Projektowanie obiektowe oprogramowania
Wzorce architektury aplikacji (4)
Wykład 12 – Repository, Unit of Work
Wiktor Zychla 2013

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.

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
- 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>
{
    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>
{
    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 );

  ...

  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.

Podczas wykładu zobaczymy przykład na żywo budowania warstwy repozytorium dla trzech przykładowych technologii mapowania obiektowo-relacyjnego : **Linq2SQL**, **Entity Framework** i **NHibernate**. Użyjemy kontenera **IoC** do konfiguracji wybranej implementacji i zobaczymy, że kod klienta (warstwy logiki biznesowej) może w ogóle nie zmieniać się przy wymianie warstwy dostępu do danych.

Podstawowa trudność jaka pojawia się przy takiej implementacji polega na tym, że niektóre technologie mapowania OR tworzą własne klasy modelu dziedzinowego. Z kolei wymienne repozytorium nie może więc zależeć od żadnej konkretnej implementacji klas modelu dziedzinowego.

Jak rozwiązać ten problem? Literatura sugeruje żeby posłużyć się wzorcem **ViewModel** (patrz Mark Seemann, „Dependency Injection in .NET"), jednak to nie jest dobry pomysł z uwagi na brak możliwości implementacji leniwych zależności typu parent-child w klasach modelu. Wydaje się to również nieefektywne.

Na wykładzie pokażemy, że jedno z możliwych rozwiązań polega na opisaniu 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**).