Zadanie 1 - Kolekcje, testy jednostkowe, Dependency Injection

Cel

- wykorzystanie kolekcji platformy .NET w przykładowym zastosowaniu:
    - katalogowanie, np. katalog produktów, wykaz klientów,
    - rejestracja zdarzeń, np. lista faktur,
    - opis stanu, np. stan biblioteki, stan magazynu.
- definiowanie API dla biblioteki (publiczne deklaracje)
- użycie Dependency Injection, gdzie decyzja o wyborze zachowania jest odroczona do czasu realizacji programu

Część 1

a) Utwórz projekt C# biblioteki klas, zawierający wzajemnie powiązane klasy danych, reprezentujące wybrany proces biznesowy, np. bibliotekę, sklep internetowy, magazyn, itp.

- Klasa `Wykaz` ma reprezentować elementy wykazu z danymi opisującymi osoby (jak: czytelnicy, klienci).
- Klasa `Katalog` ma opisywać pozycje w słowniku (jak: opisy książek, opisy produktów), przy czym:
    - Konieczne będzie tu określenie klucza używanego do dostępu do danych (`string`, `int`, itp).
    - Nie należy dodawać do tej klasy właściwości typu "CzyWypozyczona", "KtoWypozyczyl", "IloscProduktow", "Cena",
      czyli informacji, które będzie można uzyskać z innych struktur danych.
- Klasa `OpisStanu` ma opisywać wystąpienia odnoszące się do pozycji słownikowych (jak: egzemplarz książki - opis książki, data zakupu; stan magazynu - produkt, jego ilość, cena netto, stawka podatku).
- Klasa `Zdarzenie` ma opisywać relacje wiążące osoby oraz wystąpienia odnoszące się do pozycji słownikowych (jak: wypożyczenia - kto, który egzemplarz, data wypożyczenia, data zwrotu; faktura - kto, kiedy, który stan magazynowy produktu, ilość, cena i stawka dla produktu).

Inne klasy mogą zostać wprowadzone w celu normalizacji struktury danych (autorzy książek, pozycje faktur, sprzedawcy wystawiający rachunki, magazynierzy realizujący operacje magazynowe).

b) Utwórz klasę gromadzącą obiekty z danymi

Jeśli nie masz lepszego pomysłu, proponowana jest nazwa `DataContext`. Jej pola mogą być publiczne, gdyż obiekt tej klasy nie będzie bezpośrednio dostępny w innych częściach systemu. Przeznaczeniem tej klasy

jest tylko gromadzenie danych, bez dalszych operacji na nich. Pozwala to traktować ją jako zastępnik bazy danych, lub dokument w pamięci przechowujący w jednym obiekcie wszystkie dane systemu. Przy okazji uzyskuje się obiekt, który można później serializować i deserializować, wygodnie realizując operacje np. zapisu dokumentu do pliku i ponowne odczytanie dokumentu z pliku.

Pola klasy gromadzącej obiekty z danymi zadeklaruj w postaci kolekcji:

- Dane z informacjami o elementach wykazu należy przechowywać w obiekcie `List`.
- Pozycje słownikowe należy przechowywać w obiekcie `Dictionary<Klucz, Katalog>`.
- Zdarzenia łączące opisy stanu z elementami wykazu należy przechowywać o obiekcie klasy `ObservableCollection`.
- Opisy stanu można przechowywać w obiektach dowolnej z powyższych kolekcji, np. `List` lub `ObservableCollection`.

d) Dodaj klasę zarządzającą obiektami danych

Jeśli nie masz lepszego pomysłu, proponowana jest nazwa `DataRepository`. Dodaj w niej pole prywatne typu `DataContext`. Klasa zarządzająca obiektami danych będzie rozbudowana w dalszej części zadania.

