Projektowanie aplikacji bazodanowych w .NET

Wykład 4

Paweł Rajba

Instytut Informatyki

Uniwersytet Wrocławski

Plan wykładu

- NHibernate
  - Interfejsy
  - Trwałość automatyczna i przezroczysta
  - Definicja metadanych odwzorowujących
  - Odwzorowanie dziedziczenia
  - Mapowanie kolekcji, asocjacje
  - Cykl życia obiektów
  - Trwałość przechodnia
  - Pobieranie obiektów, strategie sprowadzania danych

Hibernate

Zaczniemy od prostego przykładu:

- `HelloWorld`

Interfejsy w Hibernate

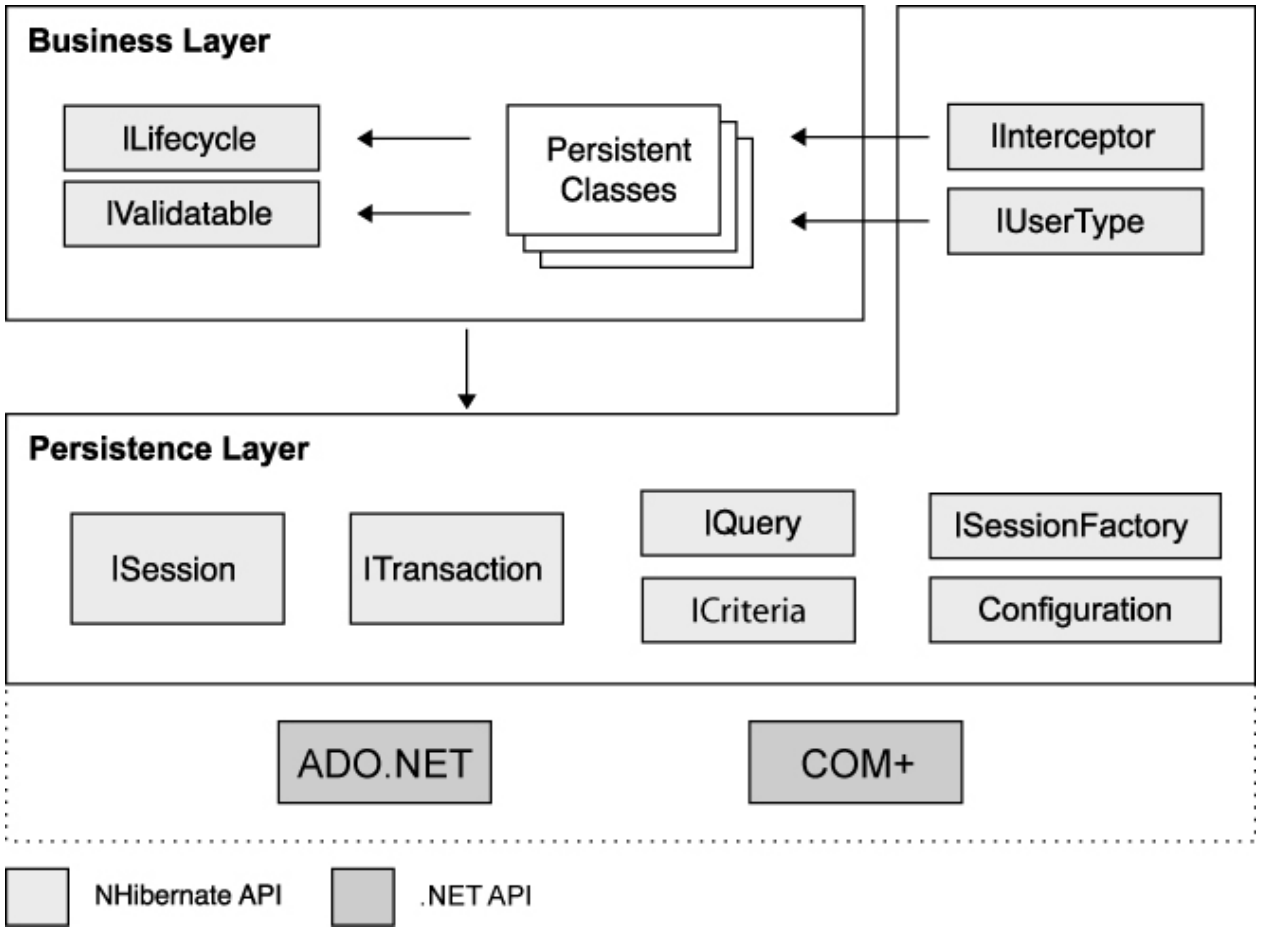
- Możemy podzielić na grupy:
  - Interfejsy do wykonywania operacji CRUD:

    `Session`, `Transaction`, `Query`
  - Interfejs służący do konfiguracji hibernate: Configuration
  - Interfejsy służace do obsługi zdarzeń hibernate (callback):

    `Interceptor`, `Lifecycle`, `Validatable`
  - Interfejsy pozwalające rozszerzyć funkcjonalność mapowania:

    `UserType`, `CompositeUserType`, `IdentifierGenerator`

Interfejsy w Hibernate



Źródło: C. Bauer, et al. NHibernate in Action, Manning

Kluczowe interfejsy

Session

- Jeden z podstawowych interfejsów
- Koszt jego utworzenia i usunięcia jest niewielki
  - szczególnie w przypadku aplikacji Webowych — obiekt będzie tworzony i niszczony przy każdym żądaniu
- Można mysleć, że jest kanał do bazy wraz z buforem obiektów
- Nazywany też czasami zarządcą trwałości (umożliwia m.in. na pobieranie i utrwalanie obiektów)

Kluczowe interfejsy

SessionFactory

- Służy do tworzenia obiektów `Session`
- Jego utworzenie jest dosyć kosztowne i zalecane jest utworzenie jednej instancji dla całej aplikacji
  - W przypadku dostępu do wielu baz, dla każdej potrzebna będzie osobna instancja
- Ten obiekt korzysta z plików mapujących oraz tworzy odpowiednie struktury i zapytania SQL

Kluczowe interfejsy

Configuration

- Służy do konfiguracji i uruchomienia hibernate
- Wczytuje plik konfiguracyjny
- Pierwszy obiekt związany w hibernate

Transaction

- Opcjonalny w użyciu
- Pozwala samodzielnie zarządzać transakcjami

Kluczowe interfejsy

Query i Criteria

- `Query` pozwala zadawać zapytania w języku HQL lub SQL
- `Criteria` jest podobny, ale pozwala zadawać zapytania zorientowane obiektowo
- Interfejsy działają zawsze w kontekście jakiejś sesji

Interfejsy wywołań zwrotnych

- Interfejsy pozwalają przechwytywać zdarzenia takie jak zapisanie, wczytanie, czy usunięcie obiektu
- Obiekty mogły implementować interfejsy `Lifecycle` i `Validatable`, ale uznano to za zły kierunek i nie jest zalecane
- Zalecane jest używanie interfejsu `Interceptor`, którego użycie nie wymaga implementacji czegokolwiek w samych obiektach biznesowych

Typy

- Hibernate ma własny zestaw typów wbudowanych, które dobrze koreluje z typami .NET
- Hibernate pozwala także na tworzenie własnych typów poprzez interfejsy `IUserType`, `ICompositeUserType` i `IParameterizedType`

Interfejsy rozszerzeń

- Hibernate daje możliwość implementacji interfejsów, dzięki którym możemy zastąpić domyślne zachowanie samodzielnie napisanym kodem.

Co możemy zaimplementować?

- Generowanie wartości klucza głównego (interfejs `IIdentifierGenerator`)
- Dialekt SQL-a (klasa abstrakcyjna `Dialect`)
- Strategie buforowania (interfejsy `ICache` i `ICacheProvider`)

Interfejsy rozszerzeń

Interfejsy rozszerzeń. Co możemy zaimplementować? c.d.

