**Міністерство освіти і науки України**

**Національний авіаційний університет**

**Навчально-науковий інститут комп’ютерних**

**інформаційних технологій**



Курсова Робота

з дисципліни «Об’єктивно-орієнтоване програмування»

«Система тестування: створення тестів та тестування знань студентів»

Виконав студент:

групи ПІ-224Б

Ляшенко Б.М.

Перевірив викладач:

Дишлевий О.П.

Київ 2022

Історія контролю

|  |  |  |
| --- | --- | --- |
| **Дата** | **Опис** | **Автор** |
| 10.11.2022 | Система тестування: створення тестів та тестування знань студентів | Ляшенко Богдан Михайлович |

**Зміст**

1. **[Опис завдання.](#І_завдання)**

1. **[Опис Індивідуального завдання.](#ІІ_індивідуальне)**

1. **[Опис шарів проекту та загальна архітектура проекту.](#ІІІ_шари)**

1. **[Опис компонентів проекту.](#ІV_компоненти)**

1. **[Діаграма класів.](#V_діаграма)**

1. **[Опис особливостей реалізації.](#VІ_особливості)**

1. **[Описання функціоналу та використання.](#VІІ_функціонал)**

1. **[Використані джерела.](#VІІI_джерела)**

1. **[Код проекту.](https://github.com/cotuk3/CourseWork)**
2. **Скріншоти.**

**[Завдання](#І)**

1. В якості типу застосування обрати консольне застосування з командним рядком, як виняток, win forms, WPF.
2. Спроектувати та реалізувати систему класів, в основу якої покладено логічну структуру даних, наведену у варіанті, для накопичення та обробки даних домену відповідно варіанту курсової роботи.
3. Структура програмної системи курсового проекту:

Код програмної системи має складатись не менш як з трьох частин (окремих проектів) відповідно до багатошарової архітектури системи, де шарами архітектури є:

шар (рівень) доступу до даних (DAL), шар бізнес-логіки (BLL), шар представлення (інтерфейс програмної системи) (PL). Тип програмного модулю – DLL.

Шар доступу до даних організувати таким чином, щоб він забезпечував збереження даних у файлах за допомогою серіалізації або у БД із застосуванням Entity Framework.

Тип програмного модулю – DLL.

В шарі бізнес-логіки побудувати класи, що представляють об’єкти та дії над ними відповідно до предметної області, наприклад, читачі, книги та абонементи, читач може взяти книгу на свій абонемент при її наявності у сховищі та повернути її у заданий термін. Для створення наборів об’єктів предметної області (студенти, викладачі, книги, замовлення, страви, тощо) використовувати класи-узагальнені колекції. Всі операції бізнес-логіки використовують об’єктну модель даних. Якщо дані потрібно отримати зі сховища даних або зберегти у сховищі, шар бізнес-логіки звертається до шару доступу до даних, передаючи\отримуючи збережені дані.

Шар представлення – це додаток типу відповідного до обраного інтерфейсу (CUI чи GUI). Людина-користувач через шар представлення взаємодіє з програмною системою через інтерфейс, реалізований шаром представлення. Для виконання дій на вимогу користувача шар представлення звертається до шару бізнес-

логіки, передаючи в нього ведені користувачем дані та команди виконати певні дії. Результати виконаних дій, отримані від шару бізнес-логіки, шар представлення відображує користувачу у консолі\формах.

