ЛАБОРАТОРНАЯ РАБОТА №1 ПО ПРЕДМЕТУ «ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ И ПРОГРАММИРОВАНИЕ»

ТЕМА «МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ, ВИРТУАЛЬНОЕ НАСЛЕДОВАНИЕ»

ЦЕЛЬ РАБОТЫ: изучить принципы и получить практические навыки при использовании множественного наследования; рассмотреть случаи, когда необходимо использовать виртуальное наследование.

В языке *C++* имеется возможность наследовать производный класс одновременно от двух и более базовых классов. Общая форма такого наследования имеет следующий вид:

```
class BaseClass1 {};
class BaseClass2 {};
class DerivedClass : public BaseClass1, public BaseClass2 {
        //содержимое класса DerivedClass
};
```

При множественном наследовании имена базовых классов разделяются запятыми, и перед каждым именем базового класса указывается свой спецификатор наследования (*public, private, protected*).

Иерархическая структура, в которой производный класс одновременно наследуется от нескольких базовых классов, называется множественным наследованием. В этом случае производный класс, имея собственные компоненты, имеет доступ *к protected-* и *public*-компонентам базовых классов. Конструкторы базовых классов при создании объекта производного класса вызываются в том порядке, в котором они указаны в списке при объявлении производного класса (рисунок 1).

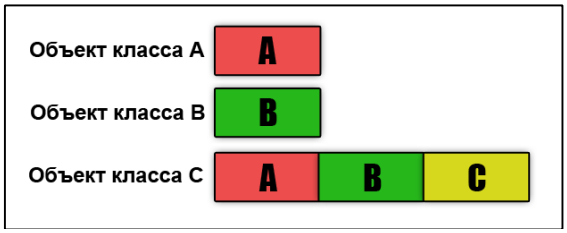


Рисунок 1 – Расположение в памяти элементов объекта класса, имеющего множественное наследование

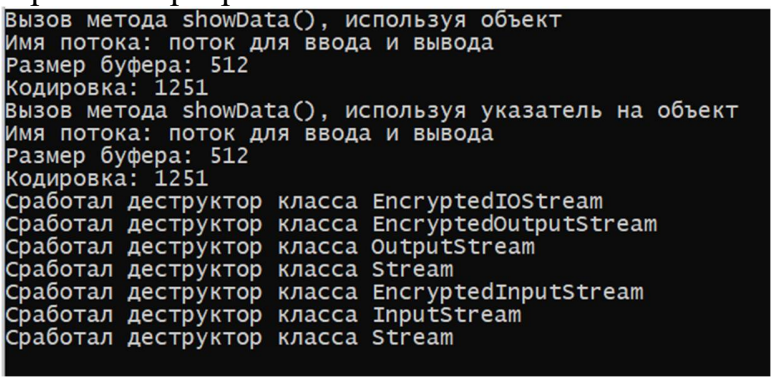
Ниже приведен пример программы, использующей множественное наследование.

```
//Пример №1. Использование множественного наследования классов
#include <iostream>
#include <string.h>
#include <windows.h>
using namespace std;
class Stream {
protected:
	string name; // название потока
	int bufferSize;
public:
	Stream(string name, int bufferSize);
	~Stream() { cout << "Сработал деструктор класса Stream" << endl; }
	void showData();
};
class InputStream : public Stream {
public:
	int read();
	InputStream(string name, int bufferSize);
	~InputStream() { cout << "Сработал деструктор класса InputStream" << endl; }
	void showData();
};
class OutputStream : public Stream {
public:
	int write();
	OutputStream(string name, int bufferSize);
	~OutputStream() { cout << "Сработал деструктор класса OutputStream" << endl; }
	void showData();
};
class EncryptedInputStream :public InputStream {
protected:
	string encoding; // кодировка используемая в потоке
public:
	EncryptedInputStream(string encoding, string name, int bufferSize);
	~EncryptedInputStream() { cout << "Сработал деструктор класса
EncryptedInputStream" << endl; }
	void showData();
};
class EncryptedOutputStream :public OutputStream {
protected:
	string encoding; // кодировка используемая в потоке
public:
	EncryptedOutputStream(string encoding, string name, int bufferSize);
```

```
        ~EncryptedOutputStream() { cout << "Сработал деструктор класса
EncryptedOutputStream" << endl; }
        void showData();
};
class EncryptedIOStream : public EncryptedInputStream, public  EncryptedOutputStream {
public:
        EncryptedIOStream(string encoding, string name, int bufferSize);
        ~EncryptedIOStream() { cout << "Сработал деструктор класса EncryptedIOStream" <<
endl; }
        void showData();
};
Stream::Stream(string name, int bufferSize):bufferSize(bufferSize)
{ this->name = name; }
InputStream::InputStream(string name, int bufferSize) : Stream(name, bufferSize) {}
OutputStream::OutputStream(string name, int bufferSize) : Stream(name, bufferSize) {}
EncryptedInputStream::EncryptedInputStream(string encoding, string name, int bufferSize) :
InputStream(name, bufferSize) { this->encoding = encoding; }
EncryptedOutputStream::EncryptedOutputStream(string encoding, string name, int bufferSize) :
OutputStream(name, bufferSize) { this->encoding = encoding; }
EncryptedIOStream::EncryptedIOStream(string encoding, string name, int bufferSize) :
        EncryptedInputStream(encoding, name, bufferSize),
        EncryptedOutputStream(encoding, name, bufferSize) {};
void Stream::showData() {
        cout << "Имя потока: " << name << endl;
        cout << "Размер буфера: " << bufferSize << endl;
}
void InputStream::showData() { Stream::showData(); }
void OutputStream::showData() { Stream::showData(); }
void EncryptedInputStream::showData() {
        InputStream::showData();
        cout << "Кодировка: " << encoding << endl; }
void EncryptedOutputStream::showData() {
        OutputStream::showData();
        cout << "Кодировка: " << encoding << endl;
}
void EncryptedIOStream::showData() {
        EncryptedInputStream::showData();
}
void ex1() {
        SetConsoleCP(1251);
        SetConsoleOutputCP(1251);
        EncryptedIOStream encryptedIOStream("1251", "поток для ввода и вывода", 512);
        EncryptedIOStream* ptrEncryptedIOStream = &encryptedIOStream;
        cout << "Вызов метода showData(), используя объект" << endl;
        encryptedIOStream.showData(); // вызов метода showData(), используя объект
        cout << "Вызов метода showData(), используя указатель на объект" << endl;
```

```
	ptrEncryptedIOStream->showData(); // вызов метода showData, используя указатель на объект
}
```

Результат работы программы:



