Projektowanie obiektowe oprogramowania
Wzorce architektury aplikacji (3)
Wykład 11 – Repository, Unit of Work
Wiktor Zychla 2019

Repository – dodatkowa warstwa abstrakcji na obiektową warstwę dostępu do danych. Zwykle Repository kontroluje dostęp do jednej „kategorii" danych (np. jednej tabeli z bazy danych)

Unit of Work – kompozyt wielu repozytoriów – zarządza ich czasem życia i pozwala na dostęp do nich z jednego miejsca. Dodatkowo bierze na siebie m.in. zarządzanie transakcjami.

1 Repository

Zalety wprowadzenia repozytorium jako warstwy abstrakcji:

- 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**! – można wyobrazić sobie implementacje Repository które odwołują się do *baz nierelacyjnych*, do *usług katalogowych*, są implementacje które do baz relacyjnych dostają się za pomocą jakiegoś silnika ORM i są też takie które używają niskopoziomowego interfejsu typu ADO.NET do dostępu do danych)
- Umożliwienie łatwego zastępowania implementacji repozytorium – testy jednostkowe warstw przyległych bez efektów ubocznych dzięki implementacjom typu fake/stub.

Wady wprowadzenia repozytorium jako warstwy abstrakcji:

- Dodatkowa warstwa w architekturze aplikacji
- Wątpliwości odnośnie interfejsu jaki powinno implementować repozytorium (**generic repository** czy **concrete repository**?)
- Jeżeli technologia ORM sama z siebie jest już zaprojektowana według wzorców Repo/UoW, to dodatkowe jej opakowywanie może być dyskusyjne (por. m.in. https://stackoverflow.com/a/19059282/941240)

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**) (w innych technologiach zapytań mógłby to być inny uniwersalny interfejs zapytań np. **JPA, HQL** itp).
- Problemem generycznego repozytorium jest to, że różne technologie w różny sposób implementują uniwersalne języki zapytań (LINQ/JPA/…), w szczególności wyrażenia mogą być poprawnie interpretowane w pewnych implementacjach a w innych nie. I nagle klient może się przekonać że 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 jest kompozytem wielu repozytoriów:

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

    void SaveChanges();

    // opcjonalne
    void BeginTransaction();
    void CommitTransaction();
    void RollbackTransaction();
}
```

Można powiedzieć w uproszczeniu, że o ile Repository jest abstrakcją dostępu do pojedynczej tabeli w bazie relacyjnej, to Unit of Work jest dostępem do wszystkich repozytoriów z jednego miejsca, czyli do wszystkich tabel.

Klient zawsze korzysta z instancji Unit of Work tworząc sesję dostępu do danych. W ramach jednego Unit of Work poszczególne repozytoria współdzielą ten sam kontekst (czyli jakieś niskopoziomowe szczegóły implementacji, połączenie do bazy danych, uchwyty itp.)

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

Uwaga! Bywa, że wzorzec Repository jest opisywany w oderwaniu od Unit of Work, a co za tym idzie – tak też bywa implementowany (repozytoria bez Unit of Work). Jest to podejście błędne i rodzi niepotrzebne trudności przy przekazywaniu repozytoriów do wyższych warstw aplikacji.

3 Abstrakcje klas modeli

Podczas wykładu zobaczymy przykład na żywo budowania warstwy repozytorium dla dwóch przykładowych technologii mapowania obiektowo-relacyjnego: **Linq2SQL** i **Entity Framework**.

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

Formalniej: klient będzie chciał pisać taki uniwersalny kod, w którym konkretna implementacja będzie ukryta za fabryką:

```
var uow   = new UowFactory().Create();
var users = uow.Users.Where( u => u.Name == "foo" );

var user  = users.FirstOrDefault();
```

Jakiego typu jest `user`?

Zapewne gdzieś w projekcie jest

```
public class User
{
    ...
}
```

I teraz – jeśli taka klasa modelu ma współpracować z konkretnym dostawcą implementacji, to musi za każdym razem spełniać inne wymagania.

Niestety, tu: zasadnicza różnica między tymi dwiema technologiami polega na sposobie implementacji właściwości nawigacyjnych w klasach modeli (**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.

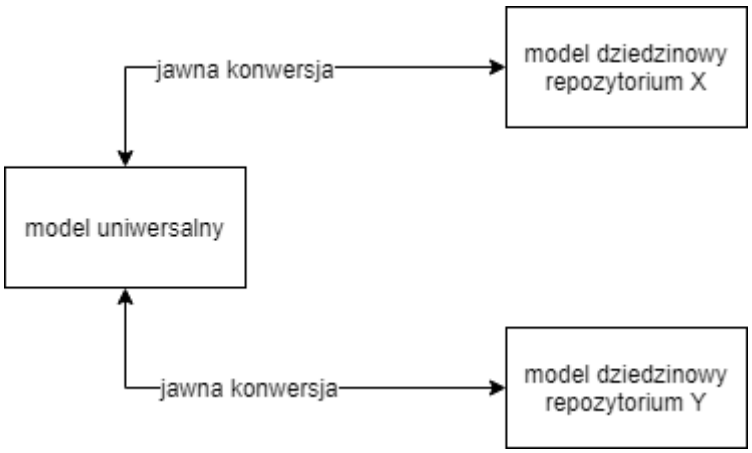
Skoro raz właściwości nawigacyjne implementuje automat i są one jawnie zaimplementowane w kodzie (Linq2SQL) a innym razem implementuje je generator proxy, to oznacza że tych dwóch różnych wymagań nie da się spełnić jedną i tą samą klasą modelu - potrzeba **dwóch** różnych typów modeli, każdemu z podejść odpowiada bowiem inny model.

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

Jak rozwiązać ten problem?

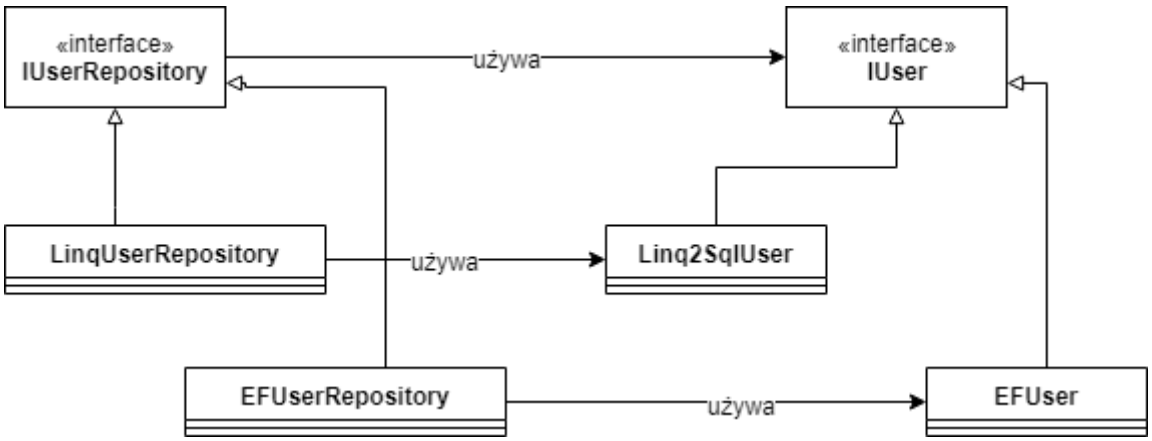
Literatura sugeruje żeby posłużyć się wzorcem **ViewModel** (patrz Mark Seemann, „Dependency Injection in .NET"). W podejściu tym mamy jeden *uniwersalny* model, niezwiązany z modelem utrwalanym (**persistence model**) – czyli pozwalamy każdej implementacji repozytorium mieć swój

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 łatwej implementacji leniwych właściwości nawigacyjnych w klasach takiego 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**. Klasy POCO/POJO to klasy, które

- nie dziedziczą z żadnej dodatkowej, pomocniczej klasy
- ich właściwości nawigacyjnie nie mają żadnej implementacji (poza oznakowaniem `virtual`)

Spośród wszystkich możliwych implementacji repozytorium odrzucilibyśmy więc takie, które nie spełniają warunku zgodności z takimi klasami modeli.

4 Organizacja struktury repozytoriów i units of work

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

Pakiet/zestaw określający abstrakcje (kontrakty) dla kodu klienckiego:

- Klasy modeli – jeden zbiór abstrakcji (interfejsów) opisujący modele (**IUser**, **IAddress**).

```
public interface IParent
{
    int ID { get; set; }
    string ParentName { get; set; }
    IEnumerable<IChild> Children { get; set; }
}

public interface IChild
{
    int ID { get; set; }
    int ID_PARENT { get; set; }
    string ChildName { get; set; }
    IParent Parent { get; set; }
}
```

- Jeden zestaw abstrakcji repozytoriów – w zależności od wyboru jeden generyczny interfejs (**IGenericRepository<T>**) lub interfejsy szczegółowe (**IUserRepository**, **IAddressRepository**)
- Jedna abstrakcja **Unit of Work**

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

      void Insert(T t);
      void Update(T t);
      void Delete(T t);

      IQueryable<T> Query { get; }
  }

  public interface IUnitOfWork
  {
      IGenericRepository<IParent> Parent { get; }
      IGenericRepository<IChild> Child { get; }
  }
```

- **Local Factory** jako API dostępu do instancji UoW

```
public class UnitOfWorkFactory
{
    private static Func<IUnitOfWork> _provider;

    public static void SetProvider( Func<IUnitOfWork> provider )
    {
        _provider = provider;
```

```
    }

    public IUnitOfWork Create()
    {
        return _provider();
    }
}
```

Kod kliencki

- Referencja wyłącznie do zestawu określającego abstrakcje i Local Factory – tyle wystarczy żeby pisać kod kliencki

```
static void Main(string[] args)
 {
     // referencja do uow zawsze Local Factory
     var uow = new UnitOfWorkFactory().Create();

     // dodanie elementu
     var parent = uow.Parent.New();
     parent.ParentName = "nowy parent";
     uow.Parent.Insert(parent);

     // zapytanie
     foreach ( var child in uow.Child.Query )
     {
         Console.WriteLine(child.ChildName);
     }

     Console.ReadLine();
 }
```

Pakiety z implementacjami (kod kliencki nie używa ich wprost, są konfigurowane przez Local Factory z Composition Root, tak jak widzieliśmy to na poprzednim wykładzie)

- Wiele zestawów implementujących abstrakcje modeli, dla każdego typu repozytorium inna implementacja (**EFUser**, **EFAddress**, **LinqUser**, **LinqAddress**)
- Wiele implementacji abstrakcji repozytoriów, dla każdego typu repozytorium inna implementacja (**EFUserRepository**, **EFAddressRepository**, **LinqUserRepository**, **LinqAddressRepository**)
- Wiele implementacji abstrakcji Unit of Work, dla każdego typu UoW inna implementacja (**EFUnitOfWork**, **LinqUnitOfWork**)

Na przykład dla Linq2Sql implementacja jest dość oczywista, jedyne trudniejsze miejsce polega na tym, że automatycznie wygenerowane klasy modeli **nie implementują** interfejsów opisujących abstrakcje modelu (**IChild, IParent**), dlatego trzeba dodatkowym kodem „podpowiedzieć” że jednak implementują.

```
public class Linq2SqlChildRepository : IGenericRepository<IChild>
 {
     private ParentChildDataContext _context { get; set; }
     public Linq2SqlChildRepository( ParentChildDataContext context )
     {
         this._context = context;
```

```
    }

    public IQueryable<IChild> Query
    {
        get
        {
            return this._context.Childs;
        }
    }

    public void Delete(IChild t)
    {
        this._context.Childs.DeleteOnSubmit(t as Child);
        this._context.SubmitChanges();
    }

    public void Insert(IChild t)
    {
        this._context.Childs.InsertOnSubmit(t as Child);
        this._context.SubmitChanges();
    }

    public IChild New()
    {
        return new Child();
    }

    public void Update(IChild t)
    {
        this._context.SubmitChanges();
    }
}

public class Linq2SqlParentRepository : IGenericRepository<IParent>
{
    private ParentChildDataContext _context { get; set; }
    public Linq2SqlParentRepository(ParentChildDataContext context)
    {
        this._context = context;
    }

    public IQueryable<IParent> Query
    {
        get
        {
            throw new NotImplementedException();
        }
    }

    public void Delete(IParent t)
    {
        this._context.Parents.DeleteOnSubmit(t as Parent);
        this._context.SubmitChanges();
    }

    public void Insert(IParent t)
    {
        this._context.Parents.InsertOnSubmit(t as Parent);
        this._context.SubmitChanges();
```

```
    }

    public IParent New()
    {
        return new Parent();
    }

    public void Update(IParent t)
    {
        this._context.SubmitChanges();
    }
}

public class Linq2SqlUnitOfWork : IUnitOfWork
{
    private ParentChildDataContext _context { get; set; }

    public Linq2SqlUnitOfWork( ParentChildDataContext context )
    {
        this._context = context;
    }

    public IGenericRepository<IChild> Child
    {
        get
        {
            return new Linq2SqlChildRepository(this._context);
        }
    }

    public IGenericRepository<IParent> Parent
    {
        get
        {
            return new Linq2SqlParentRepository(this._context);
        }
    }

    public void Dispose()
    {
        this._context.Dispose();
    }
}

// dodatkowy kod w którym dodaje się implementację abstrakcji modeli
// do implementacji wygenerowanej przez automat
public partial class Child : IChild
{
    IParent IChild.Parent
    {
        get
        {
            return this.Parent;
        }

        set
        {
            this.Parent = (Parent)value;
        }
```

```
    }
}

public partial class Parent : IParent
{
    public IEnumerable<IChild> Children
    {
        get
        {
            return this.Children;
        }

        set
        {
            this.Children = value;
        }
    }
}
```

Główny moduł aplikacji

- Composition Root – konfiguracja dostawcy dla **Local Factory**, czyli metody fabrykującej która będzie wykonywać się za każdym razem kiedy klient zawoła **Create**

```
private static void CompositionRoot()
{

    UnitOfWorkFactory.SetProvider(() =>
   {
       var connectionString = ConfigurationManager.AppSettings["cs"];
       var context = new ParentChildDataContext(connectionString);

       return new Linq2SqlUnitOfWork(context);
   });
}
```

- (opcjonalne) Użycie kontenera IoC do zarządzania mapowaniem abstrakcji na implementacje – w sensie: provider dla Local Factory który używa IoC