Софийски университет „Климент Охридски“
Факултет по математика и информатика

ПРОЕКТ
ОБЕКТНО-ОРИЕНТИРАНО ПРОГРАМИРАНЕ

летен семестър 2023/2024

Проект №7
БАЗА ОТ ДАННИ

Изготвил: Стоян Стоянов Иванов
Специалност: Информатика, 2 курс
Факултетен номер: 9MI0400132

Ръководител на курса:
доц. д-р Петър Армянов

юни 2024 г.

СЪДЪРЖАНИЕ

1. Увод

1.1. Описание и идея на проекта

Основната идея на проекта е разработката на програма, чиято цел е да управлява данни в дадена база данни. Базата данни се състои от серии от таблици, като всяка таблица е записана в собствен файл. Тази база е записана в главен файл, наречен каталог, съдържащ списък от таблиците. Всяка отделна таблица е изградена от масиви от данни, които имат връзка помежду си. Базата е организирана така, че да използва записи и полета, за да може данните да се използват по-лесно и по-ефективно.

1.2. Цел и задачи на разработката

Целта на този проект е да се реализира система, управляваща база данни. Нужно е да се осъществят методите: внасяне на таблица в базата данни, предоставяне на списък с имената на всички таблици, извеждане на специфичен тип информация за определена таблица, показване на редове на дадена таблица, записване на съдържанието на таблица в отделен файл, посочване на редове на дадена таблица, съдържаща търсена от потребителя информация, създаване на нова таблица, обхващаща упоменати от потребителя данни, които са включени в друга таблица, разширяване на специфична таблица, актуализиране на дадена таблица с въведена от потребителя информация, вмъкване и премахване на нови редове на определени таблици.

Една от основните задачи на базата данни е съхранението на големи количества данни в структуриран формат и предоставянето на механизми за съхранение и организация на тези данни, позволяващи бърз достъп и ефективност при търсенето и обработката им. Осъществява се контрол на достъпа до данните, защита на данните от неправомерен достъп и осигуряване на цялостност и съответствие на данните. Друга важна задача е предоставянето на възможност за извличането на специфична информация от базата данни за анализ, отчетност и вземане на решения. Това позволява откриването на тенденции, генерирането на отчети и вземането на решения на базата на наличните данни.

1.3. Структура на документацията

Документацията на проекта се състои от файловете, необходими за компилирането на проекта (.cpp и .hpp), каталог-образец, заедно с примерните таблици, съдържащи се в него, и текстовият файл, включващ документацията на проекта.

Документацията започва с анализ на задачата и подходът за решение. След това е представено кратко описание на класовете, които са създадени за решение на задачата. Завършва се с идеи за бъдещи подобрения.

2. Преглед на предметната област

2.1. Основни дефиниции, концепции и алгоритми, които ще бъдат използвани

Програмата използва градивните елементи на обектно-ориентираното програмиране за реализация на поставените задачи. Включени са класове, които са тип данни, осигуряващи рамка за създаване на различни обекти. Данните и функциите на класа се наричат членове. Обектите от своя страна представляват инстанции на дадени класове. Те дефинират специфични свойства и поведение за изпълнение на кода. Терминът "инстанция" се отнася до връзката между обекта и неговия клас. Програмата се възползва и от понятието метод – това е функция, определяща поведението на даден клас или на обекти в него. Методите са специфични за класа и конкретизират начина, по който обектът може да използва или модифицира данни. Друга важна част е конструкторът[1] - специален вид метод, използващ се за автоматично създаване на обектите на класа. Съществуват и специални видове конструктори, като например конструктори за копиране, които създават специфични типове обекти или използват определени правила за създаване на обектите. Други важни понятия, фундаментални за правилната работа на тази обектно-ориентирана програма, са селекторите и мутаторите, които се използват съответно за вземането на стойност и промяната на член данна. Ценен принцип е и абстракцията. Чрез нея крайният потребител вижда само интерфейс, а не целия вътрешен код. Оползотворяват се и идеите за енкапсулиране, наследяване и полиморфизъм. Чрез енкапсулирането се защитава определена информация в рамките на даден клас от останалата част от кода. Използвайки наследяване, се обобщават няколко класа под един общ знаменател. Полиморфизмът се отнася до способността на различните подкласове в йерархията да реагират на една и съща команда по свой собствен начин.