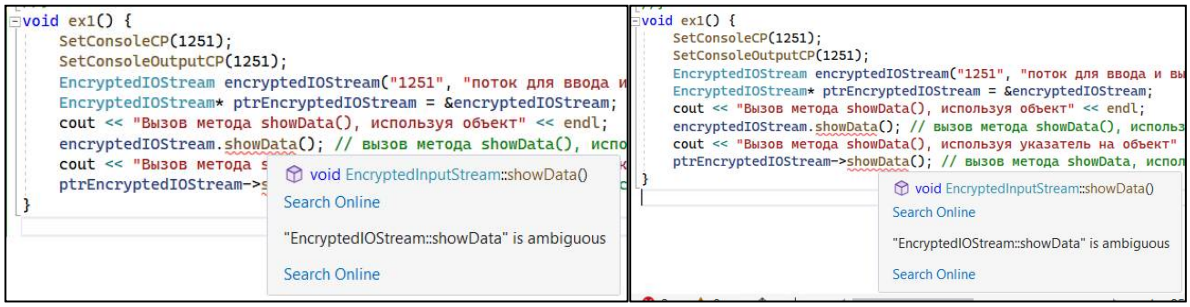
При применении множественного наследования возможно возникновение нескольких конфликтных ситуаций. ***Первая конфликтная ситуация*** возникает из-за конфликта имен методов или полей нескольких базовых классов:

```
class EncryptedInputStream :public InputStream {
protected:
	string encoding; // кодировка используемая в потоке
public:
	//...
	void showData();
};

class EncryptedOutputStream :public OutputStream {
protected:
	string encoding; // кодировка используемая в потоке
public:
	//...
	void showData();
};

class EncryptedIOStream : public EncryptedInputStream, public  EncryptedOutputStream {
public:
	EncryptedIOStream(string encoding, string name, int bufferSize);
	~EncryptedIOStream() { cout << "Сработал деструктор класса EncryptedIOStream" << endl; }
	//void showData();
};
void ex1() {
	SetConsoleCP(1251);
```

```
    SetConsoleOutputCP(1251);
    EncryptedIOStream encryptedIOStream("1251", "поток для ввода и вывода", 512);
    EncryptedIOStream* ptrEncryptedIOStream = &encryptedIOStream;
    cout << "Вызов метода showData(), используя объект" << endl;
    encryptedIOStream.showData(); // вызов метода showData(), используя объект
    cout << "Вызов метода showData(), используя указатель на объект" << endl;
    ptrEncryptedIOStream->showData(); // вызов метода showData, используя указатель
на объект
}
```



При таком вызове функции *showData()* компилятор не может определить, к какой из двух функций классов *InputStream* и *OutputStream* выполняется обращение. Неоднозначность можно устранить, явно указав, какому из базовых классов принадлежит вызываемая функция:

```
encryptedIOStream.EncryptedInputStream::showData();//вызов метода showData(), используя объект
ptrEncryptedIOStream->EncryptedOutputStream::showData();//вызов метода showData, используя указатель на объект
```

Вторая конфликтная ситуация возникает при многократном включении некоторого базового класса:

```
class Stream {
public:
    void showData() {}
};
class InputStream : public Stream {
    // компоненты класса InputStream
};
class OutputStream : public Stream {
    // компоненты класса OutputStream
};
class InputOutputStream : public InputStream, public OutputStream { };
int main() {
    InputOutputStream ioStream;
    ioStream.showData();//InputOutputStream::showData is ambiguous
    return 0;
}
```

Проблема состоит в том, что при создании объекта класса *InputOutputStream* создаются два (одинаковых) объекта базового класса *Stream* (для объектов класса *InputStream* и *OutputStream*). Решение этой проблемы состоит в использовании **виртуального наследования (*virtual inheritance*):**

```
class InputStream : public  virtual Stream {
        // компоненты класса InputStream
};
class OutputStream : public virtual Stream {
        // компоненты класса OutputStream
};
class InputOutputStream : public InputStream, public OutputStream { };
```

В этом случае при создании объекта класса *InputOutputStream* происходит однократное создание виртуального объекта класса *Stream*. **Виртуальное наследование базового класса гарантирует, что в любом производном классе будет присутствовать только одна его копия.**

Применение оператора "::" (разрешения контекста, разрешения области видимости) позволяет программе "ручным способом" выбрать версию класса. Но если необходима только одна копия базового класса, то это достигается с помощью виртуального наследования базовых классов.

Если два (или больше) класса выведены из общего базового класса, то можно предотвратить включение нескольких копий базового класса в объекте дочерних классов, выведенного из этих классов, что реализуется путем объявления наследовании базового класса виртуальным. Для этого достаточно предварить имя наследуемого базового класса ключевым словом *virtual*.

Разница между обычным базовым и виртуальным классами становится очевидной только тогда, когда этот базовый класс наследуется более одного раза. Если наследование базового класса объявляется виртуальным, то только один его экземпляр будет включен в объект наследующего класса. В противном случае в этом объекте будет присутствовать несколько его копий.

Рассмотрим следующую иерархию классов:

```
class Animal {
public: void eat();
};
class Mammal : public Animal {//(mammal – млекопитающее)
public: string getColor();
};
class WingedAnimal : public Animal {// (winged – с крыльями)
```

```
public: void flap();
};
//A bat is a winged mammal (Летучая мышь относится к классу млекопитающие,отряд рукокрылые)
class Bat : public Mammal, public WingedAnimal {};
//обратите внимание, что метод eat() не переопределен в классе Bat
int main() {
        Bat bat;
        bat.eat();
}
```

Для вышеприведенного кода вызов *bat.eat()* является неоднозначным. Он может относиться как к *Bat::WingedAnimal::Animal::eat()* так и к *Bat::Mammal::Animal::eat()*. У каждого промежуточного наследника (*WingedAnimal*, *Mammal*) метод *eat()* может быть переопределен (это не меняет сущность проблемы с точки зрения языка). Проблема в том, что семантика традиционного множественного наследования не соответствует моделируемой им реальности. В некотором смысле, сущность *Animal* единственна по сути; *Bat* — это *Mammal* и *WingedAnimal*, но свойство животности (*Animalness*) летучей мыши (*Bat*), оно же свойство животности млекопитающего (*Mammal*) и оно же свойство животности *WingedAnimal* — по сути это одно и то же свойство. Такая ситуация обычно именуется «ромбическое, ромбовидное наследование» (*diamond inheritance*, рисунок 2) и представляет собой проблему, которую призвано решить виртуальное наследование.

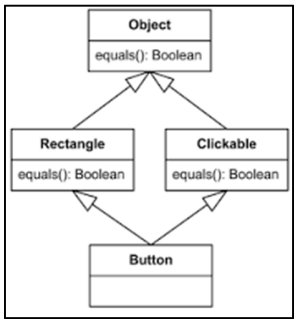


Рисунок 2 – Диаграмма наследования классов в виде ромба

При наследовании часть базового класса и наследника помещаются в оперативной памяти друг за другом. Таким образом, объект класса *Bat* это на самом деле последовательность объектов классов (*Animal*, *Mammal*, *Animal*, *WingedAnimal*, *Bat*), размещенных последовательно в памяти, при этом *Animal* повторяется дважды, что и приводит к неоднозначности. Можно переопределить классы следующим образом:

```
class Animal {
public: void eat();
};
class Mammal : public virtual Animal {//(mammal – млекопитающее)
public: string getColor();
};
class WingedAnimal : public virtual Animal {// (winged – с крыльями)
public: void flap();
};
//A bat is a winged mammal (Летучая мышь относится к классу млекопитающие,отряд рукокрылые)
class Bat : public Mammal, public WingedAnimal {};
//обратите внимание, что метод eat() не переопределен в классе Bat
int main() {
	Bat bat;
	bat.eat();
}
```

Теперь, часть *Animal* объекта класса *Bat::WingedAnimal* та же самая, что и часть *Animal*, которая используется в *Bat::Mammal*, и можно сказать, что *Bat* имеет в своем представлении только одну часть *Animal* и вызов *Bat::eat()* становится однозначным.

