Софийски университет „Св. Климент Охридски“

Факултет по математика и информатика

Проект

Бази от данни

курс Обектно-ориентирано програмиране

Информатика

летен семестър 2019/2020

Изготвил: Ивон Цонкова, Информатика, 5 гр, ФН: 45664

Съдържание

Увод

1.1. Описание и идея на проекта

Проектът реализира програма, която работи с прости бази от данни. Базите данни се състоят от серии от таблици, като всяка таблица е записана в собствен файл. Базата данни е записана в главен файл (каталог), които съдържа списък от таблиците в базата данни, като за всяка таблица е зададено име и файл, в който таблицата е записана.

1.2. Цел и задачи на разработката

Целта е да се създаде програма, която да може да чете, записва , модифицира и принтира таблици, състоящи се от колони от различен тип прост данни. Програмата трябва да поддържа и лист от всички заредени таблици, тяхното име и файлът от който са били прочетени.

Задачите на разработката включват:

- проектиране на общата архитектура използвайки принципите на ООП програмирането
- осигуряване на работа с прости типове данни, както и поддържането на "NULL" стойност за всяка от тях
- осигуряване на начин за създаване на колони, които да поддържат типовете данни
- разработване на начин за създаване на таблици състоящи се от колони, които поддържат различни типове данни
- имплементиране на различните функционалности, които да работят директно със стойности от таблица
- дизайн на метод за принтиране на таблиците на диалогов прозорец
- разработване на четене и записване на таблици в/от файл
- разработване на база от данни, която да съдържа всички заредени таблици
- дизайн на потребителския интерфейс
- тестване

1.3. Структура на документацията

Документацията се състои от:

- преглед на предметната област: дефиниране на проблема, изследване на функционалните изисквания, разглеждане на подходи и модели, които ще бъдат използвани в решението
- проектиране: обща архитектура и диаграми на структура
- реализация на класовете, оптимизация и тестване
- Заключение

Преглед на предметната област

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

Основната концепции върху която ще структурираме проектът е *полиморфизъм* - "Един интерфейс, множество от различни реализации" , който ще бъде използван за имплементирането на различните типове данни и съответните им колони. Друга концепция която ще използваме е *интерфейс*, или така наречения абстрактен клас, който в рамките на c++ се разбира като всеки клас, който има поне една чиста виртуална функция. Таблиците ще могат да бъдат принтирани на *диалогов прозорец*, като под това понятие ще разбираме графичен потребителски интерфейс предназначен да показва информация и (или) да получава отговор от потребителя.

2.2. Дефиниране на проблеми и сложност на поставената задача

Основният проблем с който се сблъскваме при реализацията на програмата е имплементирането на различните типове колони в една таблица. Сложността на задачата произлиза от фактът, че всички контейнери или n-мерни масиви , които c++ поддържа са създадени да съдържат в себе си обекти от един и същи тип. Един от основните методи за справяне с този проблем е използването на полиморфични контейнери. Така сложността на задачата се свежда до откриването на начин да се имплементира базов клас, който да обхваща и отговаря на различието при работа с трите типа данни, като в същото време предоставя общи методи, чрез които те да се достъпват през полиморфичния контейнер (тоест колоните).

Друг проблем е свързан с принтирането на заредените и модифицирани от програмата таблици. Едно от основните изисквания е таблиците да могат да се преглеждат по страници които се събират върху един екран на диалоговия прозорец, като това поставя проблемът как да се намерят размерите му и как да се форматира изхода.

2.3. Методи за решаване на поставените проблеми

Основните методи използвани за справяне с проблема при полиморфизма са използването на чисти виртуални функции и абстрактни класове. Чрез създаване на общ абстрактен интерфейсен клас, ние може да наследим и предефиниране общи методи за трите типа данни. Така ние ще можем да създадем полиморфичен контейнер от стойности (таблица от колони от различен тип), които ще изпълняват общи методи, които са имплементирани по различен начин според вида на типа колона (което зависи от типа на данните в нея).

Проблемът свързан с принтирането на заредените таблици ще решим чрез използването на 2 стандартни библиотеки: ***<iomanip>*** - за форматиране на поток и ***<sys/ioctl.h>*** , която ще предостави размерите на диалоговия прозорец (в случая терминалът).