Część 2

Należy przygotować API do wypełniania kolekcji przykładowymi danymi. Można to zrealizować wprowadzając klasę abstrakcyjną (*abstract class*) lub interfejs (*interface*). Różne implementacje tego abstrakcyjnego typu będą później wykorzystane w dalszych częściach zadania (poniżej).

Użycie konkretnej implementacji ma być realizowane na zasadzie wzorca *Wstrzykiwania Zależności (Dependency Injection, DI)*, gdzie odpowiedzialność za kontrolę wybranych czynności przenoszona jest na zewnątrz obiektu a wyboru dokonuje się w trakcie realizacji programu.

Celem zastosowania wspomnianego wzorca jest aby klasa przechowująca dane nie decydowała o typie używanego obiektu (np. jednej, wybranej klasy `Wypelnianie{...}`) i nie tworzyła go samodzielnie w konstruktorze, tylko żeby decydował o tym kod tworzący i konfigurujący obiekty aplikacji. Taki sposób zarządzania zależnościami między obiektami pozwala na pozbycie się ścisłych zależności pomiędzy częściami systemu. Pozwala to także łatwiej (lub w ogóle) testować części niezależnie, w oderwaniu od całości systemu.

Do przejrzenia:

- Wikipedia (EN + przykłady): Dependency Injection, Wikipedia (PL): Wstrzykiwanie zależności
- Przykład: TP / DependencyInjection
- Źródło: Inversion of Control Containers and the Dependency Injection pattern, Martin Fowler
- Źródło: Dependency Injection in .NET, Mark Seemann

Rozwiązania *DI* oparte są o przekazywanie z zewnątrz do wybranego obiektu innych obiektów, których używa on do realizacji swoich zadań.
Typowe sposoby realizacji *DI* to:

- Constructor Injection - obiekt przekazywany jest jako wymagany parametr konstruktora klasy,

- Method Injection - obiekt przekazywany jest jako parametr wywołania metody,
- Setter Injection / Property Injection - obiekt wpisywany jest do pola lub właściwości klasy,
- Dependency Injection Container - technologie wykorzystujące powyższe sposoby realizacji, gdzie specyfikację zależności określa się poprzez atrybuty lub konfigurację - a realizację zależności wykonują biblioteki przy uruchamianiu aplikacji.
  Przykłady: Castle Windsor, StructureMap, Ninject, Spring.NET, Autofac, Unity, Managed Extensibility Framework (MEF), Prism Library.

Alternatywne do *DI* wzorce - patrz np. Factories, Service Locators, and Dependency Injection:

- Factory Method, Abstract Factory, Simple Factory - zamiast używać słowa kluczowego `new`, tworzenie obiektu deleguje się do fabryki obiektów,
- Service Locator - klasy usług są rejestrowane przy starcie, a następnie wyszukiwane podając potrzebny typ obiektu (jest uważany czasem jako *antywzorzec*).

Wstrzykiwanie zależności

W celu prostej realizacji *DI* można dodać w kodzie klasy przechowującej dane konstruktor z parametrem określającym typ żądanego interfejsu (lub klasy abstrakcyjnej) i zapamiętywać wskazany obiekt w polu prywatnym do dalszego użytku.

Alternatywnie można utworzyć w klasie właściwość *(property)* żądanego interfejsu (lub klasy abstrakcyjnej), oraz ustawiać wartość tej właściwości po utworzeniu obiektu klasy przechowującej dane.

d) Dodaj typ abstrakcyjny definiujący API pozwalające wypełniać kolekcje danymi

- W klasie `DataRepository` dodaj konstruktor z odpowiednim parametrem lub dodaj właściwość (property, z sekcją `set`, ewentualnie `private get`).
- Przygotuj klasę np. `WypelnianieStalymi`, która będzie umieszczać w każdej kolekcji stałą, niewielką ilość obiektów o ustalonych wartościach.
- Dodaj kod konfigurujący komponenty aplikacji przed uruchomieniem, który będzie przekazywać do klasy `DataRepository` obiekt klasy `WypelnianieStalymi`.
- Klasa `DataRepository` powinna wykorzystać przekazany obiekt - w swoim konstruktorze lub w sekcji `set` właściwości - aby wypełnić kolekcje zawarte w `DataContext` przykładowymi danymi.

