет вызов метода его фактическим содержимым. В данном случае, мы можем встроить метод `moreThanFiveLateDeliveries()` в метод `getRating()`. Вот как будет выглядеть рефакторинг с использованием метода "Встраивание метода":

```java

class PizzaDelivery {

// ...

int getRating() {

return numberOfLateDeliveries() > 5 ? 2 : 1;

}

int numberOfLateDeliveries() {

// возвращаем количество задержанных доставок

}

}

```

В результате рефакторинга, метод `moreThanFiveLateDeliveries()` был удален, а его содержимое было встроено непосредственно в метод `getRating()`. Теперь метод `getRating()` напрямую сравнивает количество задержанных доставок `numberOfLateDeliveries()` с числом 5 и возвращает соответствующий рейтинг.

Встраивание метода может быть полезным, когда метод выполняет простую операцию, несет небольшую ответственность и вызывается только в одном месте. Это может улучшить читаемость кода и упростить его структуру. Однако, если метод используется в нескольких местах или выполняет сложную логику, то встраивание метода может быть нецелесообразным.

# ЭКЗАМЕНАЦИОННЫЙ БИЛЕТ №5

Вопрос №1. В чём преимущество шаблонов проектирования

Шаблоны проектирования представляют собой повторно используемые архитектурные решения, которые помогают разработчикам создавать гибкие, расширяемые и поддерживаемые программные системы. Вот некоторые преимущества использования шаблонов проектирования:

1. **Повышение повторного использования кода**: Шаблоны проектирования предоставляют готовые абстракции и структуры, которые можно использовать для решения типичных проблем в различных контекстах. Это позволяет избежать повторного написания кода и способствует повторному использованию уже проверенных и оптимизированных решений.
2. **Улучшение расширяемости и поддерживаемости**: Шаблоны проектирования способствуют созданию гибких и расширяемых систем. Они помогают выделить изменяющиеся части программы от стабильных, что упрощает добавление новых функций и модификацию существующего кода без необходимости внесения глобальных изменений.
3. **Обеспечение согласованности и стандартизации:** Шаблоны проектирования предлагают общие языковые конструкции и подходы, которые способствуют согласованности и стандартизации в проекте. Это делает код более понятным и предсказуемым для других разработчиков, участвующих в проекте.
4. **Улучшение тестируемости:** Шаблоны проектирования способствуют созданию кода, который легко тестируется и подвергается автоматическому тестированию. Они помогают выделить отдельные компоненты системы, что упрощает создание модульных тестов и улучшает общую тестируемость проекта.
5. **Повышение производительности разработки**: Использование шаблонов проектирования позволяет разработчикам сосредоточиться на решении конкретных проблем, предлагая уже установленные решения для типичных задач. Это помогает сократить время разработки и улучшить производительность команды разработчиков.

Вопрос №2. Расскажите про любой порождающий шаблон

**Фабричный метод** (Factory Method) - это шаблон проектирования, который предоставляет интерфейс для создания объектов, но позволяет подклассам выбирать классы, которые должны быть инстанциированы. То есть, он делегирует ответственность за создание объектов подклассам, вместо того чтобы создавать их непосредственно в суперклассе.

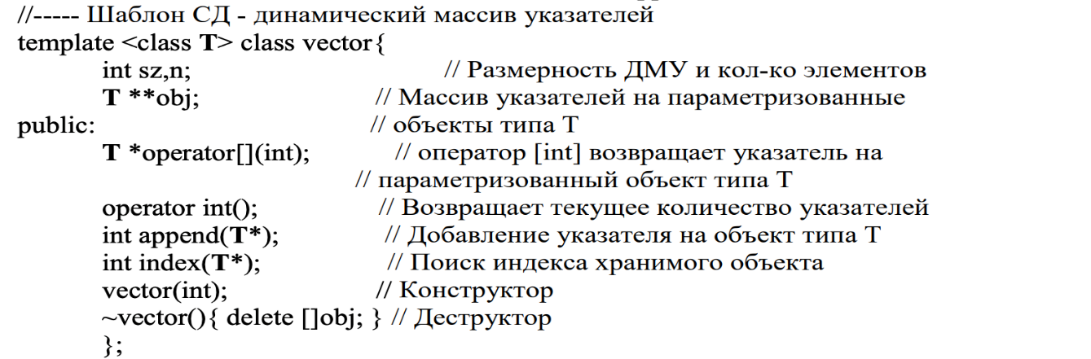
**Преимущества использования фабричного метода включают:**

1. Гибкость и расширяемость: Фабричный метод позволяет добавлять новые типы продуктов, расширяя существующую иерархию классов, без изменения кода суперкласса. Это позволяет легко добавлять новые продукты и вариации, не нарушая принцип открытости/закрытости.
2. Сокрытие деталей создания объекта: Фабричный метод скрывает детали создания объектов внутри подклассов, что делает код более чистым и упрощает уровень абстракции. Клиентский код работает с абстрактным интерфейсом фабрики и продуктов, не зная о конкретных классах, что обеспечивает слабую связность.
3. Возможность применения полиморфизма: Фабричный метод позволяет использовать полиморфизм при работе с продуктами. Клиентский код может оперировать абстрактными типами, а конкретная реализация будет выбрана на основе конкретной фабрики.

Примером применения фабричного метода может быть создание различных типов логгеров в приложении. Суперкласс фабрики определяет метод для создания логгера, а подклассы реализуют этот метод для создания конкретных типов логгеров, таких как файловый логгер, база данных логгер и т.д. Клиентский код вызывает фабричный метод для создания нужного типа логгера, но сам не зависит от конкретных классов логгеров.

**Практическое задание: ………………**

Реализуйте метод «int append(T\*)» из шаблона класса vector



#include <iostream>

#include <memory>

template<typename T>

class vector {

private:

T\* data;

size\_t capacity;

size\_t size;

public:

vector() : data(nullptr), capacity(0), size(0) {}

~vector() {

delete[] data;

}

int append(T\* item) {

if (size >= capacity) {

// Увеличиваем емкость в два раза при необходимости

size\_t newCapacity = (capacity == 0) ? 1 : capacity \* 2;

T\* newData = new T[newCapacity];

// Копируем существующие элементы в новую область памяти

for (size\_t i = 0; i < size; i++) {

newData[i] = data[i];

}

// Освобождаем старую память

delete[] data;

// Перенаправляем указатель на новую область памяти

data = newData;

capacity = newCapacity;

}

// Добавляем новый элемент в конец вектора

data[size++] = \*item;

return size - 1; // Возвращаем индекс добавленного элемента

}

};

int main() {

vector<int> vec;

int item1 = 10;

int item2 = 20;

int index1 = vec.append(&item1);

int index2 = vec.append(&item2);

std::cout << "Index of item1: " << index1 << std::endl;

std::cout << "Index of item2: " << index2 << std::endl;

return 0;

}

В данном примере реализован шаблон класса vector, который содержит метод append(). Метод append() принимает указатель T\* на элемент типа T и добавляет его в конец вектора. Если размер вектора size достигает его текущей емкости capacity, то вектор увеличивает свою емкость в два раза. После этого, существующие элементы копируются в новую область памяти большей емкости, а старая память освобождается. Затем новый элемент добавляется в конец вектора, и его индекс возвращается в качестве результата.

В функции main() создается экземпляр класса vector<int> и добавляются два элемента типа int. Затем выводятся индексы добавленных элементов. Обратите внимание, что данный пример использует динамическое выделение памяти с помощью оператора new[]. Для обеспечения безопасности и управления памятью рекомендуется использовать современные контейнеры и умные указатели, такие как std::vector и std::shared\_ptr из стандартной библиотеки C++.

# ЭКЗАМЕНАЦИОННЫЙ БИЛЕТ №6

Вопрос №1. Зачем перегружать методы

**Перегрузка методов -** это возможность определить несколько методов с одинаковым именем, но с разными наборами параметров в классе или объекте. Вот несколько причин, почему перегрузка методов может быть полезной:

* 1. **Поддержка различных типов данных:** Перегрузка методов позволяет обрабатывать различные типы параметров в зависимости от контекста. Например, у вас может быть метод calculate для класса Math, который может принимать как целые числа, так и числа с плавающей запятой. Перегрузка метода позволяет вам иметь две версии метода calculate, одну для целых чисел и другую для чисел с плавающей запятой.
  2. **Удобство использования:** Перегрузка методов может сделать код более читаемым и естественным для разработчиков, которые используют ваш класс или объект. Они могут вызывать метод с различными наборами параметров, не беспокоясь о деталях реализации.
  3. **Повышение гибкости и расширяемости**: Перегрузка методов может обеспечить гибкость в изменении поведения метода в зависимости от контекста. Вы можете добавить новые версии метода с дополнительными параметрами или изменить типы параметров, не нарушая существующий код, который вызывает метод.
  4. **Улучшение читаемости кода**: Перегрузка методов может помочь устранить необходимость использования разных имен методов для реализации похожей функциональности. Вместо этого вы можете использовать одно имя метода, но с различными параметрами, что делает код более понятным и легким для чтения.

Вопрос №2. Что такое атомарность

**Атомарность** - это свойство операции или процесса, которое означает, что он выполняется как единое неделимое действие, и во время его выполнения не может произойти промежуточных изменений или вмешательства со стороны других операций. Операция считается атомарной, если она либо полностью выполняется, либо не выполняется вовсе, без возможности нахождения в промежуточном состоянии.

Атомарность является важным концептом в многопоточном программировании и параллельных вычислениях. Когда несколько потоков или процессов одновременно обращаются к общим данным или ресурсам, атомарные операции помогают предотвратить состояние гонки и другие проблемы согласованности данных.

**Практическое задание: ………………**

Сделайте рефакторинг данной функции методом «извлечения переменной»

**void** **renderBanner**() {

**if** ((platform.toUpperCase().indexOf("MAC") > -1) &&

(browser.toUpperCase().indexOf("IE") > -1) &&

wasInitialized() && resize > 0 )

{

// do something

}

}

Метод "Извлечение переменной" (Extract Variable) позволяет выделить сложное или повторяющееся выражение в отдельную переменную, чтобы улучшить читаемость и поддерживаемость кода. В данном случае, мы можем использовать этот метод для извлечения сложного условия из функции `renderBanner()`. Вот как будет выглядеть рефакторинг с использованием метода "Извлечение переменной":

```java

void renderBanner() {

boolean isMacOS = platform.toUpperCase().indexOf("MAC") > -1;

boolean isIEBrowser = browser.toUpperCase().indexOf("IE") > -1;

boolean isInitialized = wasInitialized();

boolean isResizable = resize > 0;

if (isMacOS && isIEBrowser && isInitialized && isResizable) {

// do something

}

}

```

В результате рефакторинга, сложное условие было извлечено в четыре отдельные переменные: `isMacOS`, `isIEBrowser`, `isInitialized` и `isResizable`. Каждая переменная отражает отдельную часть условия и имеет информативное имя.

Использование извлеченных переменных упрощает чтение и понимание условия в функции `renderBanner()`. Кроме того, это позволяет избежать повторения сложного выражения и упрощает внесение изменений в логику условия.

Метод "Извлечение переменной" особенно полезен, когда сложное выражение используется несколько раз в коде или когда оно затрудняет чтение и понимание функции.

# ЭКЗАМЕНАЦИОННЫЙ БИЛЕТ №7

Вопрос №1. Какие основные виды паттернов проектирования вы знаете? Зачем необходимы паттерны проектирования?

Паттерны проектирования представляют собой повторно используемые и документированные решения для типичных проблем, возникающих при проектировании программного обеспечения. Они описывают структурированные подходы к решению конкретных задач и предлагают изученные шаблоны, которые можно применять в различных контекстах разработки.

* 1. **Порождающие паттерны.** Эти паттерны предоставляют механизмы для создания объектов с учетом особенностей конкретной ситуации. Некоторые примеры включают Singleton (Одиночка), Factory Method (Фабричный метод), Abstract Factory (Абстрактная фабрика) и Builder (Строитель).
  2. **Структурные паттерны**. Эти паттерны касаются композиции объектов и классов и предоставляют способы построения более гибких структур. Некоторые примеры включают Adapter (Адаптер), Decorator (Декоратор), Proxy (Заместитель) и Composite (Компоновщик).
  3. **Поведенческие паттерны**: Эти паттерны определяют взаимодействие между объектами и ответственность за их поведение. Они сосредоточены на коммуникации между объектами. Некоторые примеры включают Observer (Наблюдатель), Strategy (Стратегия), Command (Команда) и Iterator (Итератор).
  4. **Архитектурные паттерны**: Эти паттерны описывают высокоуровневую организацию системы и взаимодействие между ее компонентами. Примеры включают Model-View-Controller (MVC), Layered Architecture (Слоистая архитектура) и Microservices (Микросервисы).

**Паттерны проектирования необходимы по нескольким причинам:**

* 1. **Повышение повторного использования кода**: Паттерны проектирования предлагают конкретные решения для типичных проблем. Это способствует снижению дублирования кода и улучшению эффективности разработки.
  2. **Улучшение расширяемости и гибкости**: Паттерны проектирования позволяют легко изменять и модифицировать код, не нарушая существующую функциональность.
  3. **Улучшение понимания и коммуникации**: Паттерны проектирования служат средством коммуникации между разработчиками и обеспечивают более эффективное взаимодействие в команде.

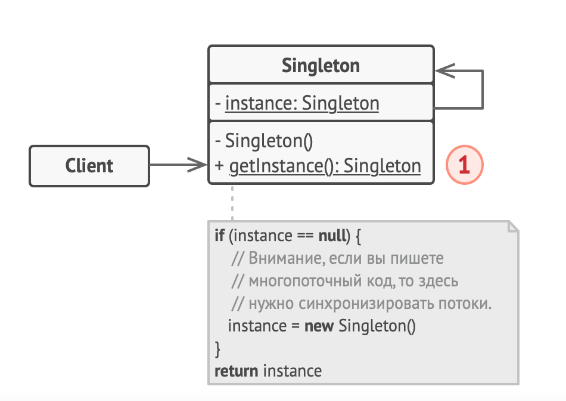
Вопрос №2. Что такое обработчик события?

**Обработчик события** - это функция или метод, который выполняется в ответ на возникновение определенного события. События могут быть инициированы пользователем (например, клик мышью, нажатие клавиши) или системой (например, завершение операции, получение данных).

Обработчики событий используются в программировании для реагирования на события и выполнения соответствующих действий. Когда событие происходит, вызывается соответствующий обработчик события, который содержит код, выполняющий нужные операции или реагирующий на событие.

**Практическое задание: ………………**

Реализуйте шаблон проектирования Singleton “Одиночка” на любом удобном языке (С++/Java/C#).



public class Singleton {

private static Singleton instance;

// Приватный конструктор, чтобы предотвратить создание экземпляров класса извне

private Singleton() {

}

// Метод для получения экземпляра класса

public static Singleton getInstance() {

if (instance == null) {

// Создаем экземпляр только при первом вызове

instance = new Singleton();

}

return instance;

}

// Другие методы класса

public void someMethod() {

// Реализация метода

}

}

В данном примере класс Singleton имеет приватное статическое поле instance, которое хранит единственный экземпляр класса. Конструктор класса Singleton объявлен как приватный, что предотвращает создание экземпляров класса извне.

Метод getInstance() проверяет, существует ли уже экземпляр класса. Если экземпляр еще не создан, то он создается при первом вызове метода getInstance(), и затем возвращается. При последующих вызовах метода getInstance() уже созданный экземпляр класса возвращается непосредственно.

Теперь, чтобы получить экземпляр класса Singleton, можно вызвать метод getInstance() следующим образом:

java

Copy

Singleton singleton = Singleton.getInstance();

Таким образом, мы можем удостовериться, что в программе существует только один экземпляр класса Singleton.

Обратите внимание, что данная реализация шаблона Singleton не является потокобезопасной. Если вы использовать эту реализацию в многопоточной среде, возможны проблемы с созданием нескольких экземпляров класса. Для обеспечения потокобезопасности рекомендуется использовать двойную проверку на null или синхронизированный блок.

# ЭКЗАМЕНАЦИОННЫЙ БИЛЕТ №8

Вопрос №1. Что такое событийно-ориентированное программирование?

**Событийно-ориентированное программирование** - это подход к разработке программного обеспечения, в котором основное внимание уделяется обработке и генерации событий. В событийно-ориентированной парадигме программа состоит из компонентов, которые реагируют на возникающие события и взаимодействуют друг с другом путем передачи сообщений. Событийно-ориентированное программирование широко используется в различных областях, включая разработку пользовательских интерфейсов, асинхронное программирование, обработку сетевых запросов и т.д.

Вопрос №2. Расскажите про любой поведенческий паттерн проектирования.

**Паттерн Observer** используется для реализации механизма подписки и оповещения, где объекты, называемые наблюдателями, автоматически получают обновления от других объектов, называемых субъектами, при изменении их состояния. Наблюдатели могут быть добавлены и удалены динамически, а субъекты могут оповещать наблюдателей о произошедших изменениях.

**Основные компоненты паттерна Observer:**

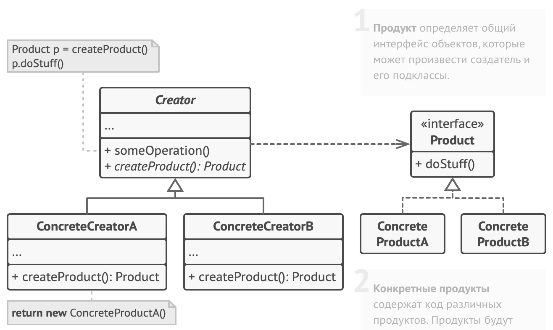
* Интерфейс Subject: Определяет методы для добавления, удаления и оповещения наблюдателей.
* Класс ConcreteSubject: Реализует интерфейс Subject и содержит состояние, которое может изменяться. Он также отправляет оповещения наблюдателям при изменении состояния.
* Интерфейс Observer: Определяет метод, который субъект использует для оповещения наблюдателя.
* Класс ConcreteObserver: Реализует интерфейс Observer и содержит логику, которая будет выполнена в ответ на оповещение от субъекта.

Примером использования паттерна Observer может быть система уведомлений, где пользователи могут подписываться на различные категории уведомлений (субъекты), а при появлении нового уведомления, все подписчики (наблюдатели) получают соответствующее оповещение.

Паттерн Observer помогает достичь слабой связанности между субъектами и наблюдателями, обеспечивая гибкость и расширяемость системы.

**Практическое задание: ………………**

Реализуйте шаблон проектирования Фабричный метод “Factory method” на любом удобном языке (С++/Java/C#).



public abstract class Coffee {

public void grindCoffee() {

// помол кофе

}

public void makeCoffee() {

// делаем кофе

}

public void pourIntoCup() {

// наливаем в чашку

}

}

public enum CoffeeType {

AMERICANO,

ESPRESSO,

CAPPUCCINO,

CAFFE\_LATTE

}

public interface CoffeeFactory {

Coffee createCoffee(CoffeeType type);

}

public class ItalianStyleAmericano extends Coffee {

// реализация для итальянского стиля американо

}

public class ItalianStyleCappuccino extends Coffee {

// реализация для итальянского стиля капуччино

}

public class ItalianStyleCaffeLatte extends Coffee {

// реализация для итальянского стиля кафе латте

}

public class ItalianStyleEspresso extends Coffee {

// реализация для итальянского стиля эспрессо

}

public class AmericanStyleAmericano extends Coffee {

// реализация для американского стиля американо

}

public class AmericanStyleCappuccino extends Coffee {

// реализация для американского стиля капуччино

}

public class AmericanStyleCaffeLatte extends Coffee {

// реализация для американского стиля кафе латте

}

public class AmericanStyleEspresso extends Coffee {

// реализация для американского стиля эспрессо

}

public class SimpleItalianCoffeeFactory implements CoffeeFactory {

@Override

public Coffee createCoffee(CoffeeType type) {

switch (type) {

case AMERICANO:

return new ItalianStyleAmericano();

case ESPRESSO:

return new ItalianStyleEspresso();

case CAPPUCCINO:

return new ItalianStyleCappuccino();

case CAFFE\_LATTE:

return new ItalianStyleCaffeLatte();

default:

throw new IllegalArgumentException("Invalid coffee type: " + type);

}

}

}

public class SimpleAmericanCoffeeFactory implements CoffeeFactory {

@Override

public Coffee createCoffee(CoffeeType type) {

switch (type) {

case AMERICANO:

return new AmericanStyleAmericano();

case ESPRESSO:

return new AmericanStyleEspresso();

case CAPPUCCINO:

return new AmericanStyleCappuccino();

case CAFFE\_LATTE:

return new AmericanStyleCaffeLatte();

default:

throw new IllegalArgumentException("Invalid coffee type: " + type);

}

}

}

public class CoffeeShop {

private final CoffeeFactory coffeeFactory;

public CoffeeShop(CoffeeFactory coffeeFactory) {

this.coffeeFactory = coffeeFactory;

}

public Coffee orderCoffee(CoffeeType type) {

Coffee coffee = coffeeFactory.createCoffee(type);

coffee.grindCoffee();

coffee.makeCoffee();

coffee.pourIntoCup();

System.out.println("Вот ваш кофе! Спасибо, приходите еще!");

return coffee;

}

}

public class Main {

public static void main(String[] args) {

CoffeeFactory italianCoffeeFactory = new SimpleItalianCoffeeFactory();

CoffeeShop italianCoffeeShop = new CoffeeShop(italianCoffeeFactory);

italianCoffeeShop.orderCoffee(CoffeeType.CAPPUCCINO);

CoffeeFactory americanCoffeeFactory = new SimpleAmericanCoffeeFactory();

CoffeeShop americanCoffeeShop = new CoffeeShop(americanCoffeeFactory);

americanCoffeeShop.orderCoffee(CoffeeType.CAPPUCCINO);

}

}

# ЭКЗАМЕНАЦИОННЫЙ БИЛЕТ №9

Вопрос №1. Какие элементы управления вы знаете?

**Некоторые из** наиболее распространенных элементов управления включают:

* 1. **Кнопка** (Button): Элемент, на который пользователь может нажать для выполнения определенного действия.
  2. **Поле ввода** (Input Field): Позволяет пользователю вводить текст или другие данные.
  3. **Выпадающий список** (Dropdown List): Предоставляет пользователю список вариантов, из которых можно выбрать один.
  4. **Флажок** (Checkbox): Позволяет пользователю выбрать или снять выбор с определенного параметра.
  5. **Переключатель** (Radio Button): Позволяет пользователю выбрать один вариант из предоставленного списка.
  6. **Список** (List): Предоставляет пользователю список элементов для выбора или просмотра.
  7. **Ползунок** (Slider): Позволяет пользователю выбирать значение из некоторого диапазона, перемещая ползунок по шкале
  8. **Таблица** (Table): Отображает данные в виде сетки, обычно с рядами и столбцами, которые можно прокручивать.
  9. **Вкладки** (Tabs): Позволяют пользователю переключаться между различными разделами или видами содержимого.
  10. **Меню** (Menu): Предоставляет пользователю список команд или опций для выбора.

Вопрос №2. Что такое обработчик события?

**Обработчик события** (Event Handler) - это функция, метод или блок кода, который выполняется в ответ на возникновение определенного события.

Обработчик события связывается с определенным элементом управления или субъектом, который генерирует событие. Когда происходит событие, обработчик события вызывается автоматически, и его код выполняется.

Например, при нажатии на кнопку (Button) в пользовательском интерфейсе, обработчик события кнопки будет вызван, и можно выполнить определенные действия, например, обновление данных, переход на другую страницу или выполнение каких-либо вычислений.

**Практическое задание: ………………**

Сделайте рефакторинг данной функции методом «извлечения переменной»

**void** **renderBanner**() {

**if** ((platform.toUpperCase().indexOf("MAC") > -1) &&

(browser.toUpperCase().indexOf("IE") > -1) &&

wasInitialized() && resize > 0 )

{

// do something

}

}

Метод "Извлечение переменной" (Extract Variable) позволяет выделить сложное или повторяющееся выражение в отдельную переменную, чтобы улучшить читаемость и поддерживаемость кода. В данном случае, мы можем использовать этот метод для извлечения сложного условия из функции `renderBanner()`. Вот как будет выглядеть рефакторинг с использованием метода "Извлечение переменной":

```java

void renderBanner() {

boolean isMacOS = platform.toUpperCase().indexOf("MAC") > -1;

boolean isIEBrowser = browser.toUpperCase().indexOf("IE") > -1;

boolean isInitialized = wasInitialized();

boolean isResizable = resize > 0;

if (isMacOS && isIEBrowser && isInitialized && isResizable) {

// do something

}

}

```

В результате рефакторинга, сложное условие было извлечено в четыре отдельные переменные: `isMacOS`, `isIEBrowser`, `isInitialized` и `isResizable`. Каждая переменная отражает отдельную часть условия и имеет информативное имя.

Использование извлеченных переменных упрощает чтение и понимание условия в функции `renderBanner()`. Кроме того, это позволяет избежать повторения сложного выражения и упрощает внесение изменений в логику условия.

Метод "Извлечение переменной" особенно полезен, когда сложное выражение используется несколько раз в коде или когда оно затрудняет чтение и понимание функции.

# ЭКЗАМЕНАЦИОННЫЙ БИЛЕТ №10

Вопрос №1. Какие основные алгоритмы поиска вы знаете?

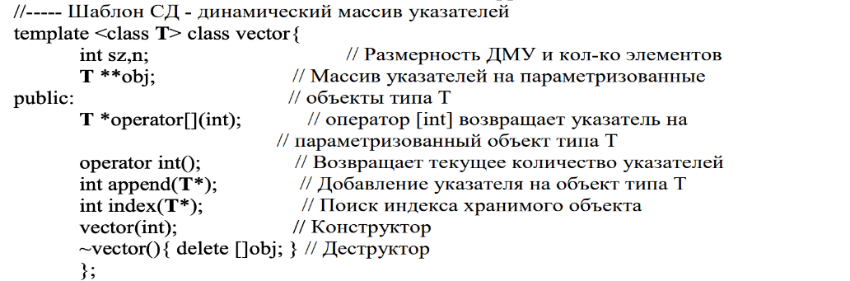
* **Линейный поиск** (Linear Search): Простейший алгоритм поиска, который последовательно проверяет каждый элемент в коллекции до нахождения искомого значения или достижения конца коллекции.
* **Бинарный поиск** (Binary Search): Эффективный алгоритм поиска, который применяется к отсортированным коллекциям. Он сравнивает искомое значение с элементом в середине коллекции и затем сужает область поиска в зависимости от результата сравнения, итеративно повторяя процесс до нахождения искомого значения.
* **Интерполяционный поиск** (Interpolation Search): Этот алгоритм также применяется к отсортированным коллекциям и использует линейную интерполяцию для нахождения примерного расположения искомого значения. Затем область поиска сужается, подобно бинарному поиску, пока искомое значение не будет найдено.
* **Поиск с использованием хеш-таблиц** (Hash-based Search): Этот алгоритм используется для поиска в хеш-таблицах, где значения хранятся в соответствии с их хеш-кодами. Он обычно выполняется за постоянное время, предоставляя эффективный поиск.
* **Алгоритмы поиска в графах**: Например, поиск в глубину (Depth-First Search, DFS) и поиск в ширину (Breadth-First Search, BFS) применяются для нахождения пути или определенного узла в графе.

Вопрос №2. Какие основные алгоритмы сортировки вы знаете?

* **Сортировка пузырьком** (Bubble Sort): Простой алгоритм, который последовательно сравнивает и меняет местами соседние элементы, пока вся коллекция не будет отсортирована.
* **Сортировка выбором** (Selection Sort): Алгоритм, который на каждой итерации находит минимальный элемент в неотсортированной части коллекции и помещает его в начало этой части.
* **Сортировка вставками** (Insertion Sort): Алгоритм, который постепенно строит отсортированную часть коллекции, вставляя каждый элемент на свое место в отсортированной части.
* **Сортировка слиянием** (Merge Sort): Рекурсивный алгоритм, который разделяет коллекцию на две половины, рекурсивно сортирует каждую половину, а затем объединяет их в одну отсортированную коллекцию.
* **Быстрая сортировка** (Quick Sort): Рекурсивный алгоритм, который выбирает опорный элемент, разделяет коллекцию на две части (элементы меньше опорного и элементы больше опорного), и рекурсивно сортирует обе части.

**Практическое задание: ………………**

Реализуйте метод «int append(T\*)» из шаблона класса vector



#include <iostream>

#include <memory>

template<typename T>

class vector {

private:

T\* data;

size\_t capacity;

size\_t size;

public:

vector() : data(nullptr), capacity(0), size(0) {}

~vector() {

delete[] data;

}

int append(T\* item) {

if (size >= capacity) {

// Увеличиваем емкость в два раза при необходимости

size\_t newCapacity = (capacity == 0) ? 1 : capacity \* 2;

T\* newData = new T[newCapacity];

// Копируем существующие элементы в новую область памяти

for (size\_t i = 0; i < size; i++) {

newData[i] = data[i];

}

// Освобождаем старую память

delete[] data;

// Перенаправляем указатель на новую область памяти

data = newData;

capacity = newCapacity;

}

// Добавляем новый элемент в конец вектора

data[size++] = \*item;

return size - 1; // Возвращаем индекс добавленного элемента

}

};

int main() {

vector<int> vec;

int item1 = 10;

int item2 = 20;

int index1 = vec.append(&item1);

int index2 = vec.append(&item2);

std::cout << "Index of item1: " << index1 << std::endl;

std::cout << "Index of item2: " << index2 << std::endl;

return 0;

}

В данном примере реализован шаблон класса vector, который содержит метод append(). Метод append() принимает указатель T\* на элемент типа T и добавляет его в конец вектора. Если размер вектора size достигает его текущей емкости capacity, то вектор увеличивает свою емкость в два раза. После этого, существующие элементы копируются в новую область памяти большей емкости, а старая память освобождается. Затем новый элемент добавляется в конец вектора, и его индекс возвращается в качестве результата.

В функции main() создается экземпляр класса vector<int> и добавляются два элемента типа int. Затем выводятся индексы добавленных элементов. Обратите внимание, что данный пример использует динамическое выделение памяти с помощью оператора new[]. Для обеспечения безопасности и управления памятью рекомендуется использовать современные контейнеры и умные указатели, такие как std::vector и std::shared\_ptr из стандартной библиотеки C++.

# ЭКЗАМЕНАЦИОННЫЙ БИЛЕТ №11

Вопрос №1. Что такое рефакторинг и какие методы вы знаете?

**Рефакторинг** - это процесс изменения внутренней структуры программного кода с целью улучшения его понимаемости, читаемости, модульности и поддерживаемости, при этом без изменения его внешнего поведения. Основная цель рефакторинга заключается в устранении технического долга и повышении качества кода.

Рефакторинг может включать следующие методы:

* **Извлечение метода**. Выделение фрагмента кода в отдельный метод для повышения его читаемости и повторного использования.
* **Переименование переменных и методов**. Изменение имени переменных, методов или классов для более ясного отражения их назначения и улучшения понимаемости кода.
* **Выделение класса.** Группировка связанных переменных и методов в отдельный класс для повышения модульности и улучшения структуры кода.
* **Удаление дублированного кода.** Идентификация и устранение повторяющихся фрагментов кода путем вынесения их в отдельные методы или классы.

Вопрос №2. Как организуется процесс рефакторинга?

1. **Понимание кода**: Важно полностью понять функциональность и структуру существующего кода.
2. **Выявление проблемных мест:** Идентификация участков кода, требующих улучшения. Это может быть связано с повторяющимся кодом, длинными методами, сложными условиями и т.д.
3. **Выбор метода рефакторинга**
4. **Создание тестов**: Написание или обновление тестов, которые проверяют правильность работы кода до и после рефакторинга.
5. **Проведение рефакторинга:** Применение выбранного метода рефакторинга к коду. Это может включать изменение структуры классов, переименование переменных, извлечение методов и т.д.
6. **Проверка работоспособности:** Запуск тестов и проверка, что после рефакторинга код продолжает работать корректно.
7. **Повторение процесса**: Рефакторинг - это итеративный процесс. Повторяйте шаги 2-6 для других участков кода, требующих улучшения.

Важно помнить, что рефакторинг должен проводиться постепенно и контролируемо, чтобы не нарушить работоспособность программы.

**Практическое задание: ………………**

Сделайте рефакторинг следующей функции методом «Встраивания метода»

**class** **PizzaDelivery** {

// ...

**int** getRating() {

**return** moreThanFiveLateDeliveries() ? 2 : 1;

}

**boolean** moreThanFiveLateDeliveries() {

**return** numberOfLateDeliveries > 5;

}

}

Встраивание метода (Inline Method) - это метод рефакторинга, который заменяет вызов метода его фактическим содержимым. В данном случае, мы можем встроить метод `moreThanFiveLateDeliveries()` в метод `getRating()`. Вот как будет выглядеть рефакторинг с использованием метода "Встраивание метода":

```java

class PizzaDelivery {

// ...

int getRating() {

return numberOfLateDeliveries() > 5 ? 2 : 1;

}

int numberOfLateDeliveries() {

// возвращаем количество задержанных доставок

}

}

```

В результате рефакторинга, метод `moreThanFiveLateDeliveries()` был удален, а его содержимое было встроено непосредственно в метод `getRating()`. Теперь метод `getRating()` напрямую сравнивает количество задержанных доставок `numberOfLateDeliveries()` с числом 5 и возвращает соответствующий рейтинг.

Встраивание метода может быть полезным, когда метод выполняет простую операцию, несет небольшую ответственность и вызывается только в одном месте. Это может улучшить читаемость кода и упростить его структуру. Однако, если метод используется в нескольких местах или выполняет сложную логику, то встраивание метода может быть нецелесообразным.

# ЭКЗАМЕНАЦИОННЫЙ БИЛЕТ №12

Вопрос №1. Как можно провести рефакторинг кода на уровне переменных?

* **Переименование переменных:** Изменение имени переменной на более понятное и описательное. Хорошие имена переменных помогают лучше понять их назначение и улучшают читаемость кода.
* **Разделение переменных**: Если переменная выполняет сразу несколько функций или хранит разные типы данных, ее можно разделить на несколько отдельных переменных. Это поможет улучшить понимаемость кода и сделать его более явным.
* **Устранение неиспользуемых переменных**: Идентификация и удаление переменных, которые больше не используются в коде. Неиспользуемые переменные только затрудняют чтение и понимание кода.
* **Локализация области видимости переменных:** Если переменная используется только внутри определенного блока кода, ее объявление и инициализация могут быть перемещены в этот блок. Это поможет уменьшить область видимости переменной и сделать код более модульным.
* **Перемещение переменных ближе к использованию:** Если переменная объявляется в начале функции, но используется только в конкретном участке кода, ее объявление можно переместить ближе к месту использования. Это делает код более легким для чтения и следует принципу минимизации области видимости переменных.

Вопрос №2. Как можно провести рефакторинг алгоритма на уровне функций?

* **Разделение длинных функций:** Если функция слишком длинная и выполняет несколько задач, ее можно разделить на несколько более маленьких функций, каждая из которых будет отвечать только за одну задачу.
* **Извлечение повторяющегося кода в функцию:** Если у вас есть повторяющийся фрагмент кода в нескольких местах, его можно извлечь в отдельную функцию и вызывать ее из нужных мест.
* **Упрощение сложных условий**: Если у вас есть функция с длинными и сложными условиями, их можно упростить путем выделения отдельных проверок в отдельные функции или использования более ясных логических операций.
* **Переименование функций**: Если название функции не отражает ее назначение или не соответствует стандартам именования, ее можно переименовать для улучшения понимаемости и согласованности с остальным кодом.
* **Удаление неиспользуемых функций**: Идентификация и удаление функций, которые больше не используются в коде. Неиспользуемые функции только ухудшают читаемость и поддерживаемость кода.
* **Перемещение функций в более подходящие места**: Если функция логически относится к другому модулю или классу, ее можно переместить туда для улучшения организации кода и соблюдения принциципа единственной ответственности.

**Практическое задание: ………………**

Сделайте рефакторинг данной функции методом «извлечения переменной»

**void** **renderBanner**() {

**if** ((platform.toUpperCase().indexOf("MAC") > -1) &&

(browser.toUpperCase().indexOf("IE") > -1) &&

wasInitialized() && resize > 0 )

{

// do something

}

}

Метод "Извлечение переменной" (Extract Variable) позволяет выделить сложное или повторяющееся выражение в отдельную переменную, чтобы улучшить читаемость и поддерживаемость кода. В данном случае, мы можем использовать этот метод для извлечения сложного условия из функции `renderBanner()`. Вот как будет выглядеть рефакторинг с использованием метода "Извлечение переменной":

```java

void renderBanner() {

boolean isMacOS = platform.toUpperCase().indexOf("MAC") > -1;

boolean isIEBrowser = browser.toUpperCase().indexOf("IE") > -1;

boolean isInitialized = wasInitialized();

boolean isResizable = resize > 0;

if (isMacOS && isIEBrowser && isInitialized && isResizable) {

// do something

}

}

```

В результате рефакторинга, сложное условие было извлечено в четыре отдельные переменные: `isMacOS`, `isIEBrowser`, `isInitialized` и `isResizable`. Каждая переменная отражает отдельную часть условия и имеет информативное имя.

Использование извлеченных переменных упрощает чтение и понимание условия в функции `renderBanner()`. Кроме того, это позволяет избежать повторения сложного выражения и упрощает внесение изменений в логику условия.

Метод "Извлечение переменной" особенно полезен, когда сложное выражение используется несколько раз в коде или когда оно затрудняет чтение и понимание функции.

# ЭКЗАМЕНАЦИОННЫЙ БИЛЕТ №13

Вопрос №1. Что такое декомпозиция кода и как она может помочь в оптимизации?

**Декомпозиция кода** - это процесс разбиения сложной задачи или функциональности на более мелкие и понятные единицы, такие как функции, методы, классы или модули. Целью декомпозиции является упрощение понимания и управления кодом, а также повышение его модульности и переиспользуемости.

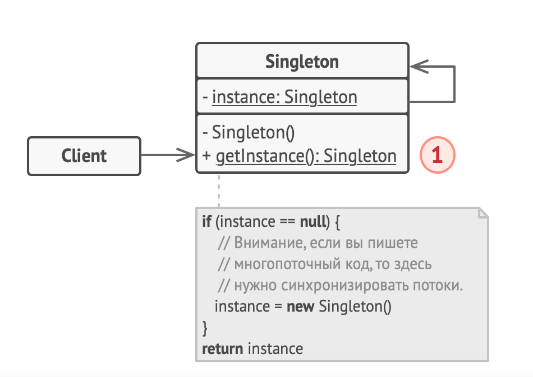
1. Упрощение понимания кода:
2. Помогает выделить логически связанные части функциональности в отдельные модули или классы – улучшении организации кода, его сопровождения.
3. При внесении изменений в функциональность, позволяет изменить только в соответствующих модулях или классах.
4. Позволяет разработчикам работать над различными модулями или классами независимо друг от друга. Это повышает эффективность и скорость разработки проекта.

Вопрос №2. Какие основные принципы следует учитывать при оптимизации программного кода?

1. **Избегайте избыточной сложности:** Сложный и запутанный код усложняет его понимание и сопровождение. Старайтесь упростить логику и структуру кода, избегая излишней вложенности, сложных условий и дублирования кода.
2. **Оптимизируйте производительность**: Идентифицируйте узкие места в коде, которые замедляют его выполнение, и производите соответствующие оптимизации. Это может включать улучшение алгоритмов, использование более эффективных структур данных и алгоритмов, а также оптимизацию работы с памятью.
3. **Обратите внимание на использование ресурсов**: При оптимизации кода необходимо учитывать использование ресурсов, таких как память, сетевые соединения или файловые операции. Обращайте внимание на потенциальные утечки памяти, неэффективное использование ресурсов и возможности их оптимизации.
4. **Пишите читаемый и поддерживаемый код:** Читаемый код является основой для его оптимизации и поддержки в долгосрочной перспективе. Используйте понятные и описательные имена переменных, методов и классов. Добавляйте комментарии и документацию, чтобы облегчить понимание кода другим разработчикам.
5. **Тестируйте изменения**: При проведении оптимизаций необходимо обеспечить тестирование изменений, чтобы убедиться, что оптимизированный код по-прежнему работает правильно и не вводит новые ошибки.

**Практическое задание: ………………**

Реализуйте шаблон проектирования Singleton “Одиночка” на любом удобном языке (С++/Java/C#).



public class Singleton {

private static Singleton instance;

// Приватный конструктор, чтобы предотвратить создание экземпляров класса извне

private Singleton() {

}

// Метод для получения экземпляра класса

public static Singleton getInstance() {

if (instance == null) {

// Создаем экземпляр только при первом вызове

instance = new Singleton();

}

return instance;

}

// Другие методы класса

public void someMethod() {

// Реализация метода

}

}

В данном примере класс Singleton имеет приватное статическое поле instance, которое хранит единственный экземпляр класса. Конструктор класса Singleton объявлен как приватный, что предотвращает создание экземпляров класса извне.

Метод getInstance() проверяет, существует ли уже экземпляр класса. Если экземпляр еще не создан, то он создается при первом вызове метода getInstance(), и затем возвращается. При последующих вызовах метода getInstance() уже созданный экземпляр класса возвращается непосредственно.

Теперь, чтобы получить экземпляр класса Singleton, можно вызвать метод getInstance() следующим образом:

java

Copy

Singleton singleton = Singleton.getInstance();

Таким образом, мы можем удостовериться, что в программе существует только один экземпляр класса Singleton.

Обратите внимание, что данная реализация шаблона Singleton не является потокобезопасной. Если вы использовать эту реализацию в многопоточной среде, возможны проблемы с созданием нескольких экземпляров класса. Для обеспечения потокобезопасности рекомендуется использовать двойную проверку на null или синхронизированный блок.

# ЭКЗАМЕНАЦИОННЫЙ БИЛЕТ №14

Вопрос №1. Какие преимущества имеют веб-сервисы перед традиционными клиент-серверными архитектурами?

1. **Платформенная независимость**: Веб-сервисы основаны на открытых стандартах, таких как HTTP, XML и JSON. Это позволяет веб-сервисам быть платформенно независимыми, что означает, что они могут быть использованы на различных операционных системах и устройствах.
2. **Легкая интеграция:** Веб-сервисы обеспечивают стандартизированный способ обмена данными между системами. Они используют открытые протоколы и форматы данных, что делает интеграцию с другими приложениями и системами более простой. Это особенно полезно в ситуациях, когда необходимо интегрировать разнородные системы.
3. **Расширяемость**: Веб-сервисы могут быть легко расширены путем добавления новых функций через добавление новых методов или ресурсов. Это позволяет системам развиваться и адаптироваться к новым требованиям без необходимости изменения клиентского кода.
4. **Масштабируемость**: Веб-сервисы могут быть развернуты на нескольких серверах или облаках, что обеспечивает горизонтальную масштабируемость. Это позволяет обрабатывать большие объемы запросов и распределить нагрузку между несколькими серверами.
5. **Легкая доступность:** Веб-сервисы доступны через сеть, что позволяет клиентам обращаться к ним из любого места, где есть подключение к сети. Это особенно полезно для распределенных команд, удаленных работников или приложений, работающих на разных устройствах.
6. **Универсальность:** Веб-сервисы могут быть использованы различными клиентскими приложениями и устройствами, включая веб-браузеры, мобильные приложения и даже другие сервисы. Это делает веб-сервисы универсальным и гибким решением для обмена данными и интеграции различных систем.

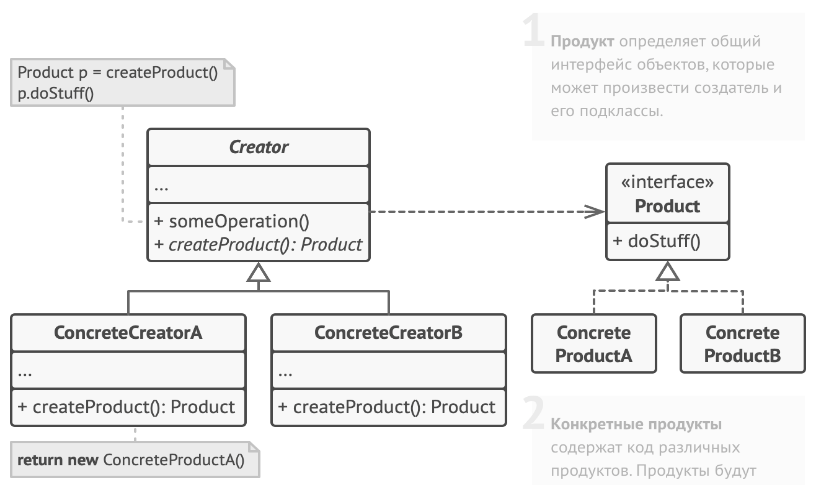
Вопрос №2. Каким образом можно добавлять элементы управления при создании графического интерфейса?

* **Ручное добавление элементов управления**: В большинстве современных графических интерфейсных фреймворков вы можете создавать элементы управления программно, используя API фреймворка. Например, веб-фреймворки, такие как React или Angular, предоставляют возможность создавать элементы управления с помощью компонентов и добавлять их на страницу или в приложение.
* **Использование визуальных редакторов:** Многие интегрированные среды разработки (IDE) и визуальные редакторы предлагают графический интерфейс для добавления элементов управления. Вы можете перетаскивать и размещать элементы управления на форме или странице, а затем настраивать их свойства и поведение.
* **Использование разметки или декларативного подхода:** Некоторые фреймворки и языки разработки графического интерфейса позволяют добавлять элементы управления с использованием разметки или декларативного подхода. Например, HTML и XML позволяют определить структуру и свойства элементов управления с помощью тегов и атрибутов. Это упрощает процесс добавления элементов управления и делает код более читаемым и легким для поддержки.
* **Использование библиотек и компонентов**: Многие фреймворки и библиотеки предлагают готовые элементы управления, которые можно легко добавить в графический интерфейс. Это может быть в виде предопределенных компонентов или плагинов, которые предоставляют готовую функциональность и стилизацию элементов управления.

В зависимости от выбранной платформы и инструментов разработки, доступны различные подходы и методы для добавления элементов управления в графический интерфейс.