**Принцип багатошаровості при будуванні архітектури є обов’язковим до застосування!**

1. Реалізувати обов’язковий контроль коректності введених даних в шарі представлення.
2. Реалізувати перевірку виняткових ситуацій в шарі бізнес-логіки та доступу до даних; в тому числі – при роботі з даними. При необхідності створити власні класи виключень (наприклад, виключення для ситуації перевищення ліміту кількості книжок на абонементі в бібліотеці). Генерація/ виникнення виняткових ситуацій, як правило відбувається на іншому рівні (рівнях), ніж їх обробка.
3. Написати модульні тести, використовуючи певний фреймворк (MSTest, Nunit, Xunit, тощо), до бізнес логіки. Мінімальне покриття тестами – 100% функціоналу однієї з сутностей, а також – мінімум 50% функціоналу усіх інших сутностей. Покриття можна продемонструвати відповідними засобами, наприклад AxoCover, CodeCoverage та ін (допускається також детальне описання функціоналу, покритого модульними тестами). Модульні тести повинні бути окремим проектом в рішенні. Для оформлення коду модульних тестів обов’язково застосовувати принцип Triple A.
4. Сутності за зв’язки між ними повинні бути спроектовані, відповідно до базових правил ООП, composition over inheritance, loose coupling – high cohesion, inversion of control (IoC). А також - з використанням узагальнень.
5. При написанні коду застосувати правила «хорошого» стилю програмування. А саме: створювати мнемонічні ідентифікатори, чітко форматувати код, застосувати при необхідності коментарі. Не використовувати public полів у класах. За необхідності доступу до них використати властивості, індексатори чи окремі методи-аксесори. Опис класів наводити в окремих файлах.

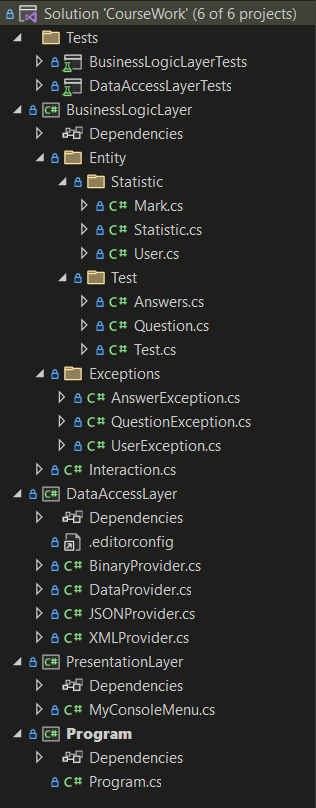
[Індивідуальне завдання](#ІІ)

1. **Керування питаннями тестів**
   1. Можливість додавати питання
   2. Можливість видаляти питання
   3. Можливість змінити питання
   4. Можливість переглянути перелік всіх питань
2. **Керування відповідями**
   1. Можливість додавати відповідь
   2. Можливість видаляти відповідь
   3. Можливість змінювати відповідь
   4. Можливість переглянути перелік всіх відповідей до конкретного питання
      1. Помітити правильну відповідь позначкою «+»
   5. Варіанти відповідей до питань генеруються автоматично
      1. Передбачити наявність правильної відповіді
3. **Керування тестами**
   1. Можливість створити тест
   2. Можливість змінювати тест
      1. змінити кількість питань у тесті
      2. змінити час відведений на одне питання
   3. Можливість порахувати процент правильних відповідей
   4. Можливість передчасно вийти з тесту
4. **Пошук**
   1. Пошук тестів
   2. Перегляд статистики тестів

[Шари проекту та загальна архітектура проекту](#ІІІ)

1. Проект складається з чотирьох шарів:

* шар доступу до даних, на ньому відбувається серіалізація та десеріалізація;
* шар бізнес-логіки, на ньому знаходяться класи-сутності, класи-винятки та клас взаємодії між шаром доступу до даних, класами-сутностями та шаром представлення;
* шар представлення містить клас меню, який взаємодії з користувачем та шаром бізнес-логіки. Всі ці 3 шари є DLL, тобто бібліотеками.
* 4 шар є ConsoleApplication, в ньому безпосередньо відбувається запуск класу меню з шару представлення.
* Також проект містить окремий project, який у свою чергу містить тести до класу взаємодії з шару бізнес-логіки.



