# Зміст



# Скорочення:

ДПЗ[1] - словник з даними про зміну ціни в залежності від днів до терміну придатності.

## Вироблення вимог:

Створити систему роботи з даними складу продуктів, яка відповідає переліку потреб:

- Зберігання даних.
- Зчитування даних.
- Маніпулювання даними. (Видалення, додавання, оновлення)
- Користувацький інтерфейс
- Обробка даних.
- Обробка помилок.
- Можливість подальшого супроводу та масштабування системи

## Проектування класів:

## Продукти:

***Сегрегація інтерфейсів:***

Основною сутністю у проекті є склад, який складається з продуктів, тож варто розпочати проектування з продумування архітектури продуктів.

У приклад візьмемо 3 типи продуктів:

- М’ясні (Назва, ціна, вага, термін придатності, тип м’яса, категорія м’яса, ДПЗ[1])
- Молочні (Назва, ціна, вага, термін придатності, ДПЗ[1])
- Фільми (Назва, ціна, жанр, тривалість, посилання, ім’я автора)

Для більш зручного пояснення наступних сутностей варто  перерахувати усі параметри, та класи, які їх містять

- Назва (М’ясні, Молочні, Фільми)
- Ціна (М’ясні, Молочні, Фільми)
- Вага (М’ясні, Молочні) < мається на увазі фізична вага (кг)
- Термін придатності (М’ясні, Молочні)
- ДПЗ[1] (М’ясні, Молочні)
- Тип м’яса (М’ясні)
- Категорія м’яса (М’ясні)
- Жанр (Фільми)
- Тривалість (Фільми)
- Посилання (Фільми)
- Ім’я автора (Фільми)

Переглянувши усі наразі існуючі параметри, можна побачити, що об’єднує усі продукти, а саме: Назва, Ціна.

Тож максимальним рівнем абстракції для продуктів, я пропоную зробити інтерфейс  IProduct, та надалі від нього відштовхуватимуся.

Тож IProduct буде містити

Властивості:

- Назва
- Ціна

Методи:

void ChangePrice(int) : змінює ціну на заданий процент.

Наслідувати:

- IComparable < потрібен буде для сортування
- ICloneable < для створення глибоких копій у подальшому використанні як Generic

Перевірка адекватності:

(надалі буду використовувати такий підхід для перевірки сутностей на вірність)

Чи має БУДЬ ЯКИЙ продукт:

Назву – так

Ціну – так

ChangePrice - так

(на цьому етапі можна посперечатися, але для максимального рівня абстракції потрібні певні допущення, які задовольняють логіку та вимоги задачі, у нашому випадку ці допущення такі, що БУДЬ ЯКИЙ продукт має: Назву, Ціну та можливість змінити ціну).

Наступний рівень абстракції:

Наразі ми маємо уявлення продуктів лише у вигляді інтерфейсу IProduct, а нам потрібно додавати нові продукти, можна було б наприклад додати для кожного продукту свій інтерфейс, як IMeatProduct, IDairyProduct, IMovieProduct, які б наслідували інтерфейс IProduct, але у такому варіанті є декілька проблем, по перше це повтор коду, наприклад IMeatProduct та IDairyProduct, мають Вагу, Термін придатності та ДПЗ, а це тільки 2 типи продуктів, їх може бути безліч створених користувачем, тоді це буде не

зручно і не про яку гнучкість та повторне використання цих інтерфейсів мови не йде, тож варто створити наступний рівень абстракції, розділивши параметри на нові, менші інтерфейси.

Спочатку варто було б розділити продукти на 2 типи: віртуальні та фізичні, додавши інтерфейси.

