**Нулевая группа**

1. **Приведите WinApi, необходимое для решения Лабораторной работы №2**

В ходе выполнения Лабораторной работы №2 предстояло работать с потоками. Потоки создавал как объекты ядра с помощью команды CreateThread. Также после их создания необходимо было дождаться их завершения с помощью команды WaitForSingleObject. В конце выполнения основной функции необходимо закрыть потоки как объекты ядра с помощью команды CloseHandle.

1. **Что такое процесс в ОС Windows**

Процесс - объект ядра, которому принадлежат системные ресурсы, используемые приложением. Выполнение каждого процесса начинается с первичного потока. В процессе своего исполнения процесс может создавать другие потоки. Исполнение процесса заканчивается при завершении работы всех его потоков.

1. **Что такое Критическая секция**

Критическая секция – механизм синхронизации, который позволяет ограничивать доступ к общим ресурсам одновременно работающим потокам так, чтобы они не мешали друг другу и не повреждали данные или состояние друг друга. Когда один поток входит в критическую секцию, другие потоки, пытающиеся войти в эту же секцию, будут заблокированы и ждать, пока первый поток не завершит выполнение кода внутри секции.

1. **Что такое Семафор**

Семафор – это неотрицательная целая переменная, значение которой может изменяться только при помощи операций, выполнение которых не может быть прервано. Семафор считается свободным, если его значение больше нуля, в противном случае семафор считается занятым.

1. **Сравнительный анализ стандарта C++ 98 с и без применения библиотеки boost (в контексте лабораторных)**

На момент написания этой работы мной было решено 3 лабораторных работы (1-3). Библиотека boost, а именно boost::thread, использовалась мной для решения лабораторной работы №2. Стандарт С++98 мне показался более трудозатратным по изучению документации прежде чем начать работу. Требуется детальное знание команд и передаваемых параметров (что за что отвечает). Т.е для работы с потоками в С++98 требовалось создать функцию определённого типа, создать handl’ы, проинициализировать их как потоки с передачей функции, следить чтоб они завершили свою работу, закрыть их… В то время как при использовании библиотеки boost::thread объём работы заметно сократился. Можно создать привычную функцию стандартных типов (например int). Потом создать поток с переданной в него функцией и параметрами, а дальше просто подключить поток командой join и он сам всё сделает (подождёт, даст результат, закроется). + к тому же библиотека boost содержит собственные удобные функции в контексте работы с потоками (например можно обратиться к потоку через this::thread , а не хранить его в переменной).

**Вторая группа**

1. **Что такое Процедурная декомпозиция**

Процедурная декомпозиция - подход в программировании, который используется для разделения сложной задачи или программы на более мелкие и управляемые части, называемые процедурами. Этот подход помогает улучшить структуру программы, сделать её более понятной, легко управляемой и сопровождаемой.

Основные принципы процедурной декомпозиции включают:

Разделение на функции или процедуры: Сложная задача разделяется на более мелкие части или подзадачи, каждая из которых решается отдельной функцией или процедурой. Эти функции могут быть независимыми и выполнять конкретные задачи.

Модульность: Каждая процедура или функция должна выполнять конкретную задачу, и их интерфейсы (параметры и возвращаемые значения) должны быть хорошо определены.

Абстракция: Процедуры позволяют скрыть детали реализации и работать с абстрактными интерфейсами. Это упрощает понимание и использование кода.

Изоляция и инкапсуляция: Процедуры могут быть инкапсулированы, что означает, что их локальные переменные и состояние недоступны извне. Это помогает избежать конфликтов и нежелательных взаимодействий между разными частями программы.

Повторное использование кода: Процедуры могут быть повторно использованы в разных частях программы или в разных программах, что способствует экономии времени и ресурсов.

1. **Что такое динамический полиморфизм**

Динамический полиморфизм - это один из ключевых концепций объектно-ориентированного программирования, который позволяет объектам разных классов реагировать на одну и ту же операцию (метод) специфичным для своего класса способом. Этот вид полиморфизма позволяет вызывать методы объектов на основе их фактического типа во время выполнения программы.

Основные элементы динамического полиморфизма включают:

Базовый класс и производные классы: В контексте ООП, у вас есть базовый (родительский) класс и один или более производных (дочерних) классов, которые наследуют от базового класса.

Полиморфные методы: У базового и производных классов могут быть методы с одинаковой сигнатурой (именем и параметрами), но с разной реализацией. Эти методы называются полиморфными методами.

Использование указателей или ссылок: Для достижения динамического полиморфизма объекты производных классов могут быть использованы через указатели или ссылки на базовый класс.

Вызов методов посредством указателей или ссылок: Вызов метода на объекте производного класса, хранимого в указателе или ссылке базового класса, приводит к выполнению соответствующей реализации метода в производном классе.

1. **Что такое инкапсуляция**

Инкапсуляция - это один из основных принципов объектно-ориентированного программирования (ООП) и концепция, которая позволяет скрыть внутренние детали реализации объекта от внешнего мира и предоставить интерфейс для взаимодействия с этим объектом. Инкапсуляция обеспечивает защиту данных и методов объекта, делая их доступными только через определенные публичные методы, называемые методами доступа (геттеры и сеттеры).

Основные принципы инкапсуляции включают:

Сокрытие данных: Переменные-члены класса (поли) обычно объявляются как закрытые (private) или защищенные (protected), что означает, что они не доступны напрямую извне класса. Это предотвращает непосредственное изменение или чтение данных объекта извне.

Публичные методы доступа: Для доступа к данным объекта предоставляются публичные методы, такие как геттеры и сеттеры. Эти методы обеспечивают контролируемый доступ к данным и позволяют выполнять проверки или манипуляции перед доступом к данным.

Сокрытие реализации: Инкапсуляция также позволяет скрывать детали реализации методов и действий объекта от внешнего кода. Внешние объекты знают, что они могут выполнить некоторые операции с объектом, но не знают, как именно они реализованы.

**Третья группа**

1. **Абстрактная фабрика**

Назначение: абстрактная фабрика предоставляет интерфейс для создания семейства взаимосвязанных или родственных объектов, не специфицируя их конкретных классов.

Другими словами: абстрактная фабрика представляет собой стратегию создания семейства взаимосвязанных или родственных объектов.

Когда следует использовать абстрактную фабрику

1) Когда система не должна зависеть от способа создания новых объектов

2) Когда создаваемые объекты должны использоваться вместе и являются взаимосвязанными

В контексте "Абстрактной фабрики", инкапсуляция происходит на двух уровнях:

Инкапсуляция создания объектов: Абстрактная фабрика позволяет клиентскому коду создавать объекты, не зная конкретных классов этих объектов. Клиент взаимодействует с фабрикой через абстрактный интерфейс, и фабрика создает объекты, которые соответствуют конкретной вариации или стилю. Это скрывает детали создания объектов от клиента.

Инкапсуляция вариаций объектов: Абстрактная фабрика позволяет создавать семейства объектов, которые совместимы между собой. Каждая конкретная реализация абстрактной фабрики может создавать объекты с разными характеристиками, и клиентский код может использовать объекты из одной фабрики вместе, не зная подробностей о различиях между объектами.

В этом репозитории приведён пример использования шаблона «Абстрактная фабрика» на языке Java.

1. **Facade**

Назначение: предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы. Фасад определяет интерфейс более высокого уровня, который упрощает использование подсистемы.

Шаблон Фасад объединяет группу объектов в рамках одного специализированного интерфейса и переадресует вызовы его методов к этим объектам.

Когда использовать фасад?

1) Когда имеется сложная система, и необходимо упростить с ней работу. Фасад позволит определить одну точку взаимодействия между клиентом и системой.

2) Когда надо уменьшить количество зависимостей между клиентом и сложной системой. Фасадные объекты позволяют отделить, изолировать компоненты системы от клиента и развивать и работать с ними независимо.

3) Когда нужно определить подсистемы компонентов в сложной системе. Создание фасадов для компонентов каждой отдельной подсистемы позволит упростить взаимодействие между ними и повысить их независимость друг от друга.

В контексте шаблона "Фасад" инкапсуляция происходит следующим образом:

Инкапсуляция подсистем: Фасад инкапсулирует в себе множество подсистемных классов и компонентов, которые могут быть сложными и иметь разные интерфейсы.

Унифицированный интерфейс: Фасад предоставляет унифицированный интерфейс, который скрывает детали взаимодействия с подсистемой. Этот интерфейс обычно предоставляет простые методы, с помощью которых клиентский код может взаимодействовать с подсистемой без необходимости знать её сложную структуру.

Снижение сложности: Фасад уменьшает сложность клиентского кода, упрощая его взаимодействие с подсистемой. Клиенту не нужно заботиться о деталях внутренней реализации системы, так как фасад предоставляет абстракцию над всеми подсистемными компонентами.

В этом репозитории приведён пример использования шаблона «Фасад» на языке Java.