# Projektowanie obiektowe oprogramowania
# Wzorce architektury aplikacji (3)
# Wykład 11 – Repository, Unit of Work
# Wiktor Zychla 2014

**Repository** – dodatkowa warstwa izolująca obiektową warstwę dostępu do danych. Repository działa na poziomie jednej klasy modelu

**Unit of Work** – ułatwia korzystanie z repository dając dostęp do wszystkich repositories z jednego miejsca. Dodatkowo bierze na siebie zarządzanie transakcjami.

## 1 Repository

Zalety wprowadzenia repozytorium jako warstwy izolującej dostęp do danych:

- Uniezależnienie warstwy przetwarzania danych (logika biznesowa) od implementacji warstwy dostępu do danych – wraz ze zmieniającymi się technologiami można łatwo dostarczać nowych, wydajniejszych implementacji repository, używających zupełnie innych technologii dostępu do danych (niekoniecznie nawet relacyjnych baz danych!)
- Umożliwienie łatwego zastępowania implementacji repozytorium – testy jednostkowe warstw przyległych bez efektów ubocznych dzięki implementacjom typu fake/stub.

Wady wprowadzenia repozytorium:

- Dodatkowa warstwa w architekturze aplikacji
- Kontrowersje wokół wzorcowego interfejsu jaki powinno implementować repozytorium (**generic repository** vs **concrete repository**?)
- Jeżeli technologia ORM sama z siebie jest już „repository”, to dodatkowe opakowywanie repository w inne repository może być dyskusyjne

Dla zadanych klas modelu

```
public class User { }

public class Account { }
```

przykładowy interfejs tzw. „generycznego repozytorium” (**generic repository**):

```
public interface GenericRepository<T>
{
    T New();

    void Insert( T item );
```

```
        void Update( T item );
        void Delete( T item );

        IQueryable<T> Query { get; }
    }
```

i jego implementacja dla jednej z klas:

```
    public class UserRepository : GenericRepository<User>
    {
        T GenericRepository<User>.New()
        {
                ...
        }

        void GenericRepository<User>.Insert( User item )
        {
                ...
        }

        void GenericRepository<User>.Update( User item )
        {
                ...
        }

        void GenericRepository<User>.Delete( User item )
        {
                ...
        }

        IQueryable<User> GenericRepository<User>.Query
        {
            get {  }
        }
    }
```

Alternatywą dla generic repository jest „**concrete repository**”:

```
   public interface ConcreteUserRepository
   {
     IEnumerable<User> RetrieveAllUsers();
     User RetrieveSingle( int Id );

     IEnumerable<User> FindAllUsersForStartingLetter( string FirstSurnameLetter );

     ...

     User New();

     void Insert( User item );
     void Update( User item );
     void Delete( User item );
    }
```

Porównanie „generic repository” i „concrete repository”:

**Concrete Repository**

- Wymaga się tylu **różnych** konkretnych interfejsów repozytoriów, ile jest klas w modelu dziedzinowym
- Każdy interfejs udostępnia metody dostępu do danych specyficzne dla konkretnego typu (na przykład użytkowników będziemy wyszukiwać według rozbudowanych kryteriów (i każde kryterium może być osobną metodą w kontrakcie repozytorium)
- Zaprojektowanie i utrzymanie interfejsów repozytoriów w dużym projekcie jest trudne i żmudne

**Generic Repository**

- Jest tylko jeden, wspólny, generyczny interfejs repozytorium, który ma wiele implementacji – jest to możliwe dlatego, że wszystkie możliwe skomplikowane warianty zapytań zamyka tu kontrakt Linq (czyli zwracanie klientowi obiektu implementującego **IQueryable**)
- Problemem generycznego repozytorium jest to, że różne technologie w różny sposób implementują Linq, w szczególności wyrażenia mogą być poprawnie interpretowane w pewnych implementacjach a w innych nie. I nagle klient, który dostaje w kontrakcie zapewnienie że repozytorium dostarcza obiektu **IQueryable**, może się przekonać że w rzeczywistości dostarczona mu implementacja A jest zbyt uboga żeby wykonać konkretne zapytanie, gdy tymczasem implementacja B radzi sobie z tym dobrze. To łamie zasadę, w której to dostawca implementacji musi odpowiadać za realizację kontraktu, a nie klient być zmuszonym do posiadania wiedzy o stanie implementacji kontraktu przez różne możliwe implementacje.

