***Тема 10: Бази от данни***

*Георги Николаев Атанасов, ФН: 3MI0800092, I курс, I група*

*Github repo: https://github.com/nigosto/command\_line\_database*

***Описание и идея на проекта:***

Основната идея на проекта е да се реализира локална база от данни, която да може да бъде управляване през командния ред. Информацията за данните в базата трябва да се съхранява във файлове, като има един основен, който пази имената на таблиците в базата и файловете, в които е съхранена тяхната информация.

***Цел и задачи на разработката:***

Целта на проекта е да се реализира цялата функционалност, необходима за правилната и лесна работа на базата данни в диалогов режим. Главните задачи са да се изгради добра структура на проекта, основана на добрите ООП принципи, да се създаде работещ и ефективен формат, в който да бъде пазена информацията, да се изградят добри релации между всички файлове с информацията и да се реализира добър потребителски интерфейс през командния ред. Освен това по време на работа на програмата трябва да се създават временни файлове за възстановяване, които да пазят незаписаните промени в случай на ненавременно прекратяване на програмата. Не на последно място трябва да се имплементират всички необходими функции, които може да притежава една база данни и то по такъв начин, че те да могат да се извикват лесно и интуитивно през командния ред.

***Обща архитектура:***

Както бе споменато по-горе основната задача, свързана с изграждането на архитектурата, е да се спазят всички ООП принципи. За целта проектът е разделен на класове, като всеки клас отговаря за определена част от изпълнението на задачата. Освен това е имплементиран един основен клас, който да разпределя работата измежду останалите. За всеки от класовете е изпълнен принципа *RAII – Resource Acquisition is Initialization.* Използвани са няколко шаблона за дизайн: *Factory Design Pattern, Prototype Design Pattern* и *Singleton Design Pattern* , като за тях ще стане на въпрос по-долу. Основната архитектура е такава: имаме абстрактен клас *Column,* който е наследен от *IntColumn, DoubleColumn* и *StringColumn*, като причината те да са наследници, а не шаблонни класове е, че имат различна логика спрямо типа на данните относно метода *validate*, а *IntColumn* и *DoubleColumn* имат по един специфичен за тях метод – *valueAsNumber*. Освен това имаме клас *Table*, който съдържа хетерогенен контейнер от колони, и клас *Database*, който съдържа обикновен контейнер от таблици (понеже *Table* няма наследници), които съответно менежират колоните в съответната таблица и таблиците в базата данни. Поради това, че в програмата се извършват много операции върху обекти от тим *std::string*, е създаден и един помощен клас *StringManipulator*, който е статичен и се използва като *namespace*, който държи помощни функции, които да обработват стрингове. Освен това е реализиран интерфейсът *ISerializeable*, който се наследява от няколко от другите класове и съдържа методи, необходими за записването и четенето от файл на обект от съответния тип.

***Реализация:***

Програмата съдържа три абстракции: на колона, на таблица и на база данни.

1. *Column* – както споменах по-горе, това е абстрактен клас с 3 наследника - *IntColumn, DoubleColumn* и *StringColumn*. Всяка колона съдържа вектор от стойности и извършва определени операции върху тях. Освен това *Column* наследява *ISerializeable* понеже информацията за стойностите по колони се записва във файла на таблицата, която държи съответните колони. Поради това, че по-нататък ще искаме да имаме хетерогенен контейнер от колони, е използван *Prototype Design Pattern,* за да може контейнерът да притежава колоните, които му се подават. Понеже като създаваме колона от файл, можем да разберем каква колона точно да създадем само по нейния тип, е използван *Factory Design Pattern* под формата на функцията *createColumn*, която е фабрика за създаване на колона по подаден тип.
2. *Table* – съдържа хетерогенен контейнер от колони и извършва определени операции върху тях. Освен това този клас има точно определ файл, в който да записва своята информация, както и логика за създаване на файл за възстановяване, и затова не наследява *ISerializeable*. За *Table* е реализирана голяма четворка, поради факта, че съдържа хетерогенен контейнер. Освен това класът съдържа една приятелска функция, която извършва операцията *Inner Join* над две таблици и създава нова таблица, чието име и файл са комбинации от имената и файловете на двете изходни.
3. *Database* – съдържа контейнер от уникални таблици. Този клас също наследява *ISerializeable* и отделно има метод, който да създава файл за възстановяване.

За извършване на връзката между потребите и програма се грижи класът *DatabaseHandler*. Той обработва командите, подадени от потребителя, и се грижи за правилното управление на файловете (включително и тези за възстановяване), съдържащи информацията за базата данни. Класът *DatabaseHandler* е *Singleton*, понеже в цялата програма той е инстанциран само веднъж – във функцията *main* (която единственото, което прави, е да вика метода *readCommands* на тази инстанция). Освен това този клас е отговорен за писането на конзолата и за обработката на грешки, като той винаги успява да се възстанови от тях.

В допълнение към класовете, отговорни за бизнес логиката, съм създал и класът *StringManipulator*, който е статичен (не може да се създаде негова инстанция) и служи като *namespace* за някои функции, които обработват стрингове. Той се използва най-вече за обработка на командите, въведени от потребителя.

***Обобщение на изпълнението на началните цели:***

Във връзка с това, че цялата функционалност от изискванията е реализирана, програмата обработва без проблем командите на потребителя, и то по доста интуитивен начин, и това че класовете работят добре помежду си, смея да твърдя, че началните цели на проекта са изпълнени. Освен това е изградена добра архитектура от файлове, пазещи информацията, както и такъв файлов формат, който да работи най-добре с програмата. Не на последно място програмата обработва грешките по такъв начин, че програмата винаги е във валидно състояние, винаги успява да се възстанови от грешката и изкарва нужните съобщения, които да дадат информация на потребителя каква точно е причината за грешката.

***Насоки за бъдещо развитие и усъвършенстване:***

На първо място за усъвършенстване на проекта бих реализирал по общоприет формат на файловете – *JSON* или *XML*, така че едновременно файловете да станат по-лесно четими и да бъдат обработвани и от други програми, които поддържат тези формати. На второ място бих премахнал изцяло употребата на указатели и бих ги заменил с умни указатели, така че управлението на паметта да не зависи от мен и по този начин да намаля шанса за изтичане на памет. Ще завърша с това, че в бъдеще може да бъде създаден потребителски интерфейс, който да замени командния ред, като по този начин приложението би станало много по-лесно за употреба от обикновения потребител и много от излишните валидации за коректност биха изчезнали.