- Zarządzanie połączeniami ADO.NET

  (interfejs `IConnectionProvider`)
- Zarządzanie transakcjami

  (interfejsy `ITransactionFactory`, `ITransaction`)
- Strategie ORM (interfejs `IClassPersister`)
- Strategie dostepu do właściwości

  (interfejs `IPropertyAccessor`)
- Tworzenie pośredników (interfejs `IProxyFactory`)

Trwałość automatyczna i przezroczysta

- Automatyzm oznacza, że ktoś inny zajmuje się warstwą ADO.NET
- Przezroczystość z kolei polega na tym, że utrwalane klasy nie są świadome utrwalania
  - teoretycznie nie różnią się od wersji nieutrwalanych
- Hibernate może być przezroczysta
  - nie trzeba nic implementować ani nic dziedziczyć
  - za utrwalanie odpowiada menedżer trwałości: interfejsy `Session` i `Query`
  - pełnej przezroczystoście nie da się uzyskać
    - np. aby asocjacje były w obie strony, trzeba dodać odpowiednie pola typu `Set` lub `List`

Definicja metadanych odwzorowujących

- W NHibernate można opisać metadane:
  - W zewnętrznych plikach XML
  - Poprzez atrybuty .NET
  - Za pomocą biblioteki Fluent
- Każda z tych metod ma wady i zalety
- Bardzo przydatne jest korzystanie z narzędzi, które potrafią wygenerować m.in.
  - Pliki odwzorowań na podstawie bazy danych
  - Schemat bazy danych na podstawie opisu metadanych
  - Pliki klas na podstawie plików odzworowań
  - Pliki klas na podstawie bazy danych

  Do NHibernate są takie narzędzia

Definicja metadanych odwzorowujących

- Dobry ORM powinien mieć
  - prosty plik konfiguracyjny, możliwy do utworzenia/edycji bez używania narzędzi lub
  - czytelny sposób „atrybutowania”
- W przypadku metadanych w XML powinno się tworzyć jeden plik XML dla jednej klasy
  - Niemniej można zdefiniować więcej klas w jednym pliku XML
- Zalecane nazewnictwo: `{NazwaKlasy}.hbm.xml`
- Odwołania do plików z odwzorowaniami są zwykle w pliku głównej konfiguracji (`hibernate.cfg.xml`)

Mapowanie poprzez XML

Struktura pliku konfiguracyjnego

```
<?xml version="1.0" encoding="utf-8" ?>

<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2">
  <class name="HelloWorld.Jednostka, HelloWorld">
    ...
  </class>
</hibernate-mapping>
```

Mapowanie poprzez XML

Podstawy odzworowania właściwości

- Do definiowania właściwości jest znacznik `<property>`
- Podstawowe atrybuty:
  - name – nazwa właściwości
  - column — nazwa kolumny w tabeli w bazie danych
    - zamiast atrybutu można utworzyć zagnieżdżony element
  - type — typ pola
    - typem może być typ Hibernate lub typ .NET
    - jeśli się go nie poda, Hibernate sam go ustali
- Właściwości wyliczane
  - Definiujemy podając atrybut `formula`
  - Wartość atrybutu to wyrażenie SQL
  - Dla takiej właściwości nie jest tworzona kolumna w bazie d.

Mapowanie poprzez XML

Podstawy odzworowania właściwości

- Strategie dostępu do właściwości
  - Strategię definiujemy określając wartość artybutu `access`
  - Dostępne wartości:
    - `property` — dostęp poprzez metody dostępowe (domyślnie)
    - `field` — dostęp bezpośrednio przez pola
    - `NazwaKlasy` — dostęp poprzez klasę implementującą interfejs `NHibernate.Property.IPropertyAccessor`
- Podawanie nazw kolumn w apostrofach
  - W przypadku gdy:
    - nazwa kolumny zawiera znaki specjalne jak np. spacji,
    - w nazwie kolumny jest istotna wielkość liter

    powinniśmy podawać nazwę kolumny w apostrofach
  - Przykład:

    ```
    <property name="'RoomNumber"' column="`Nr pokoju"`/>
    ```

Mapowanie poprzez XML

Podstawy odzworowania właściwości

- Sterowanie wstawieniami i aktualizacjami
  - Można określić, czy na danej kolumnie będą wykonywane operacje wstawienia i aktualizacji wartości
    - dokładnie: czy kolumna będzie brała udział w zapytaniach INSERT i UPDATE
  - Określamy poprzez ustawienie atrybutu `update="true|false"` lub `insert="true|false"`
  - Domyślnie oba atrybuty są ustawione na `true`
  - Można także ustawić atrybut `mutable` dla znacznika `class`, definiując domyślne zachowanie dla wszystkich kolumn
    - dla UPDATE i DELETE

Mapowanie poprzez XML

Tożsamość obiektów

- W definicji metadanych, każda klasa musi mieć zdefiniowany identyfikator
- Klucz główny definiujemy za pomocą znaczników:
  - `<id>` `<composite-id>`
- Atrybuty znacznika `<id>`
  - `name` — określenie właściwości identyfikującej
  - `type` — typ właściwości identyfikującej
  - `column` — kolumna klucza głównego w tabeli
  - `access` — sposób dostępu do właściwości
  - znacznik `generator` — określa sposób generowania wartości klucza głównego

Mapowanie poprzez XML

Tożsamość obiektów

- Jeśli nie podamy atrybutu `name`, wtedy hibernate zarządza tożsamością niejawnie
  - Nie jest to zalecane rozwiązanie
    - jest kłopot np. z obsługą obiektów odłączonych
  - Mimo braku właściwości klucza, możemy pobrać identyfikator za pomocą konstrukcji:

    ```
    long ID = (long) session.GetIdentifier(osoba);
    ```
- Wybór klucza głównego
  - Wartości klucza głównego muszą być zawsze określone
  - Nigdy nie powinno być potrzeby zmiany wartości
    - np. login użytkownika jest złym kluczem (może się zmienić)
  - Kolumna klucza powinna się dobrze indeksować

  Z powyższych powodów zalecane jest utworzenie kolumny identyfikatora będącej typu int

Mapowanie poprzez XML

Tożsamość obiektów — generowanie identyfikatorów

- Generator jest określony w znaczniku `<generator>`
  - obowiązkowo podajemy klasę generatora poprzez określenie atrybutu `class`
- Jeśli generator wymaga parametrów, definiujemy je za pomocą konstrukcji `<param name="klucz">wartość</param>`
- Istotniejsze klasy generatorów:
  - `increment`
    - Generuje identyfikatory typów `long`, `short`, `int`
    - Generator odczytuje maksymalną wartość kolumny identyfikatora i na tej podstawie generuje kolejną wartość
    - Przy rozwiązaniu rozproszonym (dostęp do bazy danych ma więcej niż jeden proces) — niedopuszczalne

Mapowanie poprzez XML

Tożsamość obiektów — generowanie identyfikatorów

- Istotniejsze klasy generatorów (c.d.):
  - `identity`
    - Generuje identyfikatory typów `long`, `short`, `int`
    - Wspiera DB2, MySQL, MS SQL Server, Sybase and HypersonicSQL
  - `sequence`
    - Generuje identyfikatory typów `long`, `short`, `int`
    - Wykorzystuje sekwencje w DB2, PostgreSQL, Oracle, SAP DB, McKoi lub generatory w Interbase
  - `hilo`
    - Generuje identyfikatory typów `long`, `short`, `int`
    - Identyfikatory są unikalne w ramach całej bazy danych

Mapowanie poprzez XML

Tożsamość obiektów — generowanie identyfikatorów

- Istotniejsze klasy generatorów (c.d.):
  - `native`
    - wybiera jeden z generatorów `identity`, `sequence` i `hilo` w zależności od możliwości bazy danych
  - `uuid`
    - Generuje identyfikatory typu `string`
    - Identyfikator jest unikalny w Internecie

      (wykorzystuje adres IP i znacznik czasu)
    - Rozwiązanie kosztowne: klucze główne typu `char` zajmują sporo miejsca i są wolniejsze