# 2 Unit of Work

Unit of Work wprowadza interfejs dla klienta::

```
public interface IUnitOfWork
{
    GenericRepository<IUser> UserRepository { get; }
    GenericRepository<IAddress> AddressRepository { get; }

    void SaveChanges();
}
```

Klient zawsze korzysta z instancji Unit of Work tworząc izolowany kontekst dostępu do danych. W ramach jednego Unit of Work poszczególne repozytoria współdzielą ten sam kontekst.

Interfejs Unit Of Work może dodatkowo przewidywać m.in. zarządzanie transakcjami czy metadanymi bazy danych.

# 3 Abstrakcje klas modeli

Podczas wykładu zobaczymy przykład na żywo budowania warstwy repozytorium dla dwóch przykładowych technologii mapowania obiektowo-relacyjnego: **Linq2SQL** i **Entity Framework**. Zasadnicza różnica między tymi technologiami polega na sposobie implementacji klas modeli (patrz notatki do wykładu o mapowaniu obiektowo-relacyjnym) i właściwości nawigacyjnych w nich (**navigation properties**):

- w przypadku **Linq2SQL** mamy do czynienia z implementacją typu **Value Holder**. Model jest generowany przez automat, klasy zawierają wygenerowany kod właściwości nawigacyjnych, którego nie można modyfikować
- w przypadku **EF** mamy do czynienia z implementacją typu **Virtual Proxy**. Model jest budowany ręcznie i oparty na klasach typu POCO.

Wymaganie jakie sobie stawiamy jest takie, że użyjemy kontenera **IoC** do konfiguracji wybranej implementacji i chcemy aby kod klienta (warstwy logiki biznesowej) w ogóle nie zmieniał się przy wymianie warstwy dostępu do danych.

Podstawowa trudność jaka pojawia się przy takim wymaganiu związana jest właśnie ze sposobem implementacji właściwości nawigacyjnych. Skoro raz właściwości nawigacyjne implementuje automat i są one jawnie zaimplementowane w kodzie (Linq2SQL) a innym razem implementuje je generator proxy, to oznacza że potrzeba **dwóch** różnych typów modeli, każdemu z podejść odpowiada bowiem inny model.

A to uniemożliwia spełnienie wymagania o niemodyfikowaniu kodu klienckiego przy wymianie repozytorium na inne repozytorium.

Jak rozwiązać ten problem? Literatura sugeruje żeby posłużyć się wzorcem **ViewModel** (patrz Mark Seemann, „Dependency Injection in .NET"). W podejściu tym mamy jeden uniwersalny model, niezwiązany z modelem utrwalanym (**persistence model**) – czyli pozwalamy każdej technologii mapowania obiektowo relacyjnego mieć swój własny zestaw klas modelu, a dodatkowo mamy ten jeden uniwersalny model i konwersje z niego do każdego z konkretnych modeli utrwalanych.

Okazuje się, że to nie jest dobry pomysł z uwagi na brak możliwości implementacji leniwych zależności typu parent-child w klasach tego uniwersalnego modelu. Wydaje się to również nieefektywne z uwagi na konieczność ciągłych konwersji z i do modelu uniwersalnego.

Na wykładzie pokażemy, że lepszym podejściem jest opisanie modelu przez interfejsy klas, a następnie implementacji repozytoriów względem interfejsów klas modelu dziedzinowego.

Inne podejście to ograniczenie się wyłącznie do tych implementacji technologii dostępu do danych, które potrafią pracować na wskazanym modelu obiektowym typu **POCO/POJO** (**Code First**).

# 4 Organizacja struktury repozytoriów i units of work

W świetle powyższych rozważań, właściwa struktura projektu powinna wyglądać następująco:

- Klasy modeli – jeden zbiór abstrakcji (interfejsów) opisujący modele (**IUser**, **IAddress**).
- Wiele zestawów implementujących abstrakcje modeli, dla każdego typu repozytorium inna implementacja (**EFUser**, **EFAddress**, **LinqUser**, **LinqAddress**)
- Jeden zestaw abstrakcji repozytoriów – w zależności od wyboru jeden generyczny interfejs (**IGenericRepository<T>**) lub interfejsy szczegółowe (**IUserRepository**, **IAddressRepository**)
- Wiele implementacji abstrakcji repozytoriów, dla każdego typu repozytorium inna implementacja (**EFUserRepository**, **EFAddressRepository**, **LinqUserRepository**, **LinqAddressRepository**)

- Jedna abstrakcja Unit of Work
- Wiele implementacji abstrakcji Unit of Work, dla każdego typu UoW inna implementacja (**EFUnitOfWork**, **LinqUnitOfWork**)
- Użycie kontenera IoC do zarządzania mapowaniem abstrakcji na implementacje