IDigitalProduct:

Властивості:

- Посилання

Наслідує IProduct

Перевірка адекватності:

Чи БУДЬ ЯКИЙ віртуальний продукт:

Має посилання – так

Є IProduct – так

IPhysicalProduct:

Властивості:

- Вага

Наслідує IProduct

Перевірка адекватності:

Чи БУДЬ ЯКИЙ фізичний продукт:

Має вагу – так

Є IProduct – так

Залишилися поза інтерфейсів такі параметри:

- Термін придатності (М’ясні, Молочні)
- ДПЗ[1] (М’ясні, Молочні)
- Тип м’яса (М’ясні)
- Категорія м’яса (М’ясні)
- Жанр (Фільми)
- Тривалість (Фільми)

- Ім’я автора (Фільми)

Термін придатності та ДПЗ[1] пов’язані  бо логіка зміни ціни продуктів залежить від терміну значення у ДПЗ[1] , а його значення залежить від Терміну придатності, тож винесемо у:

IExpirationProduct:

Властивості:

- Дата терміну придатності
- ДПЗ[1]

Методи:

- double GetPriceByExpiration() : Повертає ціну змінену відповідно терміну придатності

Наслідує IProduct

Перевірка адекватності:

Чи БУДЬ ЯКИЙ продукт з терміном придатності:

Має дату терміну придатності – так

Має ДПЗ[1] – так

Є IProduct – так

Наразі у нас залишилися такі параметри:

- Тип м’яса (М’ясні)
- Категорія м’яса (М’ясні)
- Жанр (Фільми)
- Тривалість (Фільми)
- Ім’я автора (Фільми)

Як можна побачити, вони відповідні тільки одному продукту, то чи варто занести їх саме до інтерфейсів цих продуктів ? А саме у IMeatProduct та IMovieProduct.

Йдемо по черзі:

- Тип м’яса (М’ясні) – чи ТІЛЬКИ м’ясні продукти мають це параметр? Так.
- Категорія м’яса (М’ясні)- чи ТІЛЬКИ м’ясні продукти мають це параметр? Так.
- Жанр (Фільми) – чи ТІЛЬКИ фільми мають цей параметр? Ні.
- Тривалість (Фільми) – чи ТІЛЬКИ фільми мають цей параметр? Ні.
- Ім’я автора (Фільми) – чи ТІЛЬКИ фільми мають цей параметр? Ні.

Тож можна прийти до висновку - Тип м’яса та Категорія м’яса можна напряму занести у IMeatProduct, а інші параметри варто розподілити на окремі інтерфейси.

IGenreProduct:

Властивості:

- Жанр

Наслідує IProduct

Перевірка адекватності:

Чи БУДЬ ЯКИЙ продукт з жанром:

Має Жанр – Так

Є IProduct – так

IAuthorProduct

Властивості:

- Ім’я Автора

Наслідує IProduct

Перевірка адекватності:

Чи БУДЬ ЯКИЙ продукт з Автором:

Має Ім’я Автора– Так

Є IProduct – Так

IDurationProduct

Властивості:

- Тривалість

Наслідує IProduct

Перевірка адекватності:

Чи БУДЬ ЯКИЙ продукт з Тривалістю:

Має Тривалість – Так

Є IProduct – Так

І тепер, після **сегрегації інтерфейсів**, ми маємо можливість зручно та гнучко їх використовувати, тож тепер можна перейти до наступного рівня абстракції.

Після розділення на інтерфейси, потрібно знайти можливі абстракції для наших продуктів, наприклад м’ясні та молочні продукти – їжа, і їх об’єднують такі параметри:

- Назва (М’ясні, Молочні, Фільми)
- Ціна (М’ясні, Молочні, Фільми)
- Вага (М’ясні, Молочні)
- Термін придатності (М’ясні, Молочні)
- ДПЗ[1] (М’ясні, Молочні)

Ці умови задовольняють наші інтерфейси IPhysicalProduct та IExpirationProduct

З їх поєднання ми отримаємо інтерфейс IFoodProduct:

Наслідує:

- IExpirationProduct
- IPhysicalProduct

Перевірка:

Чи БУДЬ ЯКА їжа є :

- IExpirationProduct – так
- IPhysicalProduct – так

І тепер ми маємо основу для інтерфейсів наших продуктів

IDairyProduct:

Наслідує IFoodProduct

Перевірка:

Чи БУДЬ ЯКИЙ молочний продукт:

Є IFoodProduct – так

IMeatProduct:

Властивості:

- Тип м’яса
- Категорія м’яса

Наслідує IFoodProduct

Перевірка:

Чи БУДЬ ЯКИЙ м’ясний продукт:

Має Тип м’яса – так

Має Категорія м’яса – так

Є IFoodProduct – так

IMovieProduct:

Наслідує:

- IDigitalProduct
- IAuthorProduct
- IGenreProduct
- IDurationProduct

Перевірка:

Чи БУДЬ ЯКИЙ фільм:

- Є IDigitalProduct – так
- Є IAuthorProduct – так
- Є IGenreProduct – так
- Є IDurationProduct – так

***DRY – не повторюй себе***

Маючи інтерфейси IDairyProduct, IMeatProduct, IMovieProduct, можна починати робити їх реалізації, але вони мають достатньо багато спільного, тож задля зменшення кількості повторюваного коду та підвищення абстракції, варто  додати базові абстрактні класи з базовою реалізацією методів, властивостей та конструкторів.

Першим базовим класом зручно було б зробити ProductBase тож:

abstract ProductBase:

Реалізує IProduct

Маючи ProductBase варто також зробити базу для наших їстівних продуктів:

abstract FoodProductBase:

Наслідує ProductBase

Реалізує IFoodProduct

***Реалізація моделей***

Наразі нічого не заважає почати реалізовувати конкретні моделі даних

MovieProductModel:

Наслідує ProductBase

Реалізує IMovieProduct

DairyProductModel:

Наслідує FoodProductBase

Реалізує IDairyProduct

MeatProductModel

Наслідує FoodProductBase

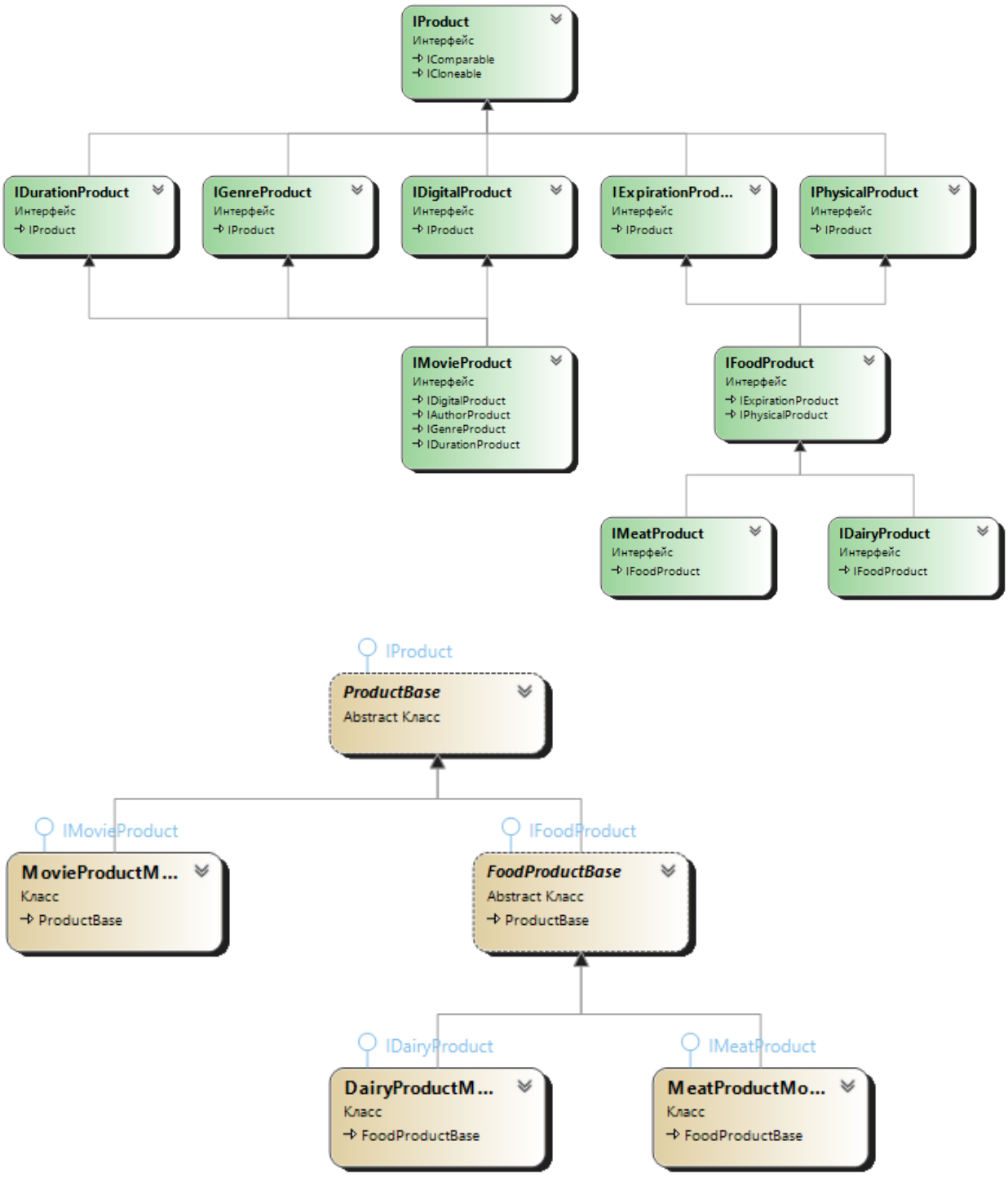
Реалізує IMeatProduct

***Висновок проектування продуктів:***

Використовуючи принципи сегрегації інтерфейсів та DRY, ми отримали механізм додавання нових продуктів, зручний до масштабування. А також такий підхід відповідає принципу Барбари Лісков

***Діаграма:***

Нажаль через брак часу не зміг реалізувати UML діаграму руцями, тож використав вбудовану можливість VS2022



## Склад продуктів+Task 12.1-2

Завдяки створенню усіх продуктів на базі IProduct, ми можемо створити ProductStorage<T>

Де T є IProduct

Наслідує IList<T>

- Має Список типу T
- Має подію, що спрацьовує при додаванні нового продукту, використовується у Add, Insert, та конструкторі. Приймає T та повертає бульове значення (Predicate) (виконання завдання 12.1)