2.2. Дефиниране на проблеми и сложност на поставената задача

Проблемът, който трябва да бъде решен, е създаването на база данни и имплементирането на методи за обработка на данните в нея. Програмата трябва да поддържа определен брой операции, които авторът е разделил на четири типа:

- Операции тип „за интерфейс“: отваряне на база данни и правилно прочитане на информацията от няколко файла – един главен, съдържащ имената на отделните таблици, и няколко други файла, съдържащи данните на отделните таблици; запазване на промените върху таблиците, направени от потребителя; запазване на отделна таблица по указан от потребителя път; представяне на списък с различните операции, които могат да бъдат извършени; затваряне на програмата;
- Операции тип „за база“: представяне на списък с имената на таблиците в базата данни; вмъкване на нова таблица в базата данни по посочено име на файла; извеждане на информация по име за типовете данни на определена от потребителя таблица като е реализиран диалогов режим; запис на съдържанието по име на специфична таблица, форматирано по специфичен начин;
- Операции тип „за таблица“: извеждане на всички редове на указана от потребителя таблица по колона, съдържание на клетка и име като информацията се представя по страници; създаване на нова таблица, обхващаща определени данни, които са включени в друга таблица

[1] Brian Overland, “C++ Without Fear”

по колони и съдържание на клетка; добавяне на нова колона с най-голям номер в дадена таблица като за всички съществуващи редове от таблицата, стойността на тази колона трябва да е празна; актуализация на таблица по колони и съдържание на клетки; изтриване на ред на таблица, съдържаща нежелана от потребителя информация; вмъкване на нов ред в определена таблица с въведени от потребителя стойности.

- Операции тип „за допълнение“: валидации за вектори от низове, вектори от цели числа и вектори от дробни числа; валидация за съдържание на клетки на таблици; добавяне и премахване на клетка към ред; добавяне и премахване на ред към таблица; добавяне на таблица към базата; транспониране на таблица за редови изглед; запис на празни стойности за всички съществуващи редове в таблицата; проверка за валидност на името на таблицата и наличие на дубликати.

Сложността на задачата би могла да бъде разделена на няколко аспекта: работа с файлова система, където се чете информация от файлове и се записва обратно във файловете или се създават нови файлове, обработка на данните – търсене, филтриране и модифициране, и управление на потребителския интерфейс. Имплементацията на всички операции и съответната валидация на данните и командите може да бъде предизвикателна. Възможни са проблеми като неправилно прочитане на данните, препълване на паметта, грешки при запис и четене от файлове, грешки в потребителския интерфейс и др. За успешно решаване на задачата е важно да се проектира подходяща архитектура на системата, с ясно разделение на отговорностите и добра организация на данните. Освен това, е необходимо да се изпълняват подходящи проверки и валидации на данните, за да се гарантира коректността на операциите и защитата на данните.

2.3. Подходи, методи за решаване на поставените проблемите.

За решаването на поставения проблеми в програмата за база данни и обработка на данни, може да се използват различни подходи и методи. Най-напред ще се използва обектно-ориентиран дизайн, както е споменато по-горе. Прилагането на ООП принципите ще спомогне за създаване на по-структуриран и модулен код, който е по-лесен за разбиране, поддръжка и разширение. За четене и запис на данни от и към файлове, ще се използва стандартния вход/изход на езика C++. Важно е да се отбележи, че трябва да се гарантира правилното отваряне, затваряне и обработка на файловете, както и да се направят съответни проверки за грешки при работа с файловата система. За съхранение на данните в програмата и обработка на таблици, може да се използват структури като масиви, списъци или дървета, но авторът е избрал да използва само масиви. Поставя се акцент върху организацията на данните – нужно е да се осигури ефективен достъп и търсене в тях. За гарантиране на коректността на данните, ще се приложат различни методи за валидация. От голямо значение е да се правят проверки за възможни грешки при въвеждане на данни от потребителя и да се предпазва програмата от нежелани операции и неконсистентни състояния. Трябва да се проектира потребителски интерфейс, който е интуитивен и лесен за използване. И не на последно място, за осигуряване на качеството на програмата, от изключителна важност да се извършват систематични тестове и проверки. Отстраняването на грешки трябва да се извършва внимателно и систематично, като се анализират причините за грешките и се предприемат подходящи действия за тяхното отстраняване.