Виртуальное наследование реализуется через добавление указателей в классы *Mammal* и *WingedAnimal*. Таким образом, *Bat* представляется, как (**ptr*, *Mammal*, **ptr*, *WingedAnimal*, *Bat*, *Animal*). **ptr* содержит информацию о смещении в памяти между началом *Mammal* и его *Animal*. Это необходимо, потому что для указателя

```
Animal* p;
```

который может ссылаться на *Animal*, на *Mammal*, на *WingedAnimal*, смещение в памяти между началом объекта и его *Animal* части неизвестно на этапе компиляции, а выясняется только во время выполнения.

Пример множественного наследования из области разработки графических интерфейсов: класс *Button* может одновременно наследоваться от класса *Rectangle* (для внешнего вида) и от класса *Clickable* (для реализации функционала (обработки события нажатия)), а *Rectangle* и *Clickable* наследуются от класса *Object* (рисунок 2). Если вызвать метод *equals()* для объекта *Button*, и в классе *Button* не окажется такого метода, но в классе *Object* будет присутствовать метод *equals()* по-своему переопределенный как в классе *Rectangle*, так и в *Clickable*, то возникает вопрос какой из методов должен быть вызван.

Языки, допускающие лишь простое наследование классов (*PHP*, C#, *Java*), предусматривают множественное наследование интерфейсов. Интерфейсы, по сути, являются абстрактными базовыми классами абстрактными методами и без полей экземпляра, но могут иметь общие разделяемые константы. Таким образом, проблема не возникает, так как всегда будет только одна реализация определенного метода или свойства, не допуская возникновения неопределенности.

Проблема ромба не ограничивается лишь наследованием. Она также возникает в таких языках, как *C* и *C++,* когда заголовочные файлы *A, B, C* и *D*, а также отдельные предкомпилированные заголовки, созданные из *B* и *C*, подключаются (при помощи инструкции *#include*) один к другому по ромбовидной схеме. Если эти два предкомпилированных заголовка объединяются, объявления *A* дублируются, и директива защиты подключения *#ifndef* становится неэффективной.

ЧТЕНИЕ ГРАФОВ НАСЛЕДОВАНИЯ

Связи между классами могут изображаются графически и с помощью языка моделирования *UML*, что облегчает их понимание и повышает презентативность. Например, диаграмма классов демонстрирует общую структуру иерархии классов системы, их кооперацией, атрибутов (полей), методов, интерфейсов и взаимосвязей (отношений) между ними. Диаграмма классов широко используется не только для документирования и визуализации, но также для конструирования посредством прямого или обратного проектирования. Рассмотрим ситуацию, в которой класс *Employee* наследует класс *PermanentEmployee*, который в свою очередь наследуется классом *PermanentManager* (рисунок 4):

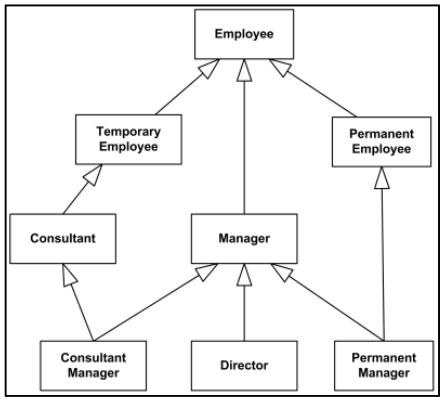


Рисунок 3 – Пример иерархии наследования с использованием языка *UML*

Стрелки на этом рисунке направлены вверх к базовому классу. **Согласно стилевой графике C++ стрелка наследования (обобщения, генерализации, generalization) должна указывать на базовый класс.** Следовательно, стрелка означает "выведен из", "является базовым".

Класс *ConsultantManager* наследуется от классов *Manager* и *Consultant*. (Другими словами, класс *ConsultantManager* имеет два базовых класса *Manager* и *Consultant*). При этом класс *Consultant* наследуется от класса *TemporaryEmployee*, а класс *Manager* — от класса *Employee*. Этот стиль графических обозначений широко используется в книгах, журналах, презентациях, документациях к языкам, платформам, фреймворкам.

Примеры использования множественного наследования показаны на рисунке 4.

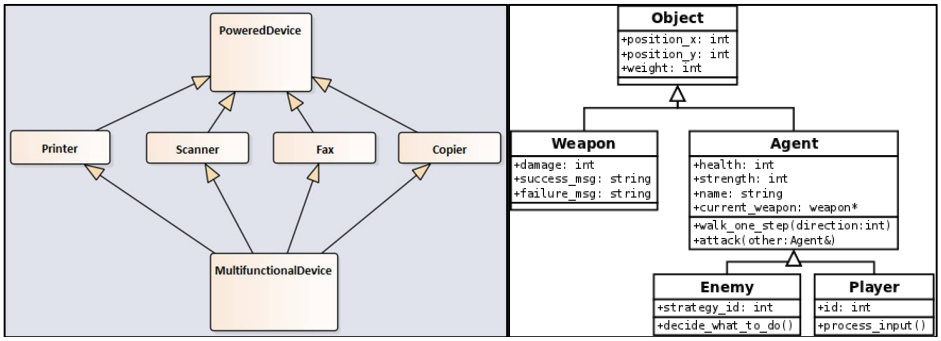


Рисунок 4 – Примеры использования множественного наследования

Применение множественного наследования может быть оправданно, если оно делает общую архитектуру системы более гибкой или практичной. Примером множественного наследования может также служить исходная архитектура библиотеки потоков ввода-вывода языка C++, которая сохранилась и в современной шаблонной архитектуре (рисунок 5).

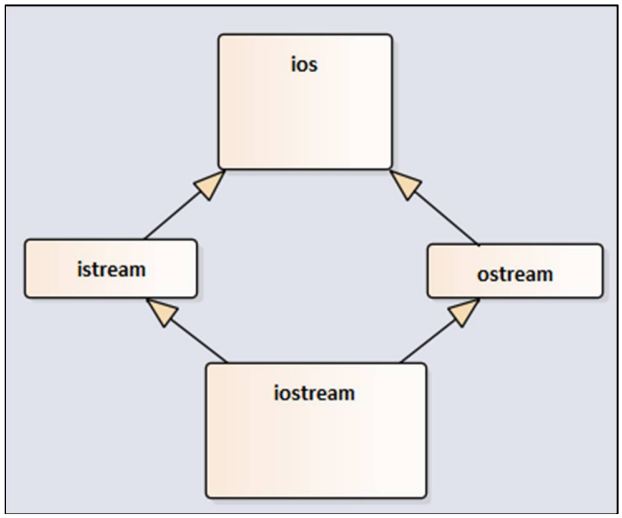


Рисунок 5 – Связи между классами библиотеки ввода-вывода языка C++

Классы *istream* и *ostream* сами по себе являются полезными, но они также могут быть объединены посредством множественного наследования в класс, сочетающий их характеристики и поведение. Класс *ios* предоставляет общую функциональность всех потоковых классов, поэтому в данном случае множественное наследование является механизмом логического структурирования программы.

НАСЛЕДОВАНИЕ ИНТЕРФЕЙСА

Одно из применений множественного наследования, не вызывающее никаких возражений, связано с наследованием интерфейса. В языке *C++* все наследование является наследованием реализации, поскольку все аспекты базового класса, интерфейс и реализация становятся частью производного класса. Унаследовать только часть класса (скажем, интерфейс) невозможно. Защищенное и закрытое наследование позволяет ограничить доступ к элементам, унаследованным от базового класса, со стороны клиентов объекта производного класса, но на самом производном классе это не сказывается; он все равно содержит все данные базового класса и может обращаться ко всем незакрытым элементам базового класса.

С другой стороны, наследование интерфейса только добавляет объявления функций в интерфейс производного класса. Такая возможность не поддерживается в языке *C++* напрямую. Стандартная методика имитации наследования интерфейса основана на наследовании от интерфейсного класса, то есть класса, содержащего только объявления методов (но не их определения). Все объявления интерфейсного класса, кроме деструктора, должны быть чисто виртуальными функциями.

```
//Пример №2. Множественное наследование классов-интерфейсов
```

```
#include <iostream>
#include <sstream>
#include <string>
using namespace std;
class Printable {
public:
	virtual ~Printable() {}
	virtual void print(ostream&) const = 0;
};
class Stringable {
public:
	virtual ~Stringable() {}
	virtual string toString() const = 0;
};
class Number : public Printable, public Stringable {
	int date;
public:
	Number(int data) { this->date = data; }
	void print(ostream& os) const {
		os << date;
	}
	int toInt() const {
		return date;
	}
	string toString() const {
		ostringstream os;
		os << date;
		return os.str();
	}
};
void testPrintable(const Printable& p) {
	p.print(cout);
	cout << endl;
}

void testStringable(const Stringable& s) {
	string buf = s.toString() + "th";
	cout << buf << endl;
}
void ex2() {
	Number a(7);
	testPrintable(a);
	testStringable(a);
}
```

Результат работы программы:

Класс *Number* «реализует» интерфейсы *Printable* и *Stringable*, то есть предоставляет реализации для методов, объявленных в этих классах. Так как *Number* наследуется от двух классов, объекты *Number* воплощают множественные связи типа «является частным случаем». Например, объект *number* может использоваться как объект *Printable*, потому что его класс *Number* открыто наследует от *Printable* и предоставляет реализацию метода *print()*.

Одна из причин, по которым в языке *C++* поддерживается множественное наследование, состоит в том, что *C++* является гибридным языком и не может использовать единую монолитную иерархию классов, как Java. Вместо этого *C++* позволяет создавать множество самостоятельных деревьев наследования, поэтому иногда требуется объединить интерфейсы двух и более деревьев в новый класс.

Если иерархия не содержит «ромбов», множественное наследование обходится без особых сложностей. При появлении ромбовидных структур желательно избавиться от дублирования подобъектов за счет введения виртуальных базовых классов.

Множественное наследование называют «командой *goto* 90-х годов». Сравнение весьма удачное: множественного наследования, как и команды *goto*, при нормальном программировании лучше избегать, но в отдельных случаях оно оказывается очень полезным. Оно принадлежит к числу «второстепенных», но крайне нетривиальных возможностей C++, предназначенных для решения проблем, возникающих в особых ситуациях.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Для чего используется множественное наследование? Чем оно отличается от простого наследования?
2. Каков порядок вызова конструкторов и деструкторов при множественном наследовании?
3. Каков порядок вызова конструкторов и деструкторов при множественном виртуальном наследовании?
4. Какие проблемы возможны при множественном наследовании и как они разрешаются при использовании виртуального наследования?
5. Какие языки не поддерживают множественного наследования классов? Почему?

ПОРЯДОК ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ:

1. Изучить теоретические сведения, полученные на лекции и лабораторной работе, ознакомиться с соответствующими материалами литературных источников.

2. Ответить на контрольные вопросы лабораторной работы.

3. Разработать алгоритм программы по индивидуальному заданию.

4. Написать, отладить и проверить корректность работы созданной программы.

5. Написать электронный отчет по выполненной лабораторной работе.

Отчет должен быть оформлен по стандарту БГУИР (Стандарт предприятия СТП 01-2017 "Дипломные проекты (работы). Общие требования"**) и иметь следующую структуру:**

1. титульный лист
2. цель выполнения лабораторной работы
3. теоретические сведения по лабораторной работе
4. формулировка индивидуального задания
5. весь код решения индивидуального задания, разбитый на необходимые типы файлов
6. скриншоты выполнения индивидуального задания
7. выводы по лабораторной работе

В РАМКАХ ВСЕГО КУРСА ООПиП ВСЕ ЛАБОРАТОРНЫЕ РАБОТЫ ДОЛЖНЫ ХРАНИТЬСЯ В ОДНОМ РЕШЕНИИ (SOLUTION), В КОТОРОМ ДОЛЖНЫ БЫТЬ СОЗДАНЫ ОТДЕЛЬНЫЕ ПРОЕКТЫ (PROJECTS) ДЛЯ КАЖДОЙ ЛАБОРАТОРНОЙ РАБОТЫ. ВО ВСЕХ ПРОЕКТАХ ПОЛЬЗОВАТЕЛЬ ДОЛЖЕН САМ РЕШАТЬ ВЫЙТИ ИЗ ПРОГРАММЫ ИЛИ ПРОДОЛЖИТЬ ВВОД ДАННЫХ. В КАЖДОМ ЗАДАНИИ НЕОБХОДИМО СОЗДАТЬ ДИАГРАММУ КЛАССОВ НА ЯЗЫКЕ UML ДЛЯ ОТОБРАЖЕНИЯ ВСЕХ СОЗДАННЫХ КЛАССОВ И СВЯЗЕЙ МЕЖДУ НИМИ.

ВАРИАНТЫ ИНДИВИДУАЛЬНЫХ ЗАДАНИЙ К ЛАБОРАТОРНОЙ РАБОТЕ №1:

1. Описать класс «Точка» с полями, описывающими координаты и цвет точки. Реализовать конструктор класса, методы получения значений полей, методы изменения значений полей, метод отображения координат точки на экране и метод перемещения точки на плоскости. Описать класс «Отрезок» с полями координаты и цвет. Реализовать методы получения значения полей, методы изменения значений полей, метод отображение координат отрезка на экране и перемещения отрезка. Используя классы «Точка» и «Отрезок», создать класс «Квадрат». Предусмотреть следующие действия с объектами созданных классов: создание объектов, отображение координат объектов на экране, перемещение объектов и изменение параметров объектов (цвет,

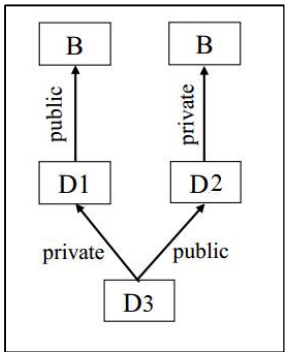
размер, координаты и т.д.). Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования. Предусмотреть метод для записи полученных данных в файл. **При необходимости самостоятельно добавить классы для реализации множественного наследования.**

2. Разработать классы «Двигатель», «Кузов», «Цвет», «Автомобиль», «Военный автомобиль», «Транспортное средство». Содержимое классов продумать самостоятельно (в каждом классе должно быть минимум два поля и два метода для работы с этими полями). Определить и реализовать связи между классами. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования. Предусмотреть метод для записи полученных данных в файл. Предусмотреть следующие действия с объектами созданных классов: создание объектов, отображение данных на экране, изменение параметров объектов, поиск необходимого объекта по параметрам. **При необходимости самостоятельно добавить классы для реализации множественного наследования.**

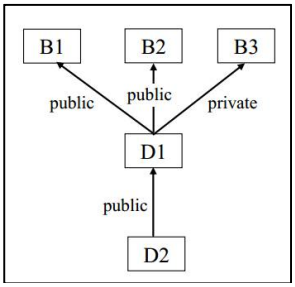
3. Имеется два класса: «Данные о работнике» (поля класса: фамилия, массив зарплат за квартал), «Налоговые данные» (поля класса: процент подоходного налога). Разработать класс «Платежная форма» для вывода итоговых данных (данных о работке и о его налоговых вычетах). **При необходимости самостоятельно добавить классы для реализации множественного наследования.**

4. Разработать набор классов «Город», «Магазин», «Банк», «Покупатель». Реализовать необходимые связи между классами. **При необходимости самостоятельно добавить классы для реализации множественного наследования.**

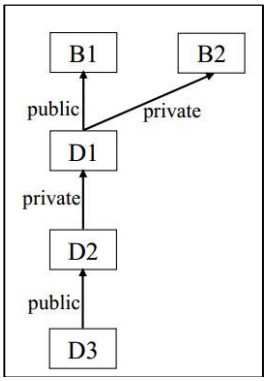
5. Построить иерархию классов согласно схеме наследования, приведенной на рисунке ниже, по любой предметной области. Каждый класс должен содержать необходимые конструкторы и методы работы с полями классов. Функция *main()* должна иллюстрировать работу с массивами объектов всех созданных классов. При необходимости атрибуты доступа при наследовании можно изменять. **При необходимости самостоятельно добавить классы для реализации множественного наследования.**



6. Построить иерархию классов согласно схеме наследования, приведенной на рисунке ниже, по любой предметной области. Каждый класс должен содержать необходимые конструкторы и методы работы с полями классов. Функция *main()* должна иллюстрировать работу с массивами объектов всех созданных классов. При необходимости атрибуты доступа при наследовании можно изменять. **При необходимости самостоятельно добавить классы для реализации множественного наследования.**

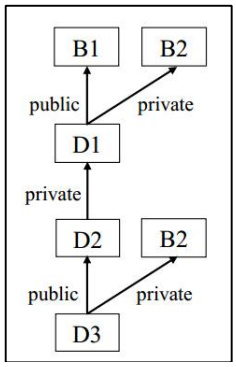


7. Построить иерархию классов согласно схеме наследования, приведенной на рисунке ниже, по любой предметной области. Каждый класс должен содержать необходимые конструкторы и методы работы с полями классов. Функция *main()* должна иллюстрировать работу с массивами объектов всех созданных классов. При необходимости атрибуты доступа при наследовании можно изменять. **При необходимости самостоятельно добавить классы для реализации множественного наследования.**

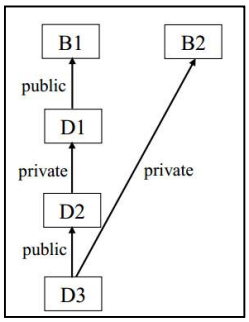


8. Построить иерархию классов согласно схеме наследования, приведенной на рисунке ниже, по любой предметной области. Каждый класс

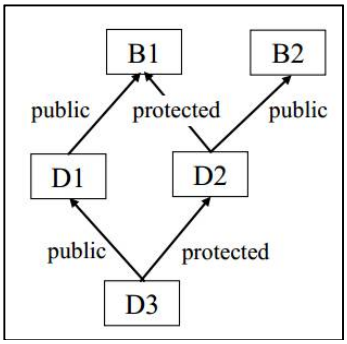
должен содержать необходимые конструкторы и методы работы с полями классов. Функция *main()* должна иллюстрировать работу с массивами объектов всех созданных классов. При необходимости атрибуты доступа при наследовании можно изменять. **При необходимости самостоятельно добавить классы для реализации множественного наследования.**



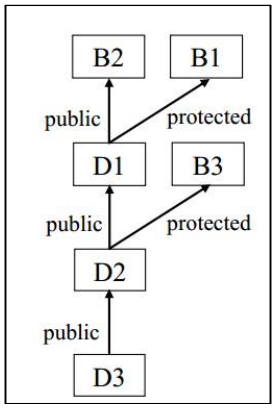
9. Построить иерархию классов согласно схеме наследования, приведенной на рисунке ниже, по любой предметной области. Каждый класс должен содержать необходимые конструкторы и методы работы с полями классов. Функция *main()* должна иллюстрировать работу с массивами объектов всех созданных классов. При необходимости атрибуты доступа при наследовании можно изменять. **При необходимости самостоятельно добавить классы для реализации множественного наследования.**



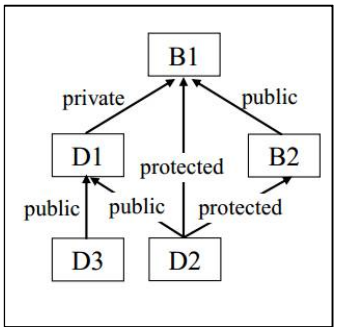
10. Построить иерархию классов согласно схеме наследования, приведенной на рисунке ниже, по любой предметной области. Каждый класс должен содержать необходимые конструкторы и методы работы с полями классов. Функция *main()* должна иллюстрировать работу с массивами объектов всех созданных классов. При необходимости атрибуты доступа при наследовании можно изменять. **При необходимости самостоятельно добавить классы для реализации множественного наследования.**



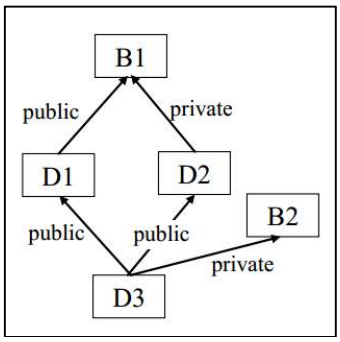
11. Построить иерархию классов согласно схеме наследования, приведенной на рисунке ниже, по любой предметной области. Каждый класс должен содержать необходимые конструкторы и методы работы с полями классов. Функция *main()* должна иллюстрировать работу с массивами объектов всех созданных классов. При необходимости атрибуты доступа при наследовании можно изменять. **При необходимости самостоятельно добавить классы для реализации множественного наследования.**



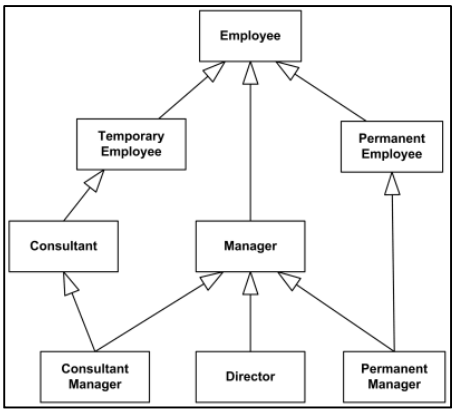
12. Построить иерархию классов согласно схеме наследования, приведенной на рисунке ниже, по любой предметной области. Каждый класс должен содержать необходимые конструкторы и методы работы с полями классов. Функция *main()* должна иллюстрировать работу с массивами объектов всех созданных классов. При необходимости атрибуты доступа при наследовании можно изменять. **При необходимости самостоятельно добавить классы для реализации множественного наследования.**



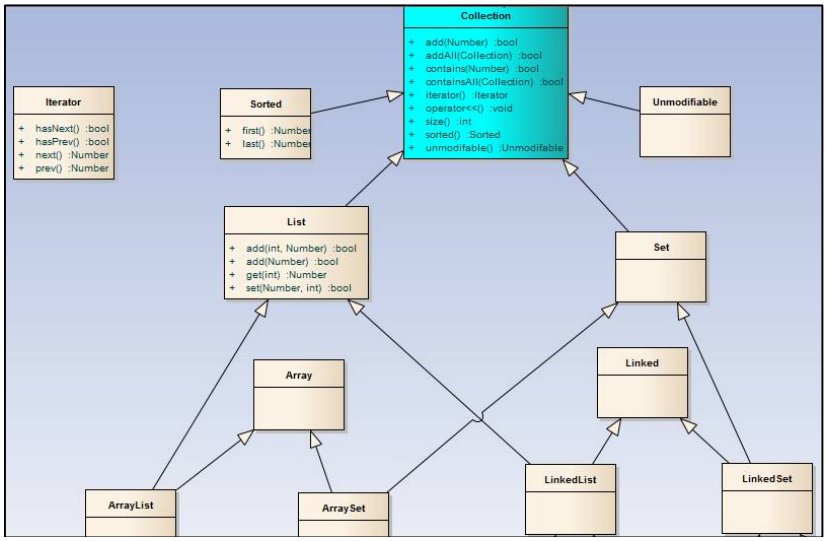
13. Построить иерархию классов согласно схеме наследования, приведенной на рисунке ниже, по любой предметной области. Каждый класс должен содержать необходимые конструкторы и методы работы с полями классов. Функция *main()* должна иллюстрировать работу с массивами объектов всех созданных классов. При необходимости атрибуты доступа при наследовании можно изменять. **При необходимости самостоятельно добавить классы для реализации множественного наследования.**



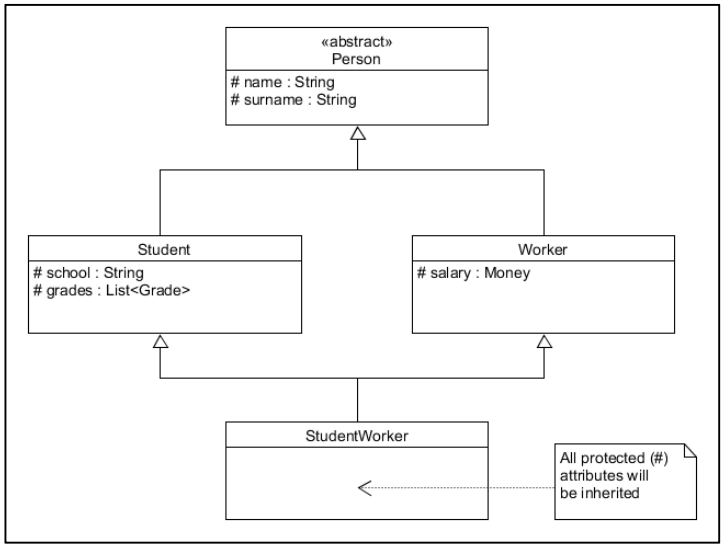
14. Построить иерархию классов согласно схеме наследования, приведенной на рисунке ниже по предметной области «HR-менеджмент». Каждый класс должен содержать необходимые конструкторы и методы работы с полями классов. Функция *main()* должна иллюстрировать работу с массивами объектов всех созданных классов. При необходимости атрибуты доступа при наследовании можно изменять. **При необходимости самостоятельно добавить классы для реализации множественного наследования.**



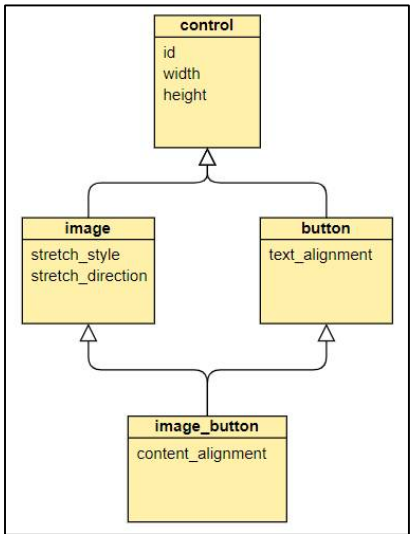
15. Построить иерархию классов согласно схеме наследования, приведенной на рисунке ниже по предметной области «Коллекции». Каждый класс должен содержать необходимые конструкторы и методы работы с полями классов. Функция *main()* должна иллюстрировать работу с массивами объектов всех созданных классов. **При необходимости самостоятельно добавить классы для реализации множественного наследования.**



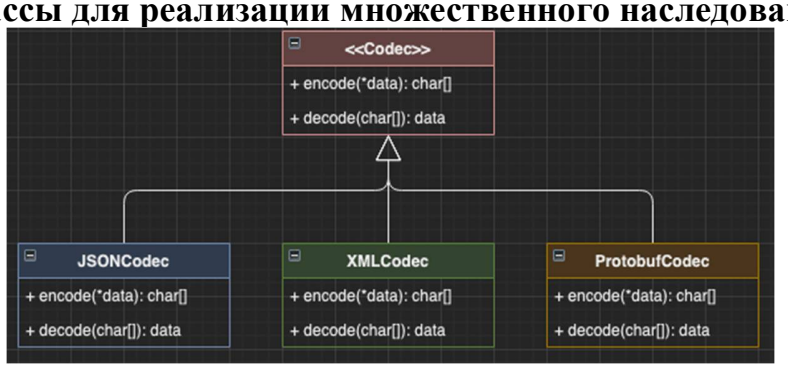
16. Построить иерархию классов согласно схеме наследования, приведенной на рисунке ниже по предметной области «Высшее учебное заведение». Каждый класс должен содержать необходимые конструкторы и методы работы с полями классов. Функция *main()* должна иллюстрировать работу с массивами объектов всех созданных классов. **При необходимости самостоятельно добавить классы для реализации множественного наследования.**



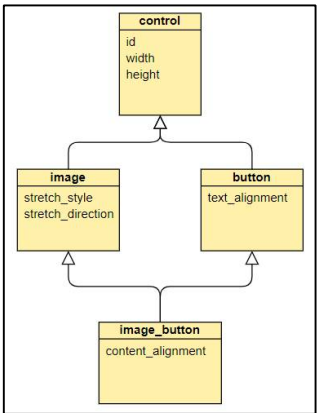
17. Построить иерархию классов согласно схеме наследования, приведенной на рисунке ниже по предметной области «Графический пользовательский интерфейс». Каждый класс должен содержать необходимые конструкторы и методы работы с полями классов. Функция *main()* должна иллюстрировать работу с массивами объектов всех созданных классов. **При необходимости самостоятельно добавить классы для реализации множественного наследования.**



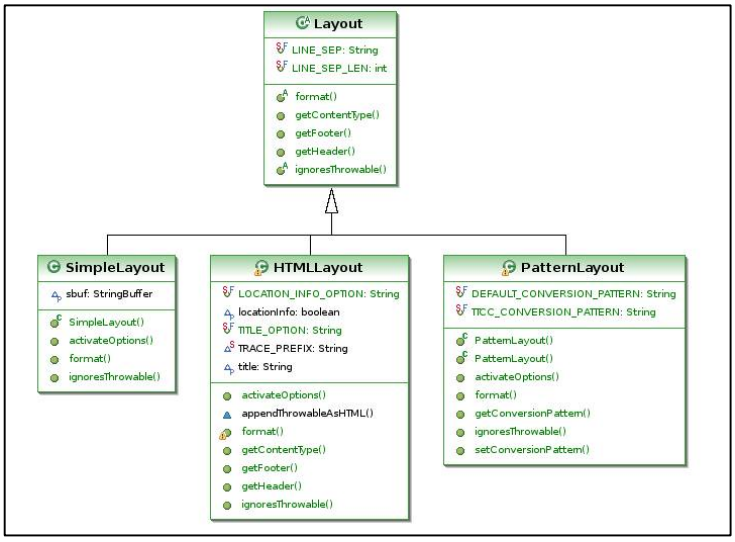
18. Построить иерархию классов согласно схеме наследования, приведенной на рисунке ниже по предметной области «Кодеки». Каждый класс должен содержать необходимые конструкторы и методы работы с полями классов. Функция *main()* должна иллюстрировать работу с массивами объектов всех созданных классов. **При необходимости самостоятельно добавить классы для реализации множественного наследования.**



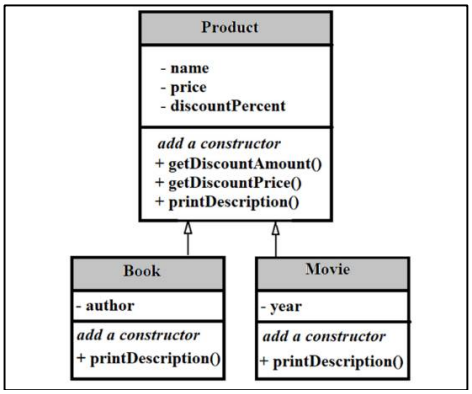
19. Построить иерархию классов согласно схеме наследования, приведенной на рисунке ниже по предметной области «Книжный магазин». Каждый класс должен содержать необходимые конструкторы и методы работы с полями классов. Функция *main()* должна иллюстрировать работу с массивами объектов всех созданных классов. **При необходимости самостоятельно добавить классы для реализации множественного наследования.**



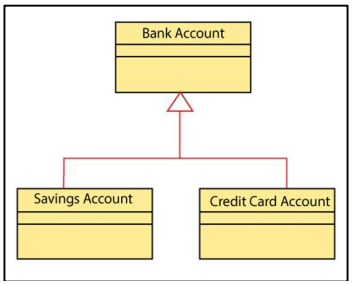
20. Построить иерархию классов согласно схеме наследования, приведенной на рисунке ниже по предметной области «Позиционирование элементов пользовательского интерфейса». Каждый класс должен содержать необходимые конструкторы и методы работы с полями классов. Функция *main()* должна иллюстрировать работу с массивами объектов всех созданных классов. **При необходимости самостоятельно добавить классы для реализации множественного наследования.**



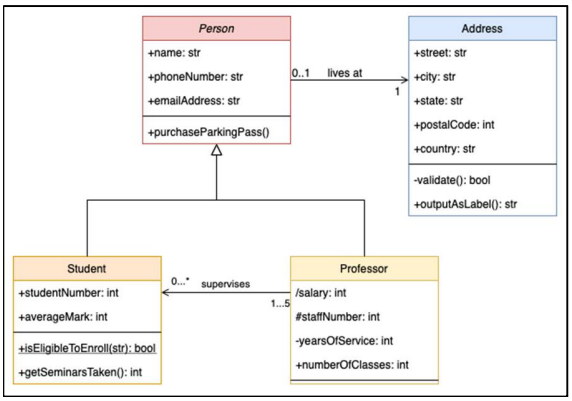
21. Построить иерархию классов согласно схеме наследования, приведенной на рисунке ниже по предметной области «Интернет магазин». Каждый класс должен содержать необходимые конструкторы и методы работы с полями классов. Функция *main()* должна иллюстрировать работу с массивами объектов всех созданных классов. **При необходимости самостоятельно добавить классы для реализации множественного наследования.**



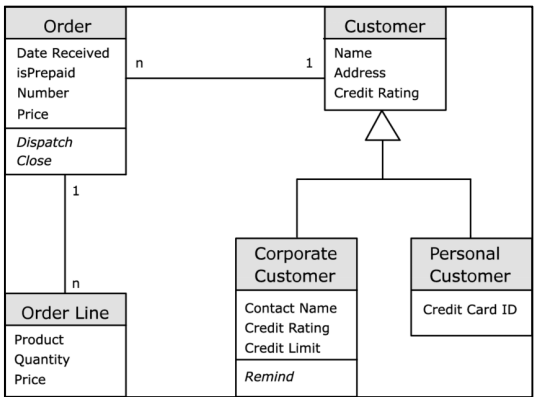
22. Построить иерархию классов согласно схеме наследования, приведенной на рисунке ниже по предметной области «Обслуживание клиентов банка». Каждый класс должен содержать необходимые конструкторы и методы работы с полями классов. Функция *main()* должна иллюстрировать работу с массивами объектов всех созданных классов. **При необходимости самостоятельно добавить классы для реализации множественного наследования.**



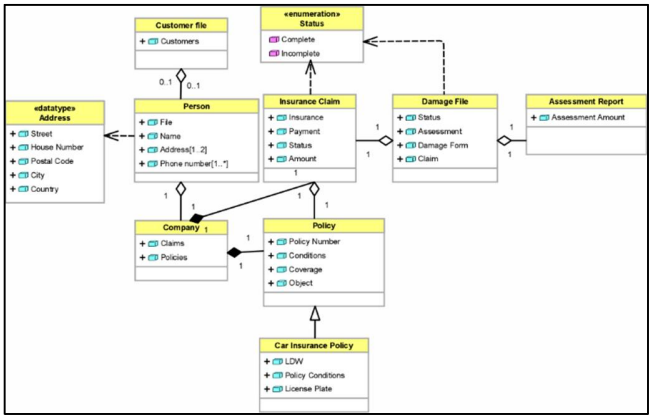
23. Построить иерархию классов согласно схеме наследования, приведенной на рисунке ниже по предметной области «HR-менеджмент». Каждый класс должен содержать необходимые конструкторы и методы работы с полями классов. Функция *main()* должна иллюстрировать работу с массивами объектов всех созданных классов. **При необходимости самостоятельно добавить классы для реализации множественного наследования.**



24. Построить иерархию классов согласно схеме наследования, приведенной на рисунке ниже по предметной области «Система учета заказов». Каждый класс должен содержать необходимые конструкторы и методы работы с полями классов. Функция *main()* должна иллюстрировать работу с массивами объектов всех созданных классов. **При необходимости самостоятельно добавить классы для реализации множественного наследования.**

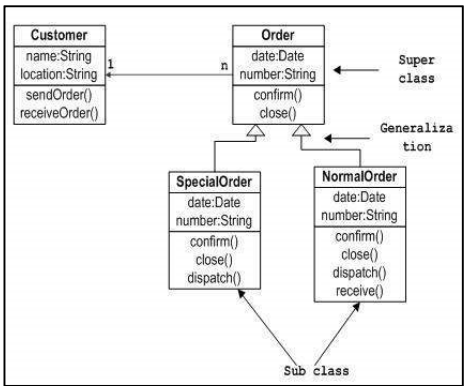


25. Построить иерархию классов согласно схеме наследования, приведенной на рисунке ниже по предметной области «Система учета заявок». Каждый класс должен содержать необходимые конструкторы и методы работы с полями классов. Функция *main()* должна иллюстрировать работу с массивами объектов всех созданных классов. **При необходимости самостоятельно добавить классы для реализации множественного наследования.**

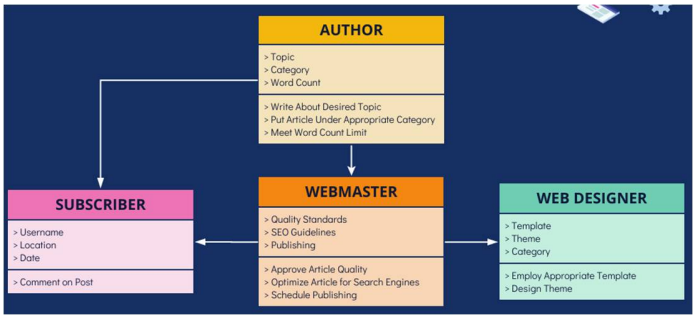


26. Построить иерархию классов согласно схеме наследования, приведенной на рисунке ниже по предметной области «Система учета заказов». Каждый класс должен содержать необходимые конструкторы и методы работы с полями классов. Функция *main()* должна иллюстрировать работу с массивами объектов всех созданных классов. **При необходимости**

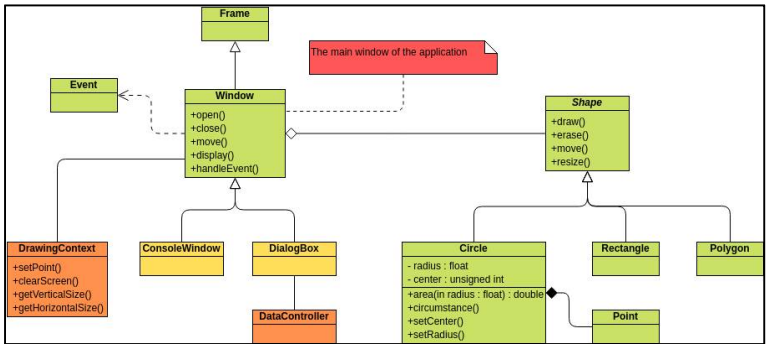
самостоятельно добавить классы для реализации множественного наследования.



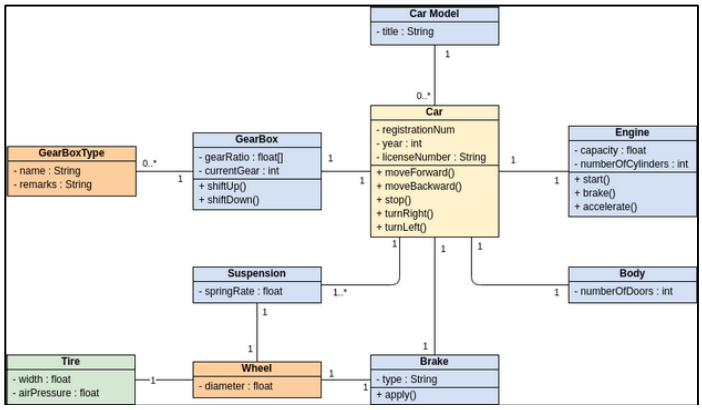
27. Построить иерархию классов согласно схеме наследования, приведенной на рисунке ниже по предметной области «Web-разработка». Каждый класс должен содержать необходимые конструкторы и методы работы с полями классов. Функция *main()* должна иллюстрировать работу с массивами объектов всех созданных классов. **При необходимости самостоятельно добавить классы для реализации множественного наследования.**



28. Построить иерархию классов согласно схеме наследования, приведенной на рисунке ниже по предметной области «Графический редактор». Каждый класс должен содержать необходимые конструкторы и методы работы с полями классов. Функция *main()* должна иллюстрировать работу с массивами объектов всех созданных классов. **При необходимости самостоятельно добавить классы для реализации множественного наследования.**



29. Построить иерархию классов согласно схеме наследования, приведенной на рисунке ниже по предметной области «Станция технического обслуживания автомобилей». Каждый класс должен содержать необходимые конструкторы и методы работы с полями классов. Функция *main()* должна иллюстрировать работу с массивами объектов всех созданных классов. **При необходимости самостоятельно добавить классы для реализации множественного наследования.**



30. Построить иерархию классов согласно схеме наследования, приведенной на рисунке ниже по предметной области «Распределение бюджета проектов». Каждый класс должен содержать необходимые конструкторы и методы работы с полями классов. Функция *main()* должна иллюстрировать работу с массивами объектов всех созданных классов. **При необходимости самостоятельно добавить классы для реализации множественного наследования.**