**[Компоненти Проекту](#ІV)**

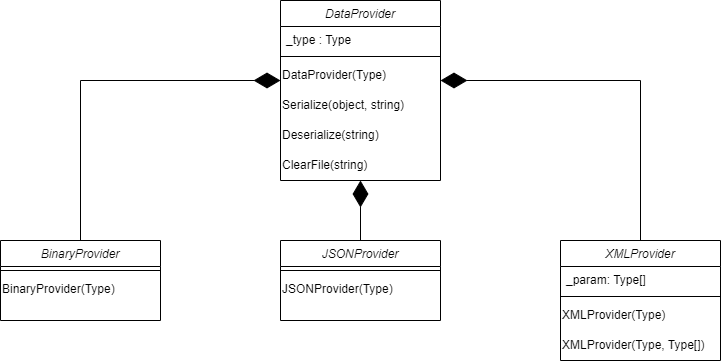
1. В проекті DataAccessLayer є базовий абстрактний клас DataProvider. В ньому є одне захищене поле \_type типу Type, користувацький конструктор, два абстрактних метода Serialize i Deserialize. Також він містить відкрий метод ClearFile. Далі від нього успадковуються усі конкретні провайдери, а саме Binary, JSON та XML.
2. Binary та JSON провайдери повністю реалізують функціонал базового класу та нічого нового не додають. А XML провайдер у свою чергу, додає нове readonly поле типу масива Type. Це поле треба для повноцінної XML серіалізації та десеріалізації.
3. **Сутності**:

* **Клас Answers** успадковується від класу List<string>. Додатково в ньому є одна константа типу int, яка показує максимальну кількість відповідей у класі. Також в ньому є дві властивості RightAnswer і UserAnswer. Окрім успадкування від List<string>, клас успадковується від інтерфейсу IFormattable. Цей інтерфейс дозволяє розробити власні формати, які можна потім використовувати при виклику методу ToString(). Для цього інтерфейсу я створив 4 константи типу string, які відповідають за формати. Є 4 формати default, test, answer і compare. Default виводить усі відповіді. У форматі test відповіді виводяться з вказанням відповіді користувача. У формати answer відповіді виводяться з вказанням правильної відповіді. А формати compare об’єдную в собі формати test i answer.
* **Клас Question** містить два поля, одне поле типу string, яке містить саме питання та поле типу Answers. В класі є властивості обгортки до властивостей з класу Answers. Також є індексатор, який повертає відповідь за індексом. Є додаткові методи Reset та CheckForRightAnswer. Метод Reset повертає властивість UserAnswer у стан null. А CheckForRigthAnswer перевіряє чи у полі Answers є правильна відповідь.
* **Клас User** використовується разом з класом Mark, для зберігання статистики проходження тесту. В класі User є дві властивості типу string? FirstName, LastName.
* **Клас Mark** містить константу типу string, яка означає формат часу. Також в класі є дві властивості, одна типу double, яка зберігає безпосередньо оцінку, та властивість типу DateTime, яка зберігає час закінчення проходження тесту.
* **Клас Statistic** успадковується від класу List<string>. Додатково містить перегрузки методів Add і AddRange. Метод Add приймає два параметри типу User і Mark. А метод AddRange приймає параметр типу IDictionary<User, Mark>. Ці методи потрібні для того, щоб додавати до статистики запис з користувачем та оцінкою. Я використовую List<string>, доцільніше було б використати словник, але словник не підтримує серіалізацію. Також клас містить два додаткових метода які повертають статистку за користувачем або за датою.
* **Клас Test** містить в собі List<Question> та поле типу Statistic. Та всі методи обгортки для взаємодії зі списком запитання та полем Statistic. Також як і клас Answers, клас Test успадковується від інтерфейсу IFormattable, для того щоб виводити відповіді у вибраному форматі.

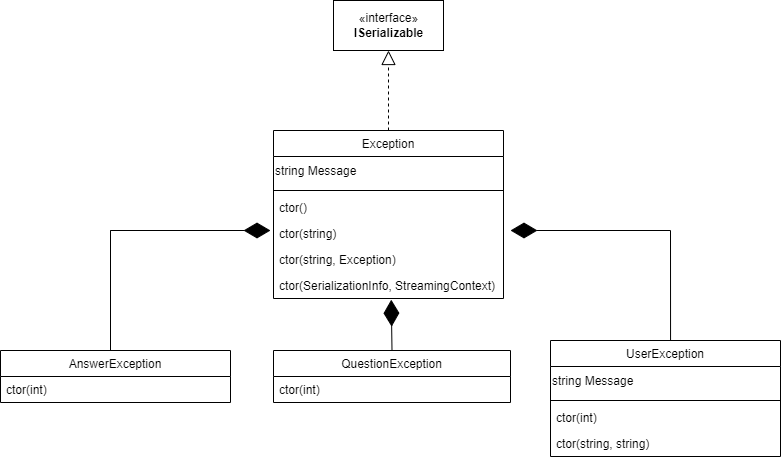
1. Ще одною особливістю є те що, в класах сутностях нічого не перевіряється на коректність. Тобто чи коректне питання, чи можна ще додати відповідь і тд. Все це перевіряється в **класі Interaction**. Так само в цьому класі перевіряється коректність назви файлу та коректність його змісту.