Mapowanie poprzez XML

Tożsamość obiektów — identyfikatory wielokolumnowe

- Tworzymy wykorzystując element `<composite-id>`
- Pierwszy sposób deklaracji:

```
<composite-id>
  <key-property name="username" type="string"/>
  <key-property name="ou" type="string"/>
</composite-id>
```

- Utworzenie obiektu — jak dotychczas
- Pobranie obiektu:

```
var u = new User() {
  Username = "Jan",
  OU = "kadry };
session.Load(typeof(User), u)
```

Obiekt sam jest swoim identyfikatorem (w pewnym sensie)

Mapowanie poprzez XML

Tożsamość obiektów — identyfikatory wielokolumnowe

- Drugie rozwiązanie: osobno tworzymy klasę identyfikatora

```
[Serializable]
public class UserID {
    public string Username; public string OU;
    public UserID() { ... }
    public override bool Equals(Object o) { ... }
    public override int GetHashCode() { ... }
}
```

  Należy pamiętać jednak, że klasa ta musi:

  - posiadać atrybut `Serializable`
  - posiadać metody `Equals()` i `GetHashCode()`

- Deklaracja pliku metadanych będzie wtedy wyglądać

```
<composite-id name="userId" class="UserId">
  ...
</composite-id>
```

Mapowanie poprzez XML

Tożsamość obiektów — identyfikatory wielokolumnowe

- Czasami jeden ze składowych klucza głównego jest kluczem obcym
- Mamy wtedy dwa rozwiązania:
  - Zadeklarować klucz jak dotychczas, a związek podkreślić w odpowiedniej asocjaji, ustawiając atrybuty `insert="false"` i `update="false"`
  - W deklaracji klucza wielokolumnowego, zamiast `<key-property>` użyć `<key-many-to-one>`:

```
<key-many-to-one name="ou"
                 class="Organization"
                 column="OrganizationID">
```

Mapowanie poprzez XML

Szczegółowe modele obiektów

- Przedstawiona poniżej koncepcja komponentów jest rozwiązaniem problemu szczegółowości
- Korzystamy z kompozycji w obiekcie C#, natomiast w pliku odwzorującym wskazujemy, aby wszystkie dane były składowane w jednej tabeli
- Struktura pliku metadanych jest następująca:

```
<class name="User" table="User">
  <!-- Właściwości ,,proste'' klasy User -->
  <component name="Address" class="Address">
    <!-- Właściwości ,,proste'' klasy Address -->
  </component>
  <!-- Kolejne komponenty -->
</class>
```

Mapowanie za pomocą atrybutów

- Atrybuty są w przestrzeni nazw `NHibernate.Mapping.Attributes`
  - Wymaga dodatkowej biblioteki, dostępna pod adresem: http://nhforge.org/media/p/8.aspx
- Podstawowymi atrybutami są:
  - `[Class]` do określenia, że klasa jest utrwalana
    - Uwaga: trzeba podać parametr Name, jego wartością MUSI być pełna nazwa klasy (namespace+nazwa) oraz nazwa Assembly
  - `[Property]` do określenia, że właściwość ma być składowana
    - Atrybut ma szereg parametrów odpowiadających atrybutom z pliku XML
  - `[Id]` do określenia kolumny klucza
  - `[Generator]` do określenia generatora dla klucza

Mapowanie wykorzystując FluentNHibernate

- Nowy sposób tworzenia mapowania dostępny w przestrzeni nazw FluentNHibernate.Mapping
- Strona domowa: http://fluentnhibernate.org/,
  Wiki: http://wiki.fluentnhibernate.org/
- Ma możliwość definiowania własnego mapowania w języku C#, ma też tworzenie mapowania automatycznie
- Narzędzie do edycji: NHibernate Query Analyzer

Przykłady

- `FactoryCreatingCost`
- `AttributeMapping`
- `FluentMappings`
- `UsingComponents`
- `UsingComponentsAttributes`
- `CompositeIdentifiers`
- `CompositeIdentifiers2`
- `IdentityMap`

Odwzorowanie dziedziczenia

Hibernate oferuje trzy podejścia do tej kwestii:

- Tabela na klasę
- Tabela na hierarchię klas
- Tabela na podklasę

Odwzorowanie dziedziczenia

Tabela na klasę

- Wszystkie właściwości klasy są w tabeli odpowiadającej tej klasie (łącznie z właściwościami dziedziczonymi)

Zaleta: jeśli zapytanie dotyczy jednej klasy, wykona się szybko i łatwo je skontruować

Wady

- Pierwszy problem jest z modyfikacją schematu
  - modyfikacja jednego pola z Payment implikuje zmiany we wszystkich tabelach pozwiązanych
- Drugi problem jest z asocjacjami polimorficznymi

Odwzorowanie dziedziczenia

Tabela na klasę

- Problem z asocjacjami polimorficznymi (c.d.)
  - Wróćmy do naszego wcześniejszego przykładu:
    User $\overset{1..*}{—}$ Payment,
    CreditCard $\rightarrow$ Payment, BankAccount $\rightarrow$ Payment
  - Zgodnie z modelem, mamy tabele: User, CreditCard, BankAccount
  - Powiedzmy, że chcemy wszystkie płatności tzw. Kowalskiego
    - no i trzeba odpytać wszystkie tabele, które mają coś wspólnego z Payment, czyli CreditCard i BankAccount
    - wykonanie tego jest nieefektywne, nawet używając UNION
  - Problem jest tym większy, im wyższe są hierarchie dziedziczenia

Odwzorowanie dziedziczenia

Tabela na klasę

Jak realizujemy ten model w Hibernate?

- Tworzymy dla każdej klasy osobny plik, wskazując dla każdej klasy osobną tabelę
- Ten model nie wymaga żadnych dodatkowych czynności konfiguracyjnych

Odwzorowanie dziedziczenia

Tabela na hierarchię klas

- Rozwiązanie polega zastosowaniu jednej tabeli dla drzewa klas ze sobą powiązanych relacją dziedziczenia
- Dodatkowo jest kolumna dyskryminatora, który określa jakiego typu jest dany wpis w tabeli

Zaleta: Łatwo zadawać zapytania zwykłe jak i te oparte na asocjacji polimorficznej

Wady:

- Przede wszystkim problem polega na tym, że w każdym wierszu w tabeli jest sporo wartości pustych
  - a im większe (i bardziej rozgałęzione) drzewo dziedziczenia, tym tego pustego więcej

Odwzorowanie dziedziczenia

Tabela na hierarchię klas

Jak realizujemy ten model w Hibernate?

- W definicji klasy bazowej określamy nazwę i typ dyskryminatora
- Za pomocą znacznika `<subclass>` określamy podklasę
- W każdej podklasie określamy jaka jest jej wartość dyskryminatora (czyli jak będzie rozpoznawana na poziomie tabeli)

Odwzorowanie dziedziczenia

Tabela na podklasę

- W tym modelu każdy byt (klasy, w tym abstrakcyjne, interfejsy) mają swoje tabele
- W tabelach tych są tylko właściwości zdefiniowane w danej klasie lub interfejsie
- Jeżeli klasa ma podklasę, wtedy jej klucz główny jest jednocześnie obcym do nadklasy i tam znajduje się reszta danych danego obiektu

Zalety

- Pełna normalizacja schematu w bazie danych
- Asocjacje polimorficzne są elegancko reprezentowane

Odwzorowanie dziedziczenia

Tabela na podklasę

Wady

- Przede wszystkim jedna: przy większej strukturze, duża złożoność obsługi (trzeba wykonywać dużo złączeń)

Jak realizujemy ten model w Hibernate?