Część 3

e) Rozbuduj klasę zarządzającą obiektami danych

W klasie `DataRepository` zaimplementuj zbiór metod typu C.R.U.D. (Create, Read, Update, Delete) do obsługi obiektów danych, dla każdej kolekcji z klasy `DataContext`:

- (*Add*) Dodawanie nowych danych do kolekcji
- (*Get*) Odczyt pojedynczych obiektów, np. na podstawie identyfikatora lub pozycji w kolekcji
- (*GetAll*) Odczyt wszystkich obiektów z kolekcji
- (*Update*) Aktualizacja danych w kolekcji - opcjonalnie, podając obiekt lub pozycję w kolekcji
- (*Delete*) Usuwanie wskazanych danych z kolekcji - podając obiekt lub pozycję w kolekcji

Do każdej metody powinny być napisane testy jednostkowe sprawdzające poprawność jej działania (!)

Część 4

f) Utwórz klasę realizującą logikę działania aplikacji

Jeśli nie masz lepszego pomysłu, proponowana jest nazwa `DataService`.

Zapewnij na zasadzie *DI* przekazanie do klasy logiki aplikacji obiektu klasy zarządzającej danymi `DataRepository`.
Upewnij się, że przekazany obiekt będzie zapamiętany w polu prywatnym klasy (aby używać tego obiektu później tylko z poziomu klasy logiki aplikacji).

g) Wprowadź mechanizmy przetwarzania przechowywanych danych:

- Wyświetlanie danych z przekazanej jako parametr kolekcji (pozycje katalogu, opisy stanu, elementy wykazu, zdarzenia)
- Wyświetlanie danych przekazanej kolekcji w postaci powiązanej, to znaczy:
    - zaczynając od elementów wykazu (np. czytelnicy, klienci),
    - za nimi zdarzenia odpowiadające kolejnym elementom wykazu (np. wypożyczenia książek, faktury),
    - które przechodząc przez opisy stanu będą indentyfikować pozycje katalogu (np. wypożyczone książki, zakupione towary).
- Operacje modyfikacji danych (dodaj zdarzenie na podstawie elementu wykazu i opisu stanu, usuń, itp.)
    - niektóre operacje będą wprost przekazywane do `DataRepository`
    - niektóre będą wymagać dodatkowych operacji (tworzenie obiektów, wyszukiwanie obiektów, itp.)
- Filtrowanie lub wyszukiwanie danych (zwracanie obiektów spełniających założone kryterium).

h) Dodaj obsługę zdarzeń generowanych przez ObservableCollection, z wyświetlaniem informacji zmianie jej zawartości (dodawanie do niej i usuwanie z niej obiektów).