2.4. Потребителски и функционални изисквания

Проектът не специфицира потребителски изисквания, за това водещ ще е основният принцип на енкапсулация в обектно ориентираното програмиране.

Проектиране

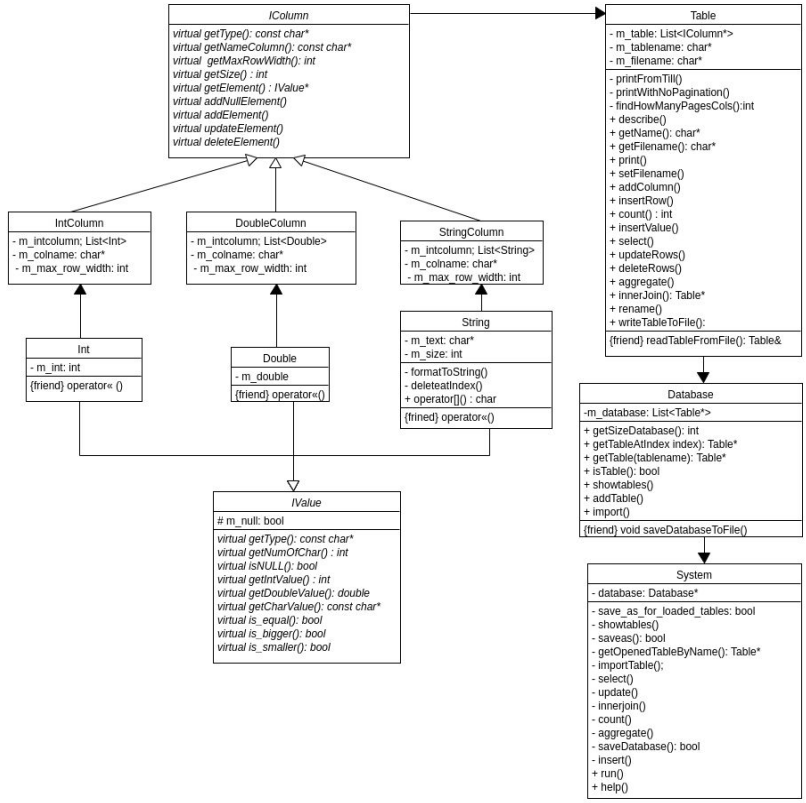
3.1. Обща архитектура и поведение

Архитектурата *(диаграма 1)* се състои от:

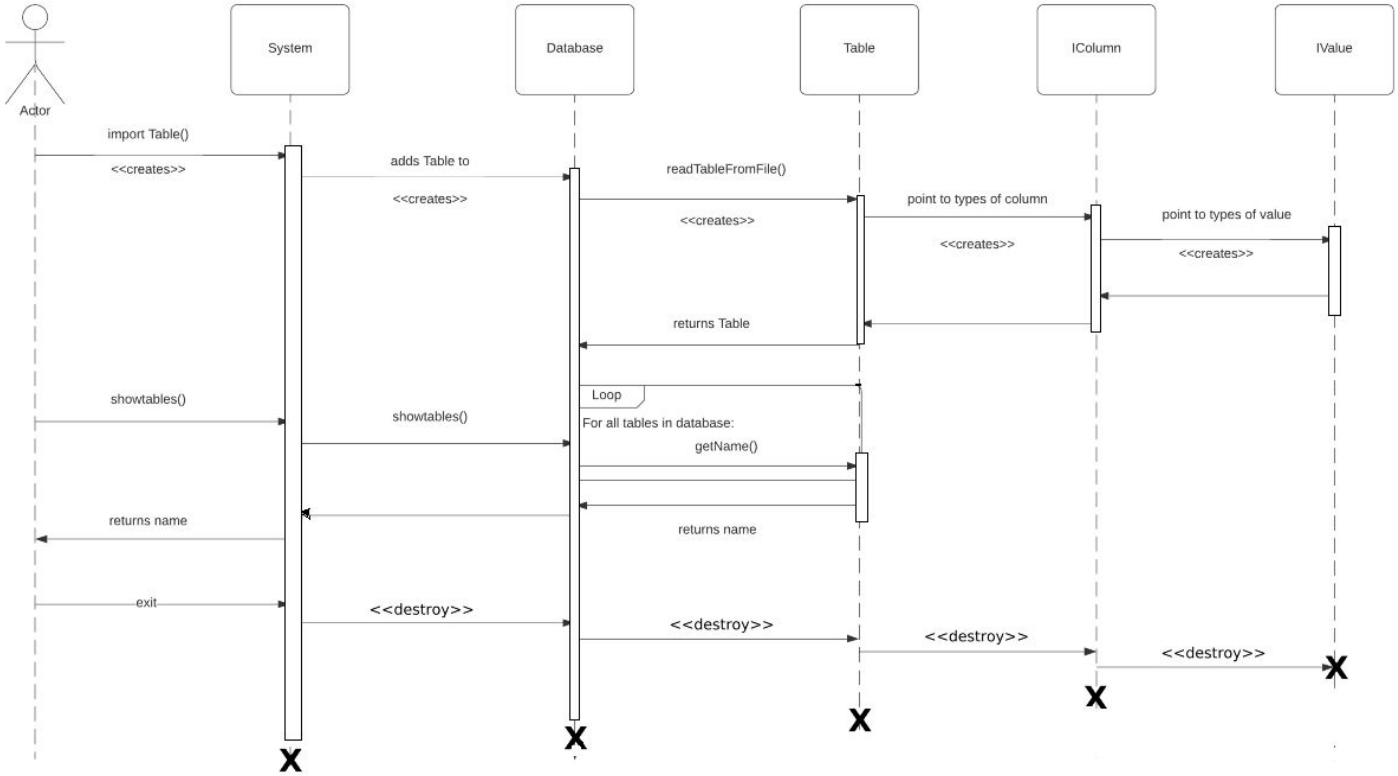
- Шаблонен клас **List** : имплементация на вектор, който може да съдържа различни типове обекти или типове данни
- абстрактен клас ***IValue***, който бива наследен от трите класа съответно **Int, Double, String**. Тези класове отговарят на трите типа данни, които една колона може да поддържа.
- Абстрактен клас ***IColumn***, който бива наследен от трите класа **IntColumn, DoubleColumn и StringColumn**. Наследяващите класове отговарят на трите вида колони, които една таблица може да съдържа в себе си. Съответно всеки от колоните се състои от лист от обекти на съответстващите им типове данни (тоест класовете **Int**, **Double, String**).
- Клас **Table**: представлява лист от указатели към обекти от абстрактния клас **IColumn**, като всеки от указателите сочи към обект от един от наследените класове **IntColumn/ DoubleColumn/ StringColumn**. Така на архитектурно ниво таблицата се явява полиморфичен контейнер.
- Клас **Database**: представлява лист от таблици.
- Клас **System**: състои се от обект от тип **Database**, в който ще се запазват всички заредени таблици. Отговаря за имплементацията на потребителския интерфейс.

Поведението на програмата *(диаграма 2)* се характеризира с :

- Потребителя работи със системата, която от своя страна записва, принтира, извиква и модифицира таблиците, които са заредени в базата данни.
- Потребителят няма пряк достъп до създаването на нови таблици.
- Класовете **IColumn** и **IValue** се достъпват при създаването на таблици, както и при всяка функционалност, която изисква модификация на стойности в дадена таблица.
- При извикването на командата close от потребителя, заредената база данни се изтрива, и клиентът може единствено да създаде нова база данни като зареди таблица от файл.
- При извикване на командата exit последователно се извикват деструкторите на класовете, като при интерфейсните класове първо се извика деструкторът на децата и след това на родителя.



Диаграма 1: UML диаграма на архитектурата



Диаграма 2: UML диаграма на поведение

4. Реализация и тестване

4.1. Реализация на класовете

- Клас **IValue** и **Int, Double, String**:
  - **IValue** е абстрактен клас с чисто виртуални функции,с единствена член данна, която ще означава дали дадената стойност е празната стойност или не.
  - Класовете **Int** и **Double** съдържат член данна със стойност от съответния прост тип данни, като надграждат над тях и позволяват да се означава дали дадена стойност е null - такава стойност ще разбираме като стойност от този тип, но за която не е изрично зададена стойност
  - Класът **String**- той съдържа член данна от указател към масив от вид char, като при разработката му се включва метод, който форматира масивът и осигурява правилното въвеждане на обект от този тип. Той изтрива кавичките и проверява за правилното включване на **** и кавички вътре в текста, както следва:

```
void String::formatToString()

{

    if ((m_text[0] == '"') && (m_text[m_size - 1] == '"'))

    {...}

    else

    {...}

    for (int i = 0; i < m_size; i++)

    {

        if (m_text[i] == '"')

        {

            if (m_text[i - 1] == '\\')

            {

                deleteatIndex(i - 1);

            }

            if (m_text[i - 1] != '\\')
```

```
                {

                    deleteatIndex(i);

                    cout << "Incorrect inclusion of \" inside the string.
The symbol will be truncated." << endl;

                }

            }

{...}
```