[Діаграма класів](#V)

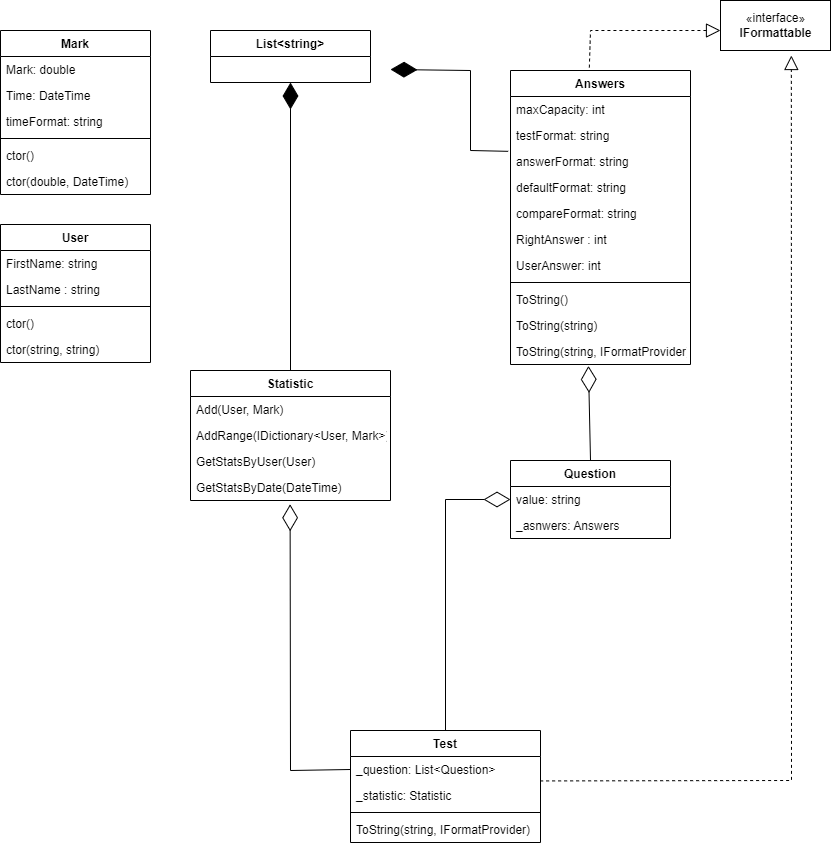
1. DataAccessLayer



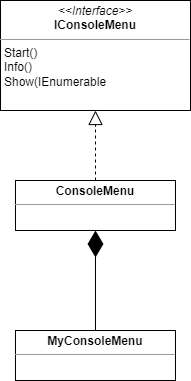
1. –



1. –



1. -



**[Особливості Реалізації](#VI)**

1. Класи Answers i Statistic замість того щоб зберігати в собі List<string> самі від нього успадковуються

**public class Answers : List<string>, IFormattable**

**public class Statistic : List<string>**

1. **Клас Interaction** замість того щоб містити окремі методи для серіалізації та десеріалізації кожного провайдера, містить два словника, які замінюють відповідні методи.

static readonly Dictionary<string, Func<string?, object?>> deser = new()

{

{ ".xml", (filePath) => new XMLProvider(typeof(Test), new Type[]

{

typeof(Question), typeof(Answers), typeof(Statistic), typeof(User), typeof(Mark)

}).Deserialize(filePath) },

{ ".dat", (filePath) => new BinaryProvider(typeof(Test)).Deserialize(filePath) },

{ ".json", (filePath) => new JSONProvider(typeof(Test)).Deserialize(filePath) }

};

static readonly Dictionary<string, Action<object, string?>> ser = new()

{

{".xml", (graph, filePath) => new XMLProvider(typeof(Test), new Type[]

{

typeof(Question), typeof(Answers), typeof(Statistic),typeof(User), typeof(Mark)

}).Serialize(graph, filePath) },

{".dat", (graph, filePath) => new BinaryProvider(typeof(Test)).Serialize(graph, filePath) },

{".json", (graph, filePath) => new JSONProvider(typeof(Test)).Serialize(graph, filePath) }

};

1. **Клас MyConsoleMenu** замість того щоб визивати методи через оператор switch, він містить словник, ключами в ньому виступають команди типу стрінг, а значеннями делегати типу Action, які в свою чергу визивають методи відповідно до команди.

static readonly Dictionary<string, Action> mainMenu = new();

static readonly Dictionary<string, Action> changeMenu = new();

mainMenu.Add("/info", () => Info());

mainMenu.Add("/create", () => Create(false));

mainMenu.Add("/create def", () => Create(true));

mainMenu.Add("/change", () => StartChanging());

mainMenu.Add("/pass", () => PassTest());

mainMenu.Add("/stats", () => StartStats());

mainMenu.Add("/show", () => ShowTest());

mainMenu.Add("/get all", () => ShowQuestions());

mainMenu.Add("/clear", () => ClearFile());

mainMenu.Add("/cls", () => Console.Clear());

mainMenu.Add("/end", () =>

{ Console.WriteLine("Bye, have a good time!"); });

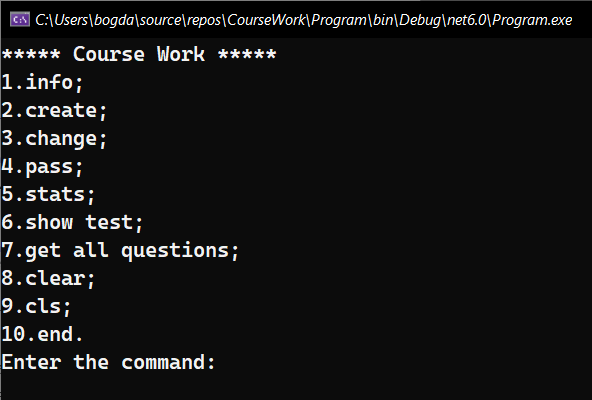
changeMenu.Add("/info", () => ChangeInfo());

changeMenu.Add("/question", () => StartChangingQuestion());

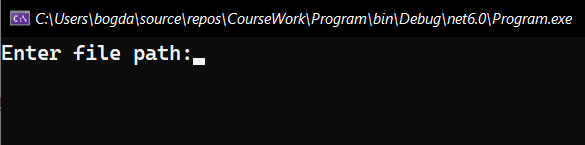
changeMenu.Add("/answers", () => StartChangingAnswers());

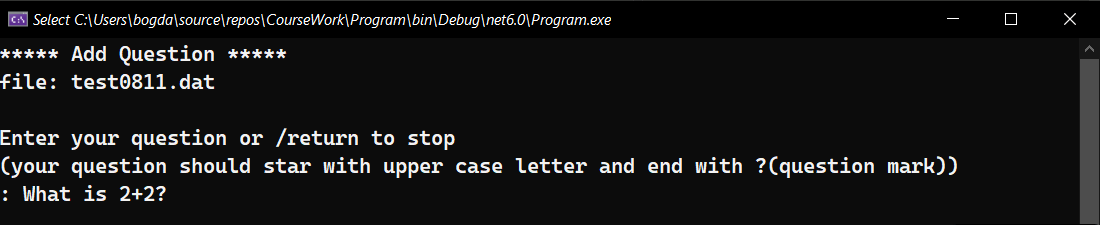
**[Описання функціоналу](#VII)**

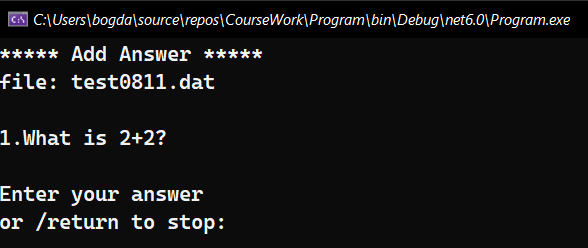
1. **При запуску програми користувача зустрічає перелік усіх команд та запит на введення команди, можна вводити як номер команди так і саму команду з / попереду.**