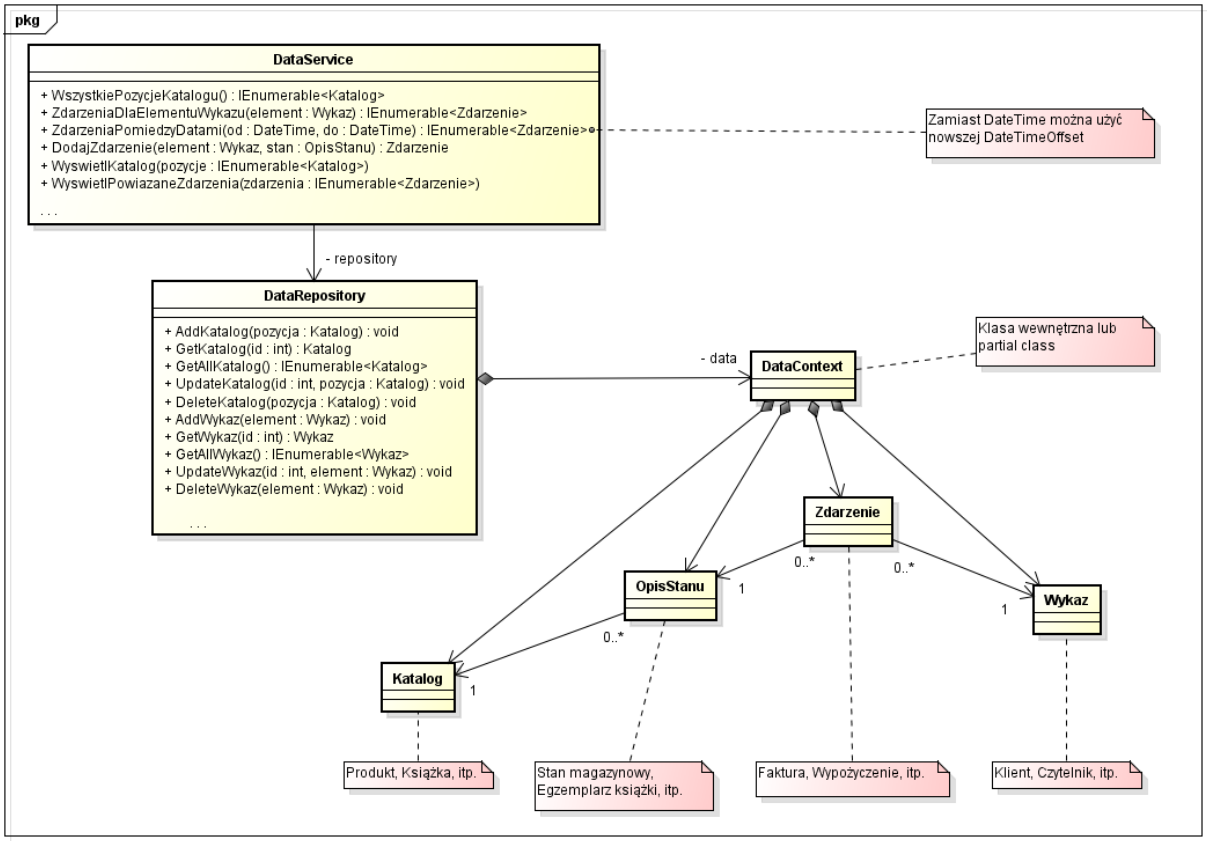
Część 5

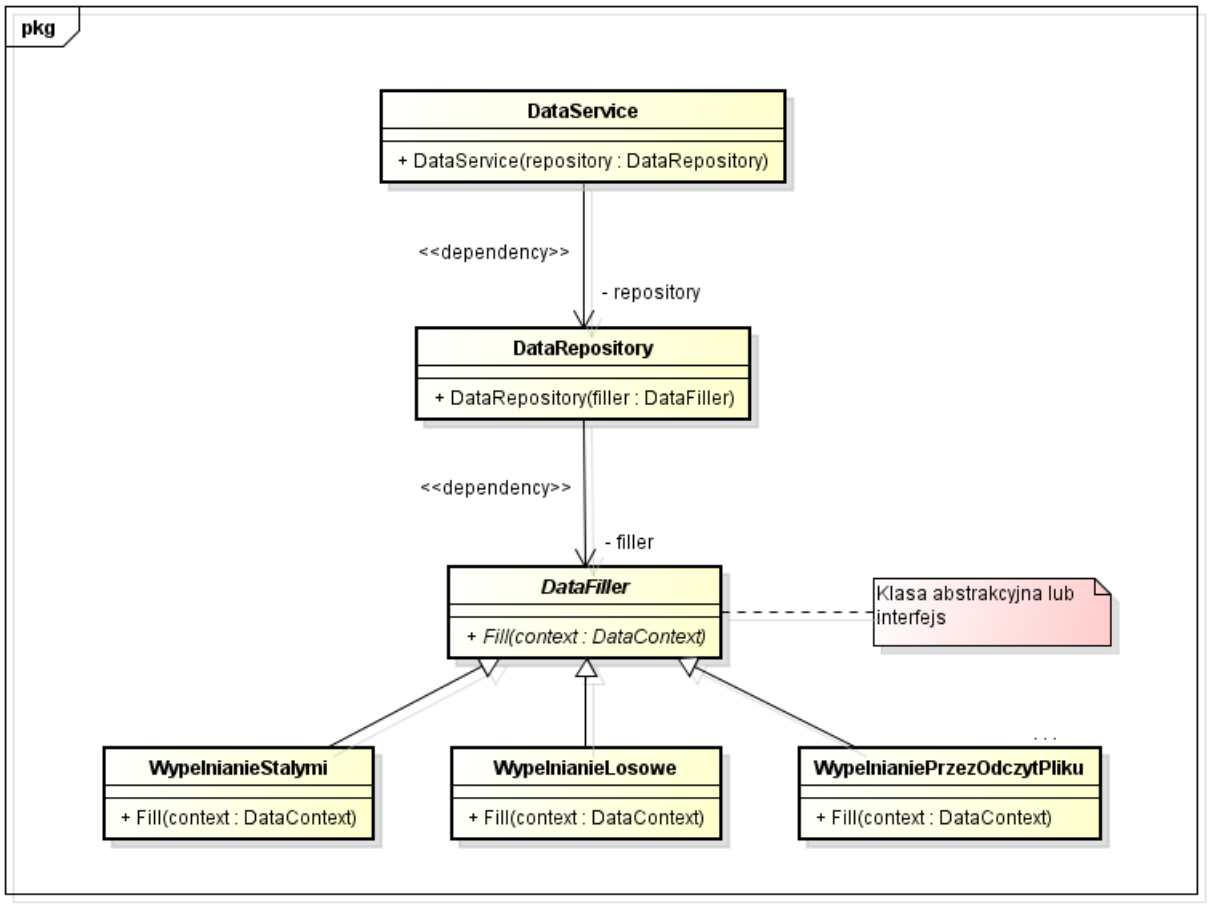
i) Utwórz inną implementację wypełniania kolekcji danymi i wykorzystaj ją do testów wydajności

- Kolejna implementacja wypełniania kolekcji danymi powinna umieszczać w kolekcjach obiekty w inny sposób niż poprzednia - opisana w punkcie **d)**.

  Jednak nadal powinno to być polimorficznie zgodne z ustalonym poprzednio API, czyli podmiana implementacji powinna wymagać minimalnych zmian w aplikacji.
- Przykładowo może to być zrealizowane poprzez:
  - odczyt danych z przygotowanego pliku tekstowego,
  - deserializację danych w formacie JSON, XML, itp.
  - odczyt do pamięci i zamiana na obiekty informacji z bazy danych SQL,
  - tworzenie obiektów o losowych wartościach.
- Sprawdź poprawność rozwiązania - tworząc testy jednostkowe.
- Sprawdź wydajność rozwiązania - np. tworząc testy jednostkowe pod kątem wydajności i wypełniając przy tym kolekcje dużą ilością losowych danych (dziesiątki i setki tysięcy obiektów). Porównaj działanie w kilku punktach odniesienia, np. przy logarytmicznie rosnącej ilości obiektów.

Diagramy UML




pkg
DataService
+ DataService(repository : DataRepository)
<<dependency>>
- repository
DataRepository
+ DataRepository(filler : DataFiller)
<<dependency>>
- filler
DataFiller
+ Fill(context : DataContext)
Klasa abstrakcyjna lub interfejs
WypelnianieStalymi
+ Fill(context : DataContext)
WypelnianieLosowe
+ Fill(context : DataContext)
WypelnianiePrzezOdczytPliku
+ Fill(context : DataContext)