3. Проектиране

3.1. Обща архитектура – ООП дизайн

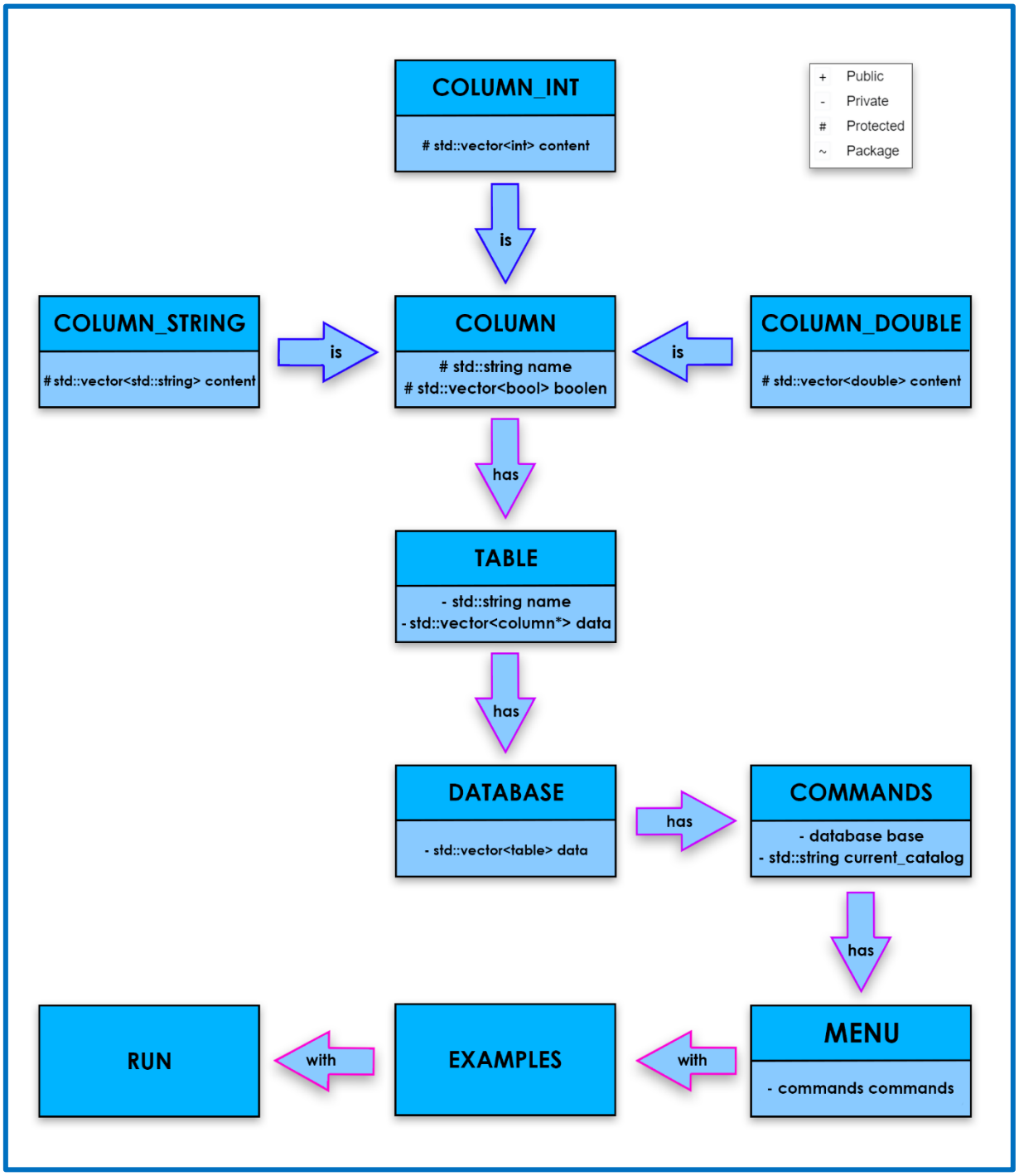
Основите на програмата се полагат с четирите основни класа *class Column, class Column_String*, *class Column_Int* и *class Column_Double (Фигура 2)*. Първият клас представлява чисто виртуален клас - колона, съдържащ член данни, характеристичен вектор и име. В него също така се съдържат селектори и мутатори, както и методи за принтиране върху екрана, записване на данните във файл и методи за промяна на данните – добавяне, смяна и изтриване на данни. Класовете *Column_String*, *Column_Int* и *Column_Double* наследяват класа Column и интерпретират гореспоменатите функции, но по свой определен начин в зависимост от задачата, която изпълняват. Класът *Column_String* има единствен вектор, съдържащ низове, от стандартната библиотека, и представлява една колона от определена таблица от базата данни. Тъй като този клас е наследник на клас Column, той също така притежава селектори и мутатори, методи за принтиране върху екрана, записване на данните във файл и методи за промяна на данните. Подобно на клас *Column_String*, класът *Column_Int* също има стандартен вектор, но този път данните му са от тип цели числа. Класът *Column_Double* пък съдържа данни от тип дробни числа. Също като *Column_String,* двата класа имат селектори и мутатори, методи за принтиране върху екрана, записване на данните във файл и методи за промяна на данните. Вече запознати с основата на системата база данни, може да се види как изглежда всяка таблица. Всеки файл, отговарящ на една таблица, е от тип *.bin* и изглежда по следния начин *(Фигура 1)*:

```
Number_Of_Columns,
Type_1, Name_1, Number_Of_Cells = K, Bool = 0|1 Data_1_1, ... Data_K
Type_2, Name_2, Number_Of_Cells = K, Bool = 0|1 Data_2_1, ... Data_K
...
Type_N, Name_N, Number_Of_Cells = K, Bool = 0|1 Data_N_1, ... Data_K
```