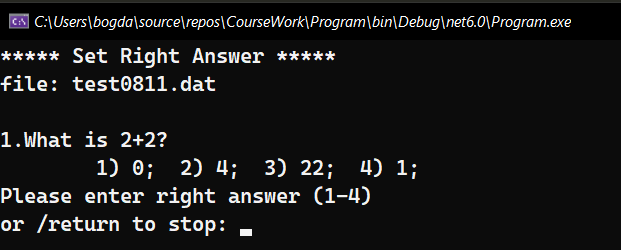


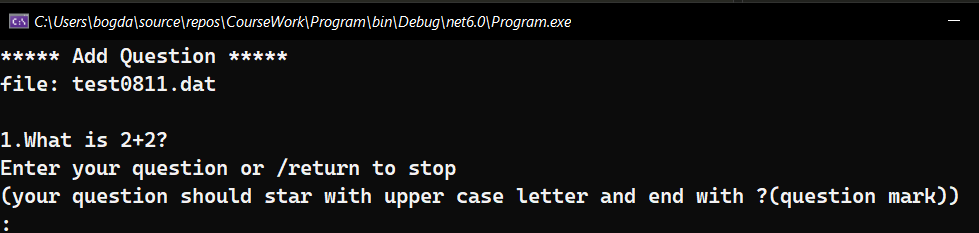
1. **Команда info – виводить перелік команд, такий же самий як при запуску програми.**
2. **Команда create, запитує файл, потім очищає його і пропонує ввести питання, а потім і відповіді до нього, і це продовжується допоки користувач не введе команду /return or /end.**



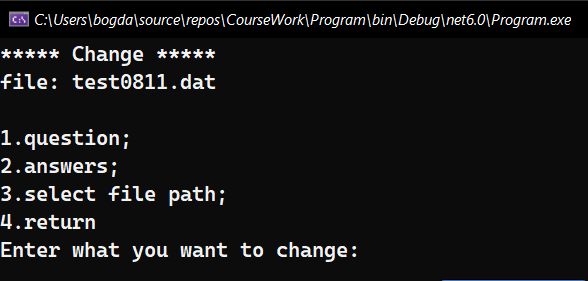




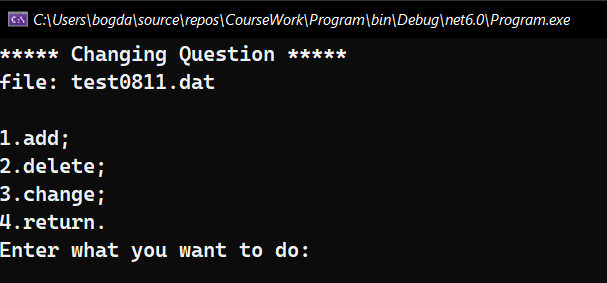


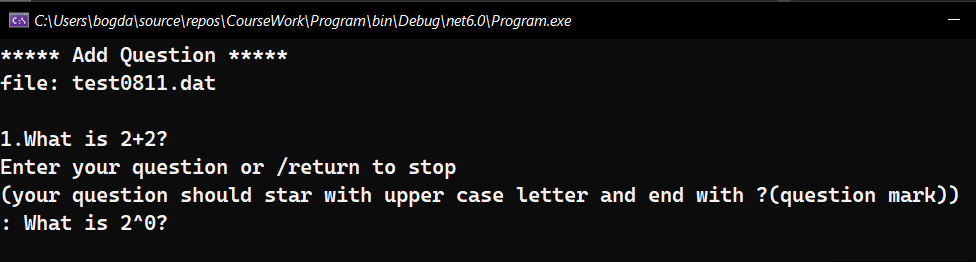


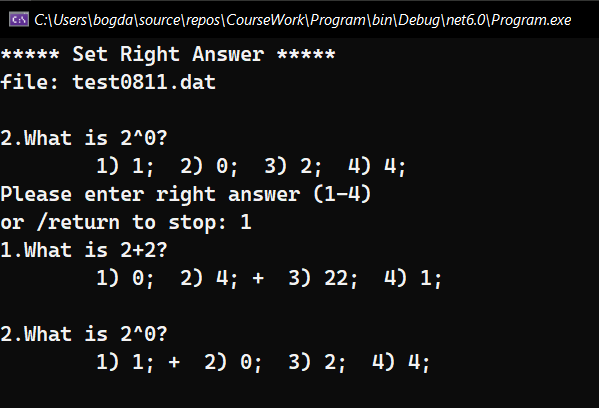
1. **Команда change, спочатку питає назву файл, а потім виводить нове меню з командами, де можна вибрати, що змінювати.**