- Має подію, що спрацьовує при невдалому додаванні продукту, вона отримує строкове значення, та повинна його обробити певним чином, наприклад записати у логи(виконання завдання 12.1)

Таким чином ми отримуємо Generic склад, що може містити у собі будь який тип продуктів та може бути створеним на основі будь якого типу що наслідує IProduct.

Проблема з глибокою копією вирішується тим що  IProduct наслідує ICloneable.

Завдяки сегрегації інтерфейсів при роботи з колекцією таких об’єктів є можливість робити вибірки по типу продуктів таким методом

```
public IEnumerable<G> GetAll<G>() where G : T
{
    foreach (T item in _products)
    {
        if (item is G result)
        {
            yield return result;
        }
    }
}
```

Також мається його перевантаження з певною умовою

```
public IEnumerable<G> GetAll<G>(Predicate<G> predicate) where G : T
        {
            foreach (T item in _products)
            {
                if (item is G result && predicate(result))
                {
                    yield return result;
                }
            }
        }
```

Тим самим виконано завдання 12.2

А завдяки наслідування IList<T> ми можемо працювати з цим класом, як з списком.

## Зберігання даних та зчитування даних

Як у процесі зчитування, так і зберігання даних буде використовуватися клас

TXTSerializedParameters:

Реалізує IDictionary<string, string>

Має:

- string PrimalLine : відповідає за початковий вигляд строки, яку зберігає цей клас у вигляді словника
- Dictionary<string, string> _parameters : зберігає дані про параметри, ключ – назва властивості, значення – значення властивості у вигляді стрічки

ToString() перевантажена так, що на виході отримуємо стрічку у вигляді:

"<{назва властивості1}: {значення властивості2}>;"<{н.в2}: {з.в2}>;

…"<{ назва властивості n}: { значення властивості n}>;

для для n значень у словнику.

Приклад для класу MeatProductModel:

```
<Type: MeatProductModel>;<MeatSpeciesProp: CHICKEN>;<MeatCategoryProp:
FIRST>;<ExpirationTime: 17.07.2022 00:00:00>;<Weight: 2>;<Price:
120>;<DaysToExpirationAndPresentOfChange: {[10, 30]}>;<Name: Крильця>;
```

Та статичний клас сервіс

FileHandlerService:

Методи:

- TryReadToObject<G>: G є new() тобто є об’єктом класу
  Приймає:
  - Out G : сюди буде записаний результат читання
  - IStreamLineReader<G> : це логіка читання з файлу
  - IStringParser<G> : логіка парсингу з TXTSerializedParameters у об’єкт типу G
  - String : шлях до файлу
- Повертає: bool : чи вдалося зчитати
  Опис: зчитує з файлу 1 строку і парсить її у об’єкт типу G

- ReadToCollection<T, G>: T є IEnumerable<G> тобто колекцією з G
  Приймає:
  - Ref T : сюди буде записаний результат читання
  - IStreamCollectionReader<T, G>: це логіка читання з файлу
  - Dictionary<string, IStringParser<G>>: логіка парсингу з TXTSerializedParameters у об’єкти типу G
  - String : шлях до файлу

  Опис: зчитує з файлу пострічно колекцію з G, та має словник з назв класів G (бо G можуть бути дочірніми класами класу G ), та логікою як ці класи парсити

- WriteToFile<T,G> T є класом
  Приймає:
  - T : цей об’єкт буде записаний у файл
  - ISerializer<G> : відповідає за логіку серіалізації, G – формат сераілізації
  - String : шлях до файлу
  - Bool : чи доповнювати файл

  Опис: Записує, серіалізований за логікою ISerializer<G>, обєкт T у файл

- WriteToFileCollection<T, G>T є IEnumerable
  Приймає:
  - T : колекція, що буде записана у файл
  - ISerializer<G> : відповідає за логіку серіалізації, G – формат сераілізації
  - String : шлях до файлу
  - Bool : чи доповнювати файл

  Опис: Записує, колекцію серіалізованих за логікою ISerializer<G>, обєктів T у файл

***Зберігання даних***

Для зберігання даних використовується методи WriteToFile та WriteToFileCollection з класу FileHandlerService

Вище вони описані, результат запису залежить від інтерфейсу

ISerializer<T>:

Методи:

- Serialize<G>
  - Приймає
    - In G : цей об’єкт буде сераілізовуватися
  - Повертає
    - T : серіалізовний формат

Опис: При реалізації інтерфейса задається T тобто повертаємий з методу Serialize серіалізований тип.

Реалізація:

TxtSerializer

Реалізує: ISerializer<TXTSerializedParameters>

Опис: цей клас за допомогою рефлексії отримує властивості та сераілізує їх у вигляді TXTSerializedParameters

Таким чином ми отримуємо можливість ззовні сереілізувати будь який об’єкт, при реалізованій для нього логікі

***Зчитування даних***

Для зчитування даних використовується методи TryReadToObject та ReadToCollection з класу FileHandlerService

Читання поділяється на 2 частини

1. Розпарсити сераілізовану строку
2. Валідувати результат у об’єкти

За парсинг строки у відповідний до валідації формат відповідає інтерфейс

IObjectGetterFromSerializedLine<T>

Методи:

- TryGetObject<G>
  - Приймає:
    - Out G : сюди буде записаний результат
    - String : строка з якої буде читатися об’єкт
    - IStringParser<G>: парсер, яким буде валідуватися результат форматування строки
  - Повертає: bool : чи вдалося зчитати

Реалізація:

TXTSerializedProductReader<T>

Реалізує IObjectGetterFromSerializedLine<T>

Опис: Використовує TXTSerializedLineAnalyzer щоб отримати TXTSerializedParameters, які заносяться у obj за допомогою IStringParser

***Зчитування колекцій***

IStreamCollectionReader<T, G>: Де T є колекцією G

Метод:

void ReadCollection (ref T obj, StreamReader stream, Dictionary<string, IStringParser<G>> validator)

Цей метод зчитує дані з stream у колекцію obj, а за логіку читання відповідає словник з назви моделі даних та парсера для цієї моделі

Подія: event LoggerOnBadFormat OnBadFormatLogger; буде відповідати за обробку логів під час парсингу

Реалізація зчитування колекції

TXTSerializedStorageReader<T> :Де T є IProduct

Наслідує:  IStreamCollectionReader<ProductStorage<T>, T>

***Парсери***

ITXTSerializedParametersParser<out T>:

Методи: T Parse(TXTSerializedParameters txtSerializedParams);  цей метод приймає TXTSerializedParameters які треба розпарсити та повертає значення T

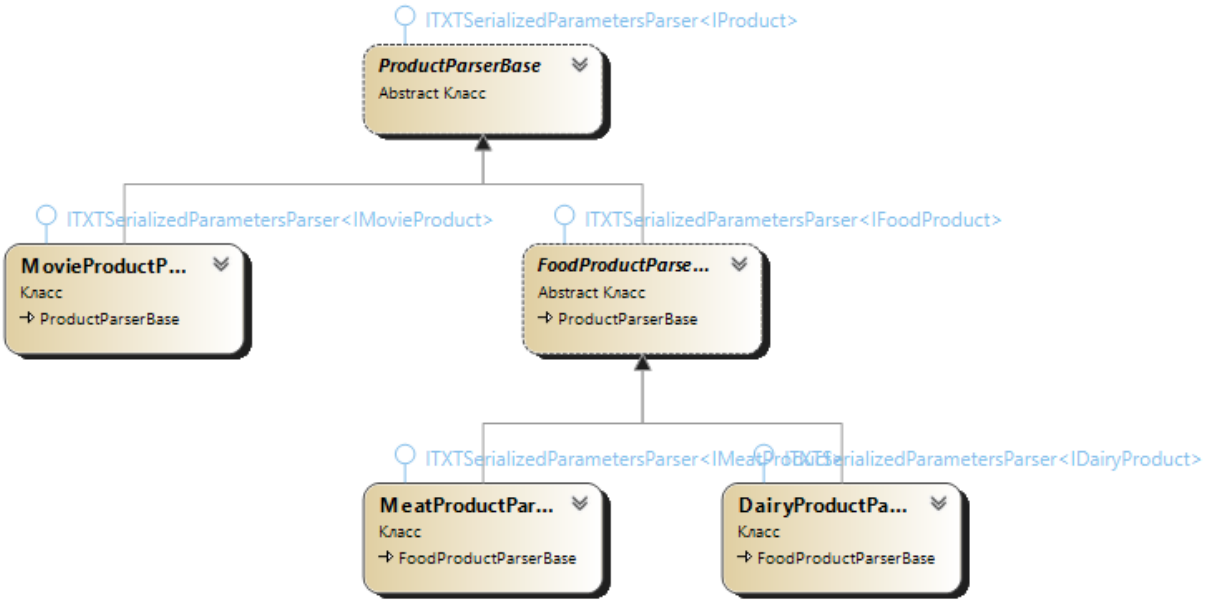
Подія: event LoggerOnBadFormat OnBadFormatLogger; буде відповідати за обробку логів під час парсингу

LoggerOnBadFormat:

delegate void LoggerOnBadFormat(string message)

***Можливість до DRY***

Завдяки сегрегації інтерфейсів продуктів і подальшого приведення до деревовидного наслідувань базових класів, тепер можна зробити те ж саме з парсерами у вигляді:




Що дуже пришвидшує роботу, зменшує кількість коду, та надає всі переваги принципу DRY

***Проблема інваріантності***

При спробі додавання до словника типу <string, ITXTSerializedParametersParser<IProduct> > Парсерів що реалізовують ITXTSerializedParametersParser<T> з дочірним до IProduct T, це не вдавалося, так як інтерфейси по замовченню інваріантні, але я знайшов можливість виправити це додавши out до T, що робить ці типи коваріантними, чи добра ця практика? Я б хотів дізнатися під час заняття.

## Обробка помилок(Logger)

У якості обробки помилок ми будемо їх зберігати у лог для цього потрібен логер, у минулих своїх роботах я використовував статичний клас у якості логера і напряму його викликав де він був потрібен, але це зменшувало гнучкість коду, тож цього разу я використовую комбінацію з подій та патерну сінглтон тобто маю клас

Logger:

Поля(усі приватні):

- Кількість створених записів
- Шлях
- _instance Об’єкт цього класу (статичний, тільки для читання)

Властивості:

- Кількість створених записів
- Шлях
- Instance (статична ) якщо _instance порожній, то записує новий об’єкт у це поле та повертає його, якщо не порожній то просто повертає

Методи:

- Log(string logLine): записує лог у файл зазначений у шляху

Цей метод буду підписуватися на події, що відповідають за логування.