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.
- Користувацький інтерфейс (відображення даних)
- Маніпулювання даними. (Видалення, додавання, оновлення)
- 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

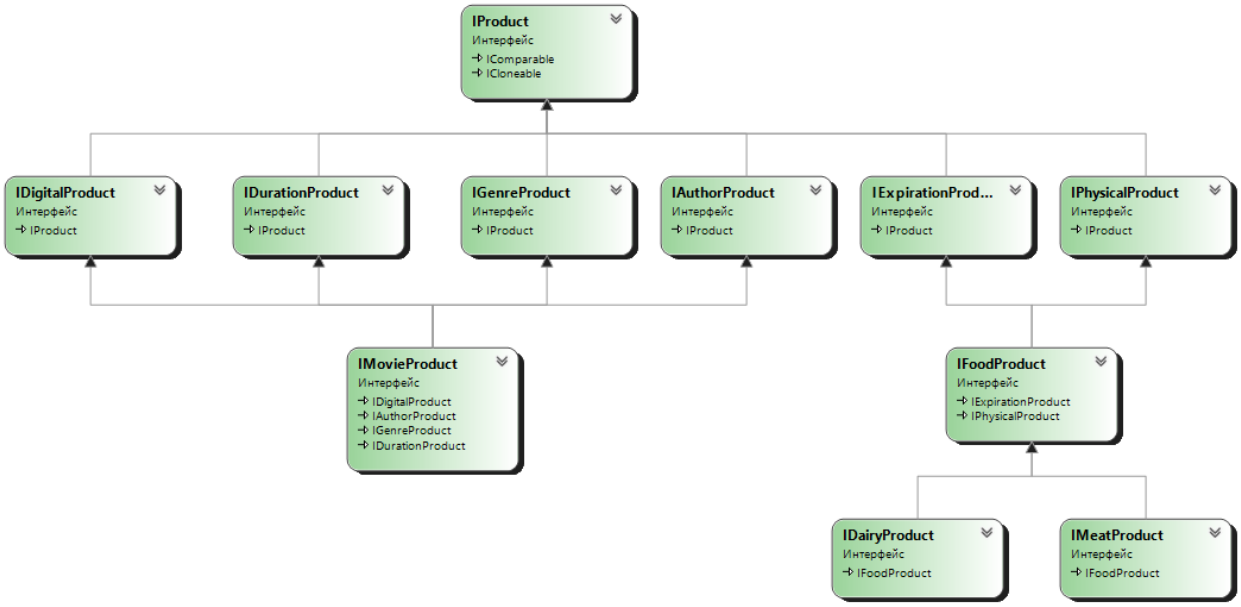
Реалізує IMeatProduct

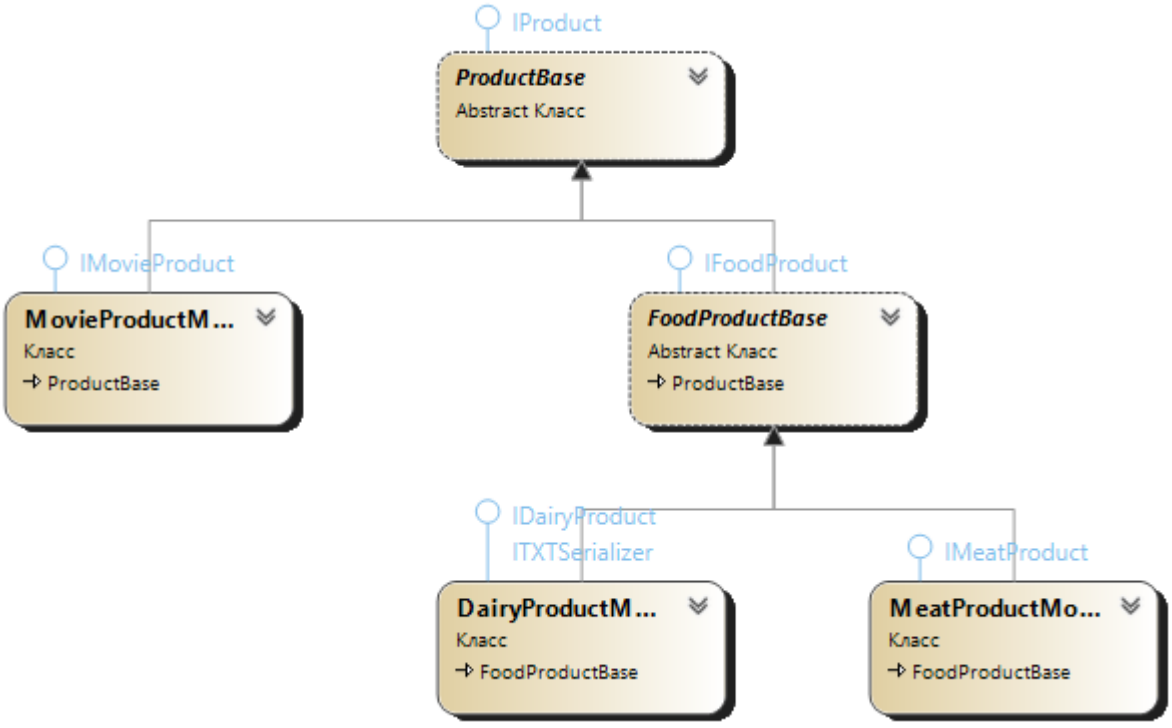
Висновок проектування продуктів:

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

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

Де T є IProduct

Наслідує IList<T>

Має Список типу T

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;
        }
    }
}
```

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

Зберігання даних

Механізм зберігання даних буде складатися з 2 етапів

1. Перетворення об’єкта у текст певного формату
2. Запис результату у файл

Формат буде txt, а форматування строки у вигляді:

<тип продукту>;<Назва властивості: значення>;<…>...

Приклад для DairyProductModel:

```
<DairyProduct>;<Name: Name>;<Price: 10,31>;<Weight: 3>;<ExpirationTime:
01.01.2022>;<DaysToExpirationAndPresentOfChange: {(3 50)(6 20)(8 10)}>;
```

Логіка форматування у текст буде містити інтерфейс

ITXTSerializer:

Метод: string SerializeTxt();

Тим самими, ми будемо знати, що будь який клас, що реалізує ITXTSerializer можна зберегти у текстовий файл.

Запис результату буде завдяки методу WriteToFile у сервісному статичному класі FileHandlerService

Void WriteToFile(ITXTSerializer, string, bool)

Приймає який об’єкт записати, шлях та isAppend

Зчитування даних

Щоб зчитати дані треба знати:

- Куди зчитувати
- Звідки зчитувати
- Як зчитувати
- Чи вдалося зчитати

В основі ми маємо 2 варіанти що можна зчитати:

- Об’єкт (продукт)
- Колекцію об’єктів (ProductStorage)

Зчитуванням колекції можна назвати послідовне зчитування її елементів та збереження їх у цій колекції, тож варто спочатку реалізувати зчитування об’єктів.

Зчитування об’єкту

Дані про наші об’єкти будуть зберігатися в одній лінії, тож створимо IStreamLineReader<T>

Методи:

```
bool TryReadLine<G>(out G obj, StreamReader stream, IStringParser<G> validator)
where G : T, new();
```

Цей метод може використовуватися для типів даних де T може бути як інтерфейсом так і реалізацією, а G повинен наслідувати T та мати можливість створення об’єкта.

Таким чином у obj можна передати тільки конкретну реалізацію типу T і це задовольняє умову out

Також цей метод приймає поток звідки буде читати та IStringParser<G>.

IStringParser<T>:

Методи: T Parse(string str); цей метод приймає строку яку треба розпарсити та повертає значення T

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

LoggerOnBadFormat:

delegate void LoggerOnBadFormat(string message)

Тепер можна перейти до реалізацій цих інтерфейсів

TXTSerializedProductReader<T>: Де T наслідує IProduct

Наслідує: IStreamLineReader<T>

Метод: `TryReadLine<G>(out G obj, StreamReader stream, IStringParser<G> validator)`

Цей клас надає можливість зчитувати дані з потоку до об’єкту типу T за логікою переданою у вигляді IStringParser<G>

Зчитування колекцій

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

Метод:

```
void ReadCollection(out 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

Інтерфейс для парсерів був описаний раніше, але ось ще раз:

IStringParser<T>:

Методи: T Parse(string str); цей метод приймає строку яку треба розпарсити та повертає значення T

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

LoggerOnBadFormat:

delegate void LoggerOnBadFormat(string message)

Логіка роботи парсерів:

- Знайти усі записи параметрів
- Валідувати значення
- Повернути модель, або null при помилках під час читання
- (під час кожного кроку при помилках створювати строку логів та в кінці за допомогою події обробити її)

Проблема інваріантності

А ще кожен парсер наслідує абстрактний клас

ProductParserBase:

реалізує IStringParser<IProduct>

Та реалізує IStringParser<IDairyProduct> < (IDairyProduct для прикладу)

Тобто інтерфейс з конктретним інтерфейсом, що буде повертати парсер

А завдяки наслідування ProductParserBase з його реалізацією IStringParser<IProduct> вирішується проблема інваріантності (але хотілося б запитати про це на занятті, бо часу на цю проблему було витрачено дуже багато, а повного розуміння так і нема)

Обробка помилок

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 класс

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

На цей метод будуть підписуватися події парсерів.