- Podklasy definiujemy za pomocą znacznika `<joined-subclass name="Klasa" table="TABLE">`
- W każdej podklasie definiujemy `<key column="col"/>`, która jest od razu kluczem obcym

Odwzorowanie dziedziczenia

Wybór strategii

- Jeśli mamy prostą aplikację, w której nie używamy asocjacji i zapytań polimorficznych, wybieramy wariant 1
  - czyli tabela na każdą klasę
- Gdy korzystamy z asocjacji i zapytań polimorficznych, ale podklasy mają mało właściwości, warto rozważyć wariant 2
  - czyli tabela na każdą hierarchię klas
- Jeżeli złożoną strukturę dziedziczenia, należy zastosować wariant 3
  - czyli tabela na każdą podklasę
- Teoretycznie warstwa trwałości nie powinna wpływać na decyzje projektowe, ale warto wiedzieć, który ORM będzie używany i przy projektowaniu tą wiedzę uwzględnić
  - żeby uniknąć problemów z np. wydajnością

Przykłady

- `Inheritance1TablePerClass`
- `Inheritance2TablePerClassHierarchy`
- `Inheritance3TablePerSubclass`

Mapowanie kolekcji

- Podstawowe rodzaje kolekcji:
  - `Set` – zbiór
  - `Bag` – wielozbiór
  - `Map` – struktura asocjacyjna
  - `List` – struktura indeksowana, czyli „kolejność ma znaczenie”
- Można też zaimplementować własny typ kolekcji implementując `IUserCollectionType`
- Do obsługi niektórych kolekcji NHibernate dostarcza przestrzeni nazw *Iesi.Collections* i *Iesi.Collections.Generic*

Mapowanie kolekcji

Deklaracja w pliku XML

- Do zadeklarowania zbioru mamy do dyspozycji znaczniki:
  - `<set>`, `<list>`, `<map>`, `<bag>`, `<array>` i `<primitive-array>`
- Jeśli deklarujemy kolekcję typów wbudowanych, korzystamy z konstrukcji:

```
<set name="PropertyName" cascade="none|all|...">
    <key column="id w tabeli PropertyName" not-null="true"/>
    <element column="NAZWAKOLUMNYWTABELI" type="string"/>
</set>
```

- Jeśli deklarujemy kolekcję obiektów klasy własnej, korzystamy z konstrukcji:

```
<set name="PropertyName">
    <key column="id w PropertyName"/>
    <composite-element class="PropertyClass">
        <property name="P1" />
    </composite-element>
```

Mapowanie kolekcji

Poprzez atrybuty

- ```
  [Set(Lazy=true, Table="ITEM_IMAGE")]
  [Key(1, Column="ITEM_ID")]
  [Element(2, TypeType=typeof(string),
      Column="FILENAME", NotNull=true)]
  ```
- ```
  [IdBag(Lazy=true, Table="ITEM_IMAGE")]
  [CollectionId(1, TypeType=typeof(int),
      Column="ITEM_IMAGE_ID")]
  [Generator(2, Class="sequence")]
  [Key(3, Column="ITEM_ID")]
  [Element(4, TypeType=typeof(string),
      Column="FILENAME", NotNull=true)]
  ```

Przykłady

- `Collections1`
- `Collections2`
- `Collections3`
- `Collections4`
- `Collections5`

Asocjacje

Asocjacje

- Reprezentują związki między obiektami, które mają swoje odbicie w związkach między tabelami
- Zwykle są najtrudniejszym do zaimplementowania elementem każdego systemu ORM
- W Hibernate asocjacje mają bardzo duże możliwości

Asocjacje

Asocjacje jednokierunkowe

- Przypuśćmy, że mamy klasy Oddzial $\overset{1..*}{\text{---}}$ Pracownik
- Powiązanie jest zdefiniowane tylko w jednym z obiektów, w związku z czym możliwe jest przejście tylko w jedną stronę
  - możemy napisać `Pracownik.Oddzial.Nazwa`,

    ale nie możemy napisać `Oddzial.S_Pracownicy`

Asocjacje dwukierunkowe

- Dają nam powiązanie w obie strony, czyli możemy napisać obie powyższe konstrukcje

Asocjacje

Rodzaje asocjacji:

- jeden-do-jednego
- wiele-do-jednego
- jeden-do-wielu
- wiele-do-wielu

Asocjacje

Asocjacje wiele-do-wielu

- Tworzymy korzystając ze znacznika `<many-to-many>`
- Wybrane atrybuty:
  - `column="column_name"` — nazwa kolumy w tabeli łączącej
  - `class="ClassName"` — nazwa skojarzonej klasy
  - `unique="true|false"` — wymusza ograniczenie unique na zdalnej kolumnie (prowadzi do asocjacji jeden-do-wielu)
  - `property-ref="propertyNameFromAssociatedClass"` — nazwa właściwości ze skojarzonej klasy (jeśli nie określony, nazwą będzie klucz główne ze skojarzonej klasy)

Asocjacje

Asocjacje jeden-do-wielu

- Tworzymy korzystając ze znacznika `<one-to-many>`
- Obowiązkowy atrybut `class="ClassName"`

Określenie końców asocjacji

- Przy definiowaniu asocjacji, jeden z końców musi mieć atrybut `inverse` ustawiony na `true`
  - Atrybut ustawiamy przy definiowaniu kolekcji, czyli np. `<set (...) inverse="true">...</set>`
- Warto też ustawić atrybut `cascade` na np. `all` lub `save-update`

Asocjacje

Asocjacje dwukierunkowe jeden-do-wielu i wiele-do-jednego

- Przykładowy plik mapujący

```
<class name="Person">
    <id name="id" column="ID"><generator class="native"/></id>
    <many-to-one name="address" column="AddressId"/>
</class>
<class name="Address">
    <id name="id" column="ID"><generator class="native"/></id>
    <set name="people" inverse="true">
        <key column="AddressId"/>
        <one-to-many class="Person"/>
    </set>
</class>
```

- Wygenerowane zostaną tabele:

```
Person (ID int not null primary key, AddressId bigint not null)
Address (ID bigint not null primary key)
```

Asocjacje

Asocjacje dwukierunkowe jeden-do-jednego

- Przykładowy plik mapujący

```
<class name="Person">
    <id name="id" column="ID"><generator class="native"/></id>
    <many-to-one name="address" column="addressId"
                 unique="true" not-null="true"/>
</class>
<class name="Address">
    <id name="id" column="ID"><generator class="native"/></id>
    <one-to-one name="person" property-ref="address"/>
</class>
```

- Wygenerowane zostaną tabele:

```
Person (ID int not null primary key,
        AddressId int not null unique)
Address (ID int not null primary key)
```

Asocjacje

Asocjacje jeden-do-wielu i wiele-do-jednego z tabelą łączącą

- Przykładowy plik mapujący

```
<class name="Person">
    <id name="ID" column="ID"><generator class="native"/></id>
    <set name="addresses" table="PersonAddress">
        <key column="PersonId"/>
        <many-to-many column="AddressId" unique="true" class="Address"/>
    </set>
</class>
<class name="Address">
    <id name="ID" column="addressId"><generator class="native"/></id>
    <join table="PersonAddress" inverse="true" optional="true">
        <key column="addressId"/>
        <many-to-one name="person" column="personId" not-null="true"/>
    </join>
</class>
```

- Wygenerowane zostaną tabele:

```
Person (ID int not null primary key )
PersonAddress (personId int not null, addressId int not null primary key )
Address (ID int not null primary key )
```

Asocjacje

Asocjacje jeden-do-jednego z tabelą łączącą

- Przykładowy plik mapujący

```
<class name="Person">
    <id name="id" column="ID"><generator class="native"/></id>
    <join table="PersonAddress" optional="true">
        <key column="personId" unique="true"/>
        <many-to-one name="address" column="addressId" not-null="true" unique="true"/>
    </join>
</class>
<class name="Address">
    <id name="id" column="ID"><generator class="native"/></id>
    <join table="PersonAddress" optional="true" inverse="true">
        <key column="addressId" unique="true"/>
        <many-to-one name="person" column="personId" not-null="true" unique="true"/>
    </join>
</class>
```

- Wygenerowane zostaną tabele:

```
Person (ID int not null primary key)
PersonAddress (personId bigint not null primary key, addressId bigint not null unique)
Address (ID int not null primary key)
```

Asocjacje

Asocjacje wiele-do-wielu z tabelