Министерство образования Республики Беларусь

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Инженерно-экономический факультет

Кафедра экономической информатики

Лабораторная работа № 10

«ОСОБЕННОСТИ ИСПОЛЬЗОВАНИЯ ОБЪЕКТНО ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ И ПАТТЕРНОВ  ПРОЕКТИРОВАНИЯ»

Вариант 1

Выполнил:         Студент гр.272303

Бадей П.Ю.

Проверил: Старший преподаватель кафедры ЭИ

         Салапура М. Н.

Минск 2023

Цель работы: изучить особенности использования объектно ориентированного программирования и паттернов проектирования.

Задачи: написать код к приложенным заданиям, пользуясь приобретенными знаниями, понять принцип работы контейнеров и итераторов стандартной библиотеки шаблонов.

**Краткие теоретические сведения**

В языке С++ методы класса, в том числе и деструктор, могут быть  виртуальным, кроме конструктора. Если метод не виртуальный, то  точный тип объекта, для которого он вызывается, известен на этапе  компиляции. Поэтому тип конструируемого объекта всегда известен во  время компиляции, в той точке, где вызывается конструктор. Тем не  менее часто необходимо конструировать объекты, тип которых  становится известен только на этапе выполнения. В данной лабораторной  работе описаны паттерны проектирования (подходы к проектированию  систем) и идиомы, решающие эту проблему.

Когда виртуальная функция вызывается через указатель или ссылку  на базовый класс, то этот указатель или ссылка используется для доступа  к так называемому v-указателю в классе (указателю на таблицу  виртуальных функций или v-таблицу). V-указатель позволяет определить  истинный тип объекта, т.е. тип, который был указан в момент его

создания. Это может быть как сам базовый класс, так и один из производных от него классов. Затем вызывается метод этого класса. Описанная выше процедура не может работать для создания виртуальных конструкторов. Прежде всего понятно, что частью этого процесса  является определение типа объекта, указанного в момент его создания*.* Это  возможно только после того, как объект сконструирован – до этого момента  еще не существует объекта данного типа, а лишь неинициализированная  область динамической памяти.

**Контрольные вопросы**

\*\*1. Преимущества объектно-ориентированного проектирования:\*\*

- \*\*Инкапсуляция:\*\* Скрытие деталей реализации, что способствует легкости сопровождения и улучшения кода.

- \*\*Наследование:\*\* Позволяет создавать новые классы на основе существующих, обеспечивая повторное использование кода и иерархию классов.

- \*\*Полиморфизм:\*\* Обеспечивает использование различных типов объектов через общий интерфейс, что упрощает обмен компонентами системы.

\*\*2. Пример функции с переменным количеством аргументов:\*\*

#include <iostream>

#include <cstdarg>

double average(int count, ...) {

va\_list args;

va\_start(args, count);

double sum = 0;

for (int i = 0; i < count; ++i) {

sum += va\_arg(args, double);

}

va\_end(args);

return sum / count;

}

int main() {

std::cout << "Average: " << average(3, 2.0, 4.0, 6.0) << std::endl;

return 0;

}

\*\*3. Пример функции, получающей указатель на функцию:\*\*

#include <iostream>

int add(int a, int b) {

return a + b;

}

int subtract(int a, int b) {

return a - b;

}

int calculate(int a, int b, int (\*operation)(int, int)) {

return operation(a, b);

}

int main() {

std::cout << "Addition: " << calculate(5, 3, add) << std::endl;

std::cout << "Subtraction: " << calculate(5, 3, subtract) << std::endl;

return 0;

}

\*\*4. Пример создания и использования перечисления:\*\*

#include <iostream>

enum Color { RED, GREEN, BLUE };

int main() {

Color myColor = GREEN;

switch (myColor) {

case RED:

std::cout << "The color is Red." << std::endl;

break;

case GREEN:

std::cout << "The color is Green." << std::endl;

break;

case BLUE:

std::cout << "The color is Blue." << std::endl;

break;

default:

std::cout << "Unknown color." << std::endl;

}

return 0;

}

\*\*5. Виртуальные конструкторы и виртуальные деструкторы в C++:\*\*

- Виртуальные конструкторы не разрешены в C++. Конструкторы не наследуются, и вызов конструктора базового класса обеспечивается автоматически.

- Виртуальный деструктор необходим для правильного освобождения памяти при удалении объекта через указатель на базовый класс.

\*\*6. Назначение паттерна Фабрика:\*\*

- Паттерн Фабрика используется для создания объектов без явного указания их классов. Это позволяет создавать объекты, не привязываясь к конкретным классам, а используя интерфейс или абстрактный класс.

\*\*7. Использование паттерна Фабрика для создания эффекта виртуального конструктора:\*\*

- Фабрика создает объекты через интерфейс, обеспечивая абстракцию от конкретных классов. Таким образом, она может служить заменой виртуального конструктора.

\*\*8. Эффект виртуального копирующего конструктора:\*\*

- Эффект виртуального копирующего конструктора можно добиться с использованием паттерна клонирования. Классы должны реализовать метод клонирования, который будет возвращать новый объект того же типа.

\*\*9. Константа времени компиляции в C++:\*\*

- Константа времени компиляции - это значение, известное на этапе компиляции. Например, `const int size = 10;`.

\*\*10. Получение имени типа:\*\*

- В C++ получение имени типа напрямую во время выполнения сложно. Однако, можно использовать `typeid` для получения информации о типе:

#include <iostream>

#include <typeinfo>

int main() {

int value;

std::cout << typeid(value).name() << std::endl; // Выводит имя типа переменной value

return 0;

}

**Индивидуальное задание**

Во всех вариантах индивидуальных заданий необходимо создать  возможность конструирования 5 ОБЪЕКТОВ РАЗЛИЧНЫХ ТИПОВ с  использованием подходов «ФАБРИКА», «ФАБРИЧНЫЙ МЕТОД С  АРГУМЕНТАМИ», «ФАБРИЧНЫЙ КОНСТРУКТОР», «ПОЛИМОРФНАЯ

ФАБРИКА», «ПОЛИМОРФНОЕ КОПИРОВАНИЕ». Для всех типов  необходимо реализовать возможность их добавления, удаления,  редактирования, вывода содержимого объектов на экран в табличном виде и  в файл.

Тема: печатная продукция.

**Листинг кода**

#ifndef PAPERPRODUCT\_HPP

#define PAPERPRODUCT\_HPP

#include "Unit.hpp"

class PaperProduct;

int paperProductsCount = 0;

std::vector<std::pair<int, PaperProduct\*>> paperProductRegistryVector;

class PaperProduct {

int papersCount;

std::string manufacturerName;

public:

PaperProduct() : papersCount(0), manufacturerName("none") {};

PaperProduct(const int& paperCount\_, const std::string& manufacturerName\_);

PaperProduct(const PaperProduct\* object) {

this->papersCount = object->papersCount;

this->manufacturerName = object->manufacturerName;

}

static void registerPaperProduct(PaperProduct\* factory) {

paperProductRegistryVector.push\_back(std::make\_pair(paperProductsCount++, factory));

}

static PaperProduct\* makePaperProduct(const int& paperProductType) {

PaperProduct\* factory = paperProductRegistryVector.at(paperProductType).second;

return factory;

}

virtual ~PaperProduct() {};

virtual PRODUCTS paperProductsType() const = 0;

virtual PaperProduct\* clone() const = 0;

};

inline PaperProduct::PaperProduct(const int& papersCount\_, const std::string& manufacturerName\_) {

this->papersCount = papersCount\_;

this->manufacturerName = manufacturerName\_;

}

#endif

**Скриншоты выполнения индивидуального задания**

 \

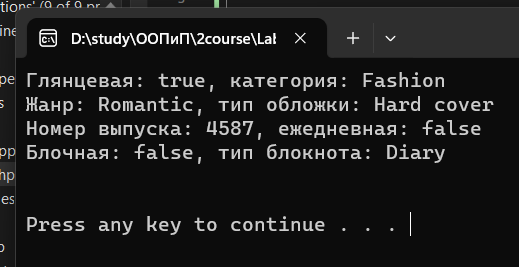


Рисунок 1.1 – Заказанные продукты

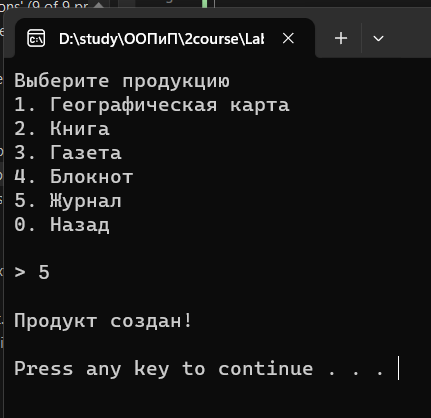


Рисунок 1.2 – Создание журнала

**Диаграмма всех созданных классов**

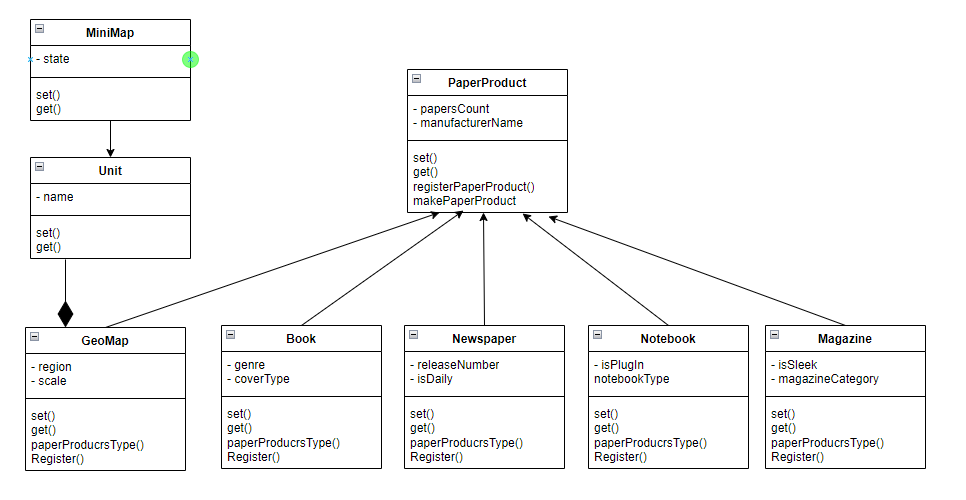


Рисунок 1.4 – Диаграмма созданных классов

**Выводы:** изучили особенности работы с ассоциативными контейнерами, написали код к поставленной задаче.