Фигура 1. Примерна таблица.

Тук *Number_Of_Columns* е броят на колоните в дадената таблица. След това е *Type_A*, където *A* е между *1* и *N*. За типовете на определените колони тук са валидни само ‘s’, ‘i’ и ‘d’, отговарящи съответно на низ, цяло число или дробно число. Следва името на всяка колона *Name_A* и броят елементи във всеки вектор, който за всяка колона е еднакъв и в случая е равен на *K*. Продължавайки по реда, преди всеки елемент на позиция преди него стои булева стойност – 0 или *1*, показваща дали елемента след нея е валиден или не.

3.2. Диаграми



Фигура 2. UML Диаграма.

4. Реализация и тестване

4.1. Реализация на класове

Както е посочено по-горе, основите на програмата се полагат с четирите основни класа *class Column, class Column_String, class Column_Int* и *class Column_Double (Фигура 2)*. *Class Column* е чисто виртуален клас - колона, съдържащ член данни характеристичен вектор и име. В него също така се съдържат селектори и мутатори, както и методи за принтиране върху екрана, записване на данните във файл и методи за промяна на данните – добавяне, смяна и изтриване на данни.

Класовете *Column_String, Column_Int* и *Column_Double* наследяват *class Column* и интерпретират гореспоменатите функции, но по свой определен начин в зависимост от задачата, която изпълняват. Класът *Column_String* има единствен вектор от стандартната библиотека, притежаващ низове, и представлява една колона от определена таблица от базата данни. От това, че този клас е наследник на клас *Column*, той също така съдържа в себе си селектори и мутатори, методи за принтиране върху екрана, записване на данните във файл и методи за промяна на данните. Подобно на клас *Column_String*, класът *Column_Int* също има вектор, но този път данните му са от тип цели числа. Класът *Column_Double* от своя страна съдържа данни от тип дробни числа. Както *Column_String,* двата класа има селектори и мутатори, методи за принтиране върху екрана, записване на данните във файл и методи за промяна на данните.

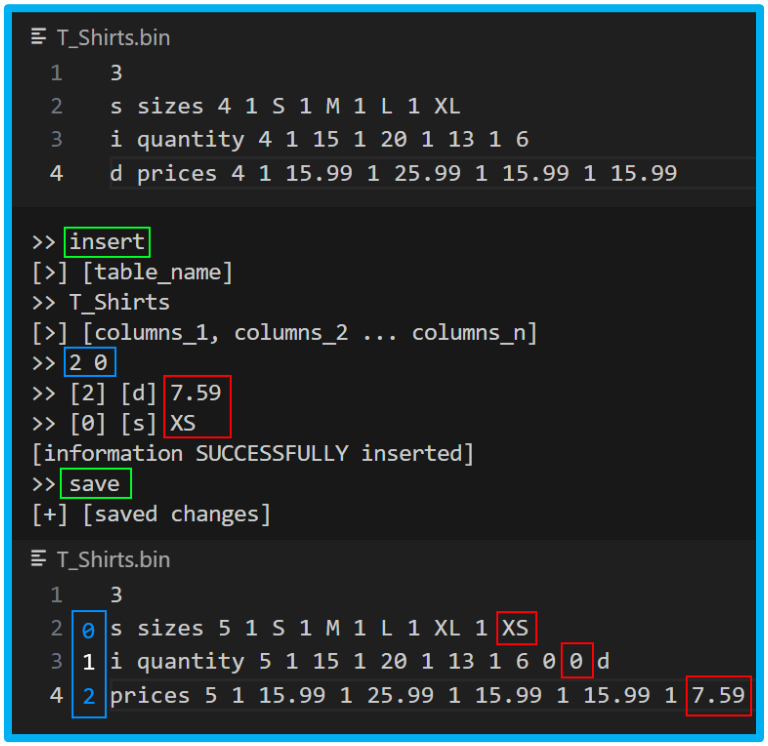
Следва *class Table*. Член-данните му са хетерогенният контейнер от указатели от клас *Column*, както и името на съответната таблица, за която отговаря. Притежава също мутатори и селектори, метод за принтиране, метод за запис на данните във файл, методи за намиране на различните типове на колоните, и проверки и функционалности за промяна, нужни на по-горни класове за това дали типът на колоната е низ, реално число или дробно число. В допълнение, съдържа функция, която трансформира дадена таблица, така че клетките й да са низове и след това я транспонира. Това е нужно, поради факта че няколко метода от по-горните класове изискват представяне по редове, въпреки че самите таблици в системата се обработват по колони.

Следващият клас е *class Database*. Той съдържа масив от *Table*, т.е. масив от таблици. Освен че се държи като вектор, притежава и мутатори, и селектори, притежава и голям набор от методи, нужни за правилното функциониране на програмата – внасяне на таблица в базата и актуализиране на каталога с новите данни, функция, представяща всички имена на съдържащите се в базата таблици, описващ метод за това колко и какви са клетките на дадена таблица, метод, който принтира на екрана редовете на определена таблица по страници, *export_base(),* който позволява на потребителя да създаде копие на таблица с ново име, функция за проверка дали таблица е в базата данни, връщайки индекса й, ако присъства, метод за изтриване на ред и не на последно място метод за внасяне на нов ред в таблица, базиран на първия ред в таблицата, тъй като той се приема винаги за верен.

Следващият клас е *class Commands*. Съдържа като член-данна един *Database* и името на текущия отворен за работа каталог. Притежава и двадесет метода. Заедно с последния клас от йерархията *class Menu,* който в себе си има един *class Commands*, позволяват работата с тази система от бази-данни.

По-долу се акцентира на всяка една функция поотделно:

- *open()* – [01] – въвежда се името на файла, ако не е открит такъв файл създава нов файл с празно съдържание;
- *save()* – [02] – запазва направените до този момент промени;
- *save_as()* – [03] – позволява на потребителя да запази определена таблица като се укаже и пътят към новата дестинация за запазването;
- *exit()* – [04] – пита потребителя дали иска да запази последно направените промени и затваря приложение.
- *close()* – [05] – затваря текущия каталог и запазва направените до този момент промени;
- *help()* – [06] – извежда меню с функциите на програмата;