**Практическое задание: ………………**

Реализуйте шаблон проектирования Фабричный метод “Factory method” на любом удобном языке (С++/Java/C#).



```java

// Абстрактный класс, представляющий продукт

abstract class Product {

abstract void operation();

}

// Конкретный класс продукта A

class ConcreteProductA extends Product {

void operation() {

System.out.println("ConcreteProductA operation");

}

}

// Конкретный класс продукта B

class ConcreteProductB extends Product {

void operation() {

System.out.println("ConcreteProductB operation");

}

}

// Абстрактный класс, представляющий фабрику

abstract class Factory {

abstract Product createProduct();

}

// Конкретная фабрика для создания продукта A

class ConcreteFactoryA extends Factory {

Product createProduct() {

return new ConcreteProductA();

}

}

// Конкретная фабрика для создания продукта B

class ConcreteFactoryB extends Factory {

Product createProduct() {

return new ConcreteProductB();

}

}

public class Main {

public static void main(String[] args) {

Factory factoryA = new ConcreteFactoryA();

Product productA = factoryA.createProduct();

productA.operation();

Factory factoryB = new ConcreteFactoryB();

Product productB = factoryB.createProduct();

productB.operation();

}

}

```

В данном примере реализован шаблон проектирования Фабричный метод. Он состоит из абстрактного класса `Product`, представляющего абстрактный продукт, и его конкретных реализаций `ConcreteProductA` и `ConcreteProductB`, представляющих конкретные продукты A и B соответственно. Класс `Product` имеет абстрактный метод `operation()`, который должен быть реализован в каждой конкретной реализации продукта.

Также в шаблоне присутствует абстрактный класс `Factory`, представляющий абстрактную фабрику, и его конкретные реализации `ConcreteFactoryA` и `ConcreteFactoryB`, представляющие конкретные фабрики для создания продуктов A и B соответственно. Класс `Factory` имеет абстрактный метод `createProduct()`, который должен быть реализован в каждой конкретной реализации фабрики и возвращает объект типа `Product`.

В методе `main()` создаются экземпляры фабрик `ConcreteFactoryA` и `ConcreteFactoryB`, затем с их помощью создаются соответствующие продукты `ConcreteProductA` и `ConcreteProductB`. Вызывается метод `operation()` на каждом продукте для выполнения специфических операций.

# ЭКЗАМЕНАЦИОННЫЙ БИЛЕТ №15

Вопрос №1. Какие методы программирования применяются при разработке консольных приложений?

* **Процедурное программирование**: Это метод программирования, основанный на организации программы в функции или процедуры, которые выполняют определенные задачи. При разработке консольных приложений можно использовать процедурное программирование для структурирования и организации кода.
* **Объектно-ориентированное программирование (ООП):** ООП является парадигмой программирования, которая основана на концепции объектов, которые являются экземплярами классов. При разработке консольных приложений можно применять ООП для создания классов и объектов, которые представляют различные компоненты приложения и выполняют определенные функции.
* **Функциональное программирование:** Функциональное программирование фокусируется на использовании функций как основной единицы разработки. В консольных приложениях можно использовать функциональное программирование для создания и комбинирования функций, которые обрабатывают данные и выполняют операции.
* **Структурное программирование**: Структурное программирование предлагает организацию программы с использованием последовательности, условных операторов и циклов. В консольных приложениях можно использовать структурное программирование для управления потоком выполнения и обработки данных.
* **Модульное программирование**: Модульное программирование предполагает разделение программы на независимые модули, которые выполняют определенные функции. В консольных приложениях можно использовать модульное программирование для организации кода в отдельные модули, что облегчает его понимание и поддержку.

Вопрос №2. Какие методы программирования применяются при разработке веб-приложений?

* **Клиент-серверная архитектура**: Веб-приложения основаны на клиент-серверной архитектуре, где клиентский браузер взаимодействует с сервером для получения данных и выполнения операций. При разработке веб-приложений необходимо учитывать эту архитектуру и использовать соответствующие методы программирования для клиентской и серверной стороны.
* **Web-фреймворки**: Web-фреймворки предоставляют набор инструментов и библиотек для разработки веб-приложений. Они облегчают разработку, обеспечивая готовые модули и функциональность, а также способы взаимодействия с базами данных, обработки запросов и шаблонизации. При разработке веб-приложений можно использовать такие фреймворки, как Django, Ruby on Rails, ASP.NET, Express.js и другие.
* **Фронтенд-разработка**: При разработке веб-приложений необходимо обращать внимание на фронтенд-разработку, которая отвечает за создание пользовательского интерфейса. Здесь можно применять языки и технологии, такие как HTML, CSS и JavaScript, а также фреймворки и библиотеки, например, React, Angular или Vue.js.
* **Бэкенд-разработка:** Бэкенд-разработка отвечает за обработку запросов от клиента, взаимодействие с базами данных, бизнес-логику и другие серверные задачи. Здесь можно использовать языки программирования, такие как Python, Java, PHP или JavaScript (с использованием Node.js), а также фреймворки и библиотеки для работы с веб-серверами, маршрутизацией запросов, базами данных и т. д.
* **Базы данных:** Веб-приложения часто используют базы данных для хранения и управления данными. При разработке веб-приложений можно применять методы программирования для работы с различными типами баз данных, такими как реляционные базы данных (например, MySQL, PostgreSQL), NoSQL базы данных (например, MongoDB, Cassandra) или базы данных в памяти (например, Redis).

**Практическое задание: ………………**

Сделайте рефакторинг следующей функции методом «Встраивания метода»

**class** **PizzaDelivery** {

// ...

**int** getRating() {

**return** moreThanFiveLateDeliveries() ? 2 : 1;

}

**boolean** moreThanFiveLateDeliveries() {

**return** numberOfLateDeliveries > 5;

}

}

Встраивание метода (Inline Method) - это метод рефакторинга, который заменяет вызов метода его фактическим содержимым. В данном случае, мы можем встроить метод `moreThanFiveLateDeliveries()` в метод `getRating()`. Вот как будет выглядеть рефакторинг с использованием метода "Встраивание метода":

```java

class PizzaDelivery {

// ...

int getRating() {

return numberOfLateDeliveries() > 5 ? 2 : 1;

}

int numberOfLateDeliveries() {

// возвращаем количество задержанных доставок

}

}

```

В результате рефакторинга, метод `moreThanFiveLateDeliveries()` был удален, а его содержимое было встроено непосредственно в метод `getRating()`. Теперь метод `getRating()` напрямую сравнивает количество задержанных доставок `numberOfLateDeliveries()` с числом 5 и возвращает соответствующий рейтинг.

Встраивание метода может быть полезным, когда метод выполняет простую операцию, несет небольшую ответственность и вызывается только в одном месте. Это может улучшить читаемость кода и упростить его структуру. Однако, если метод используется в нескольких местах или выполняет сложную логику, то встраивание метода может быть нецелесообразным.