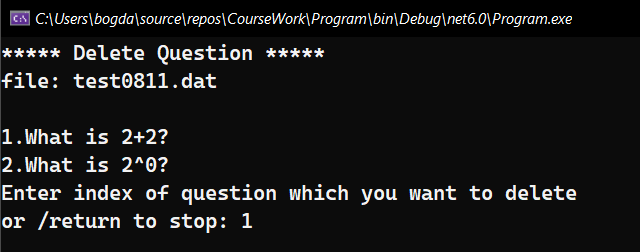


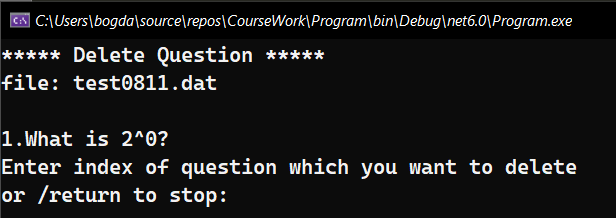
1. **Вибір команди question, призводить до появи нового меню, де можна додати, видалити або змінити питання в тесті.**

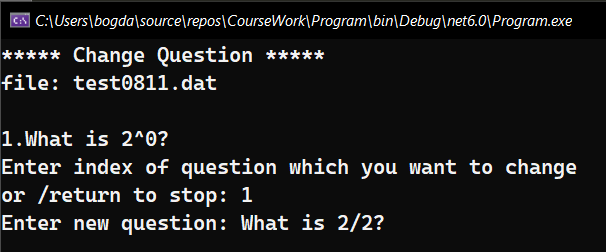




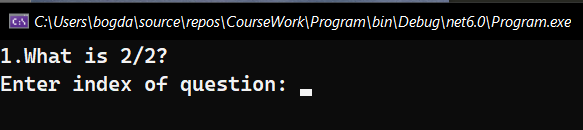


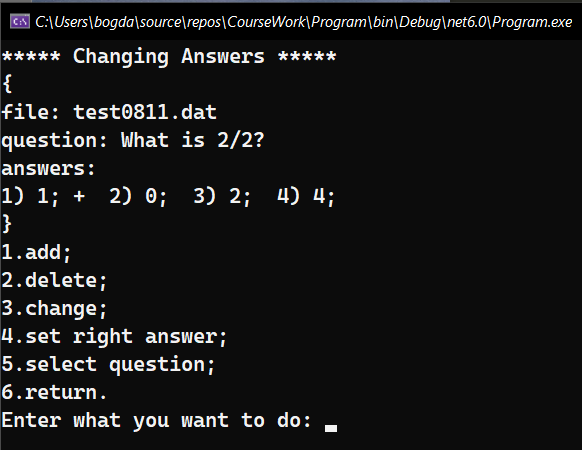






1. **Аналогічне до меню question, меню answers, команди роблять теж саме тільки з відповідями до питання і перед тим як вивести меню, програма запитує номер питання, відповіді якого будуть змінювати.**





1. **Далі усі команди виконують свій функціонал згідно з назвою. Усі скріншоти робити команд знаходяться в папці Funcitonality Screenshots.**

**[Використані Джерела](#VIII)**

1. **[C# docs - get started, tutorials, reference. | Microsoft Learn](https://learn.microsoft.com/en-us/dotnet/csharp/)**
2. [**(2) IT Курси програмування онлайн - навчання програмуванню, відео уроки | ITVDN**](https://itvdn.com/ua)
3. **CLR via C# 4th Edition G.Richter.**
4. [**Stack Overflow - Where Developers Learn, Share, & Build Careers**](https://stackoverflow.com/)