- Клас **IColumn** и **IntColumn**, **DoubleColumn** и **StringColumn**
  - IColumn - интерфейсен клас, с чисто виртуални методи
  - IntColum, DoubleColum и StringColumn- редефинира наследените виртуални методи, като осигурява, че добавената стойност ще е указател, който сочи към стойност от съответния тип, който характеризира колоната

```
void IntColumn::addNullElement()

{

       Int *new_null_int = new Int();

       m_intcolumn.addElement(*new_null_int);

       cout << "Ive added a null int" << endl;

}
```

// редефиниран метод, който осигурява добавянето на нов празен //елемент от тип Int към колона от тип IntColumn

- Клас **Table** - представлява полиморфичен контейнер от колони. Имплементира основните желани функционалности, които се характеризират с модификация на дадена таблица. Имплементирани са и методи за четене на таблица от файл и записването и във файл.

//пример за метод, който брои еднакви елементи в дадена колона

//вижда се използването на методът is_equal() деклариран в IValue, който сравнява две стойности :

```
int Table::count(int col_index, IValue *value) const

{

  int col_size = m_table.getElement(col_index)->getSize();
```

```
  int ctr = 0;

  for (int i = 0; i < col_size; i++)

  {

       IValue *row_value = m_table.getElement(col_index)->getElement(i);

       if (row_value->is_equal(value))

       ctr++;

  }

  cout << "Ctr is " << ctr << endl;

  return ctr;

}
```

- Клас **Database** - състои се от съвкупност от указатели към таблици и също така осигурява добавянето и извикването на някоя от заредените в програмата таблици

//метод, който добавя нова таблица в базата данни, като първо осигурява

//че не съществува таблица с това име

```
void Database::addTable(Table *new_table)

{

       if (isTable(new_table->getName()))

       {

       cout << "Tablename already exists!" << endl;

       return;

       }

       else

       {

       m_database.addElement(new_table);
```

```
        cout << "Successfully imported table with name " << new_table->getName()
<< " to current database!" << endl;

        return;

        }

}
```

- Клас **System** - клас, който имплементира потребителския интерфейс. Конструкторът за копиране и операторът за присвояване се изтриват, за да се забрани копирането на системата.

4.2. Управление на паметта и оптимизация

За правилното освобождаване на динамичната памет се грижат деструкторите на класът List, виртуалните деструктори и експлицитно дефинираните деструктори. След изход на програмата се освобождава заделената памет на командите, които потребителят въвежда. При заделянето на памет при въвеждане на файлови имена, имена на колони и прочие се дефинират глобални променливи, които да могат да бъдат променени при изменение на спецификите на системата.

4.3. Създаване на тестови сценарии

Планирането на тестови сценарии включва:

- Тестване на различни String входове, за да се тества правилното му форматиране
- Създаване на таблици с различна големина, вид колони и стойности, както и такива които инкорпорират “null” стойности, за да се тества правилното четене и модифициране на таблици
- Създаване на таблици с различни по големина имена, както и такива с различна ширина и височина, за да се тества разглеждането по страници на методът print()
- Тестване на потребителския интерфейс, въвеждане на грешни команди и зареждане на вече съществуващи таблици

Заключение

Програмата успешно може да прочита, модифицира и записва таблици с различни типове колони, както и да създава база от данни от заредените таблици. Насоки за бъдещото развитие може да включват инкорпориране на *Singleton Design Pattern[1]* за класът System, създаване на Command клас, който да олекоти интерфейса и инкорпориране на изключения.

Използвана литература:

[1] - “Singleton ” *Refactoring Guru*, © 2014-2020 Refactoring.Guru.
https://refactoring.guru/design-patterns/singleton