Фигура 3. Пример

- *import()* – [07] – по указано от потребителя име на файл, вмъква в системата таблицата от дадения файл;
- *showtables()* – [08] – представя имената на всички таблица в системата;
- *describe()* – [09] – показва информация за типовете на колоните на дадена таблица;
- *print()* – [10] – извежда редовете на дадена таблица по страници;
- *xport()* – [11] – записва съдържанието на таблица в нов текстов файл;
- *select()* – [12] – по определена от потребителя колона, стойност на клетка в колона и име на таблица се извеждат редовете на таблицата по страници;
- *select_onto()* – [13] – потребителят въвежда ново име за таблица, име на таблица от системата, брой колони, номер на колоните и се създава нова таблица с първовъведеното име като новосъздадената таблица съдържа само колоните с номера, които се съдържат в таблицата от системата;
- *add_column()* – [14] – *(Фигура 3, Фигура 4)* – добавя нова колона с най-голям номер в дадена таблица като за всички съществуващи редове от таблицата, стойността на тази колона е празна;

- *update()* – [15] – по име на таблица, номер на колона и стойност на клетка променя таблицата по колона и стойност на клетка с избрана от потребителя нова информация;
- *delete()* – [16] – по име на таблица, по номер на колона и стойност на клетка изтрива реда на таблица, съдържащ определената стойност
- *insert()* – [17] – *(Фигура 3)* – по име и колона вмъква нов ред в дадена таблица;
- *credits()* – [18] – показва информация за автора на проекта;

Класът Commands също така притежава два частни метода, свързани с работата на *print()* и *select_onto():*

- *io_print_by_page()* – позволя изписването на конзолата на информация от таблица по редове и по страници;
- *io_print_by_page_special()* - позволя запазването на информация от таблица по редове;

Накрая са два .hpp файла, чрез които са нужни са пълната реализация на програма за управление на бази данни. В първия присъства функцията *run(),* която се използва за въвеждане на команди от потребителя и функцията *generate_examples(),* която генерира няколко примера, за да има възможност да се представи работата на програмата по подобаващ начин.

В допълнение, съществува и *template class Primitive_Pair*, който представлява много проста версия на std::pair от стандартната библиотека и спомага за работата на горните функции.

```
//class Commands:
std::cin >> table_name;

std::size_t index = this->base.is_table_in_database(table_name);
if(index == -1) { return; }

std::cin >> column_name >> column_type;

if(column_type == 's' || column_type == 'i' || column_type == 'd')
     this->base.add_column_base(index, column_name, column_type);

//class Database:
void database::add_column_base(...) {
    this->data[index].add_column_table(name, type); }

//class Table:
void table::add_column_table(...) {
     if(type == 's') {
        std::size_t size = this->data[0]->get_size_of_column();
        std::vector<std::string> to_add;

        for (std::size_t i = 0; i < size; ++i)
            to_add.push_back("NULL");

        std::vector<bool> to_add_bool;
        for (std::size_t i = 0; i < size; ++i)
            to_add_bool.push_back(0);

        column* temp = new column_string;
        // fill the column with the upper info;
        this->data.push_back(temp);
     }
     else if(type == 'i') ...
```

Фигура 4. Част от реализацията на *add_column().*

<u>Заб.</u> Премахнати са част от функциите, променливите и общите действия, за краткост.

4.2. Управление на паметта и алгоритми. Оптимизации.

Доброто управление на паметта (memory management) е ключово за това програмата да работи ефективно. Има два основни аспекта:

- разпределяне (memory allocation, когато част от паметта се заема при изпълнението на програмата);
- освобождаване (memory deallocation, на паметта, от която програмата вече не се нуждае).

Поради голямото наличие на еднакви действия, които трябва да се извършват при работа с програмата, е нужна правилна оптимизация. На *(Фигура 5)* са показани четири метода на *class Commands*, които се използват в почти всички гореописани функции.

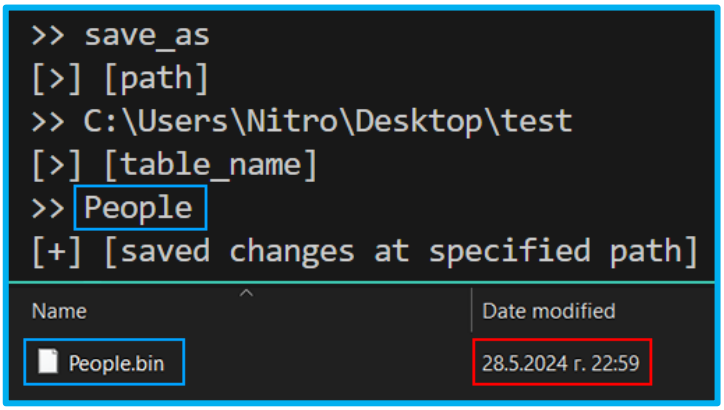
```
class commands
    /*[14]*/ void add_column();
    /*[15]*/ void update();
    /*[16]*/ void remove();
    /*[17]*/ void insert();
    /*[18]*/ void credits();

class database
    /*[14]*/ void add_column_base(std::size_t index, const std::string& info, const char type);
    /*[15]*/ void update_base(int index, int target_column, const std::string& target_info);
    /*[16]*/ void delete_base(int index, int search_column, const std::string& search_info);
    /*[17]*/ void insert_table(int index, int column_n, const std::string& input);

class table
    /*[14]*/ void add_column_table(const std::string& info, const char type);
    /*[15]*/ void update_table(int target_column, const std::string& target_value);
    /*[16]*/ void delete_table(int search_column, const std::string& search_value);
    /*[17]*/ void insert_table(int column_n, const std::string& input);

class column
    virtual void print_column() const = 0;
    /*[15]*/ virtual void update_column(const std::string& info) = 0;
    /*[16]*/ virtual int delete_column(const std::string& info) = 0;
             virtual void pop_row(int index) = 0;
    /*[17]*/ virtual void insert_column(const std::string& input) = 0;
```

Фигура 5. Оптимизация в class Compile.

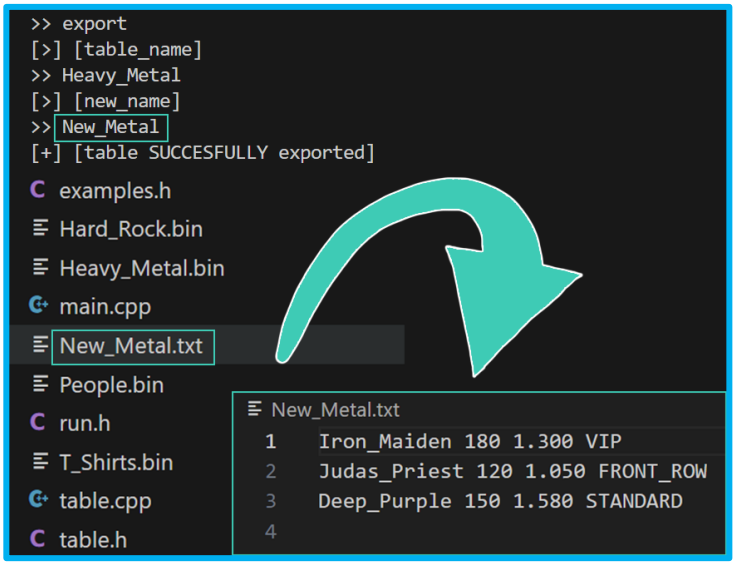
4.3. Планиране, описание и създаване на тестови сценарии.

Всичко, което може да бъде тествано, е тестов сценарий. Следователно всяка софтуерна функционалност, която е в процес на изпитване и може да бъде разделена на множество по-малки функционалности, може да се нарече „Тестов сценарий“. Тестовите сценарии помагат за оценка на софтуерното приложение според реалните ситуации. С други думи, тестването е процес по изпълняването на програмата, с цел да се открият бъгове, дефекти или грешки.

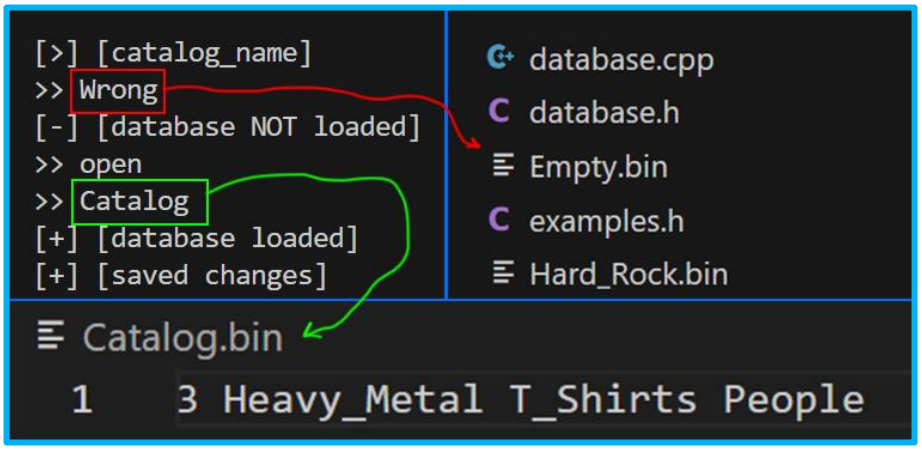
За да се покаже правилната работа на програмата, е нужно да се представят различни тестови сценарии.



Фигура 6. save_as()



Фигура 7. xport()



Фигура 8. open()

```
>> describe
[>] [table_name]
>> Heavy_Metal
[string][integer][double][string]
```

Фигура 9. describe()

People.bin

```
4                                   2
s firstnames 5 1 Velizar 1 Georgi 1 Stoyan 1 Dimitar 1 Petar
s secondnames 5 0 NULL 0 NULL 1 Stoyanov 1 Ivanov 1 Ognqnov
s thirdnames 5 1 Telbiiski 0 NULL 1 Ivanov 0 NULL 0 NULL
i ages 5 1 30 0 -24 0 -20 0 -21 1 20
```

```
>> delete
[>] [table_name]
>> People
[>] [search_column]
>> 2
[>] [search_value]
>> Ivanov
[+] [deleted SUCCESSFULLY infomation]
>> save
[+] [saved changes]
```

People.bin

```
4                                   2
s firstnames 4 1 Velizar 1 Georgi 1 Dimitar 1 Petar
s secondnames 4 0 NULL 0 NULL 1 Ivanov 1 Ognqnov
s thirdnames 4 1 Telbiiski 0 NULL 0 NULL 0 NULL
i ages 4 1 30 0 -24 0 -21 1 20
```

Фигура 10. delete()

```cpp
void column_int::write_into_file(std::ofstream& out) const
{
    if(out.is_open())
    {
        std::string name = this->name;
        out << name << ' ';
        std::size_t out_size = this->content.size();
        out << out_size << ' ';
        for (std::size_t i = 0; i < out_size; ++i)
        {
            out << this->boolen[i] << ' ';
            out << this->content[i] << ' ';
        }
    }
```

Фигура 11. column_int::write_into_file()

5. Заключение

5.1. Обобщение на изпълнението на началните цели

В заключение, програмата успешно използва градивните елементи на обектно-ориентираното програмиране, чрез които е разработена програмата, която успешно управлява данни в база данни. Тя осигурява възможност за извличане на специфична информация от базата данни, нужни за откриването на тенденции, генерирането на отчети и вземането на решения на базата на наличните данни.

5.2. Насоки за бъдещо развитие и усъвършенстване

За бъдещото развитие и усъвършенстване на проекта може да се разгледа добавянето на шаблони, да се оптимизират алгоритмите за обработка на данните за по-висока ефективност, да се подобри на интерфейса на програмата за по-лесна употреба и да се добавят още допълнителни механизми за сигурност и защита на данните, да се подобри представянето на всяка една отделна единица като *std::pair* от самата информация и булева стойност за нейната валидност. Вместо един характеристичен вектор, присъстващ във файла с основната информация за дадена таблица би могло да се създаде допълнителен файл, който да представлява булева матрица, показваща валидността на всяка клетка. Тези подобрения биха увеличили функционалността и удобството на програмата, отваряйки възможности за разширени използвания и подобрено потребителско изживяване.

Използвана литература

1. Brian Overland – “C++ Without Fear” (преведено).
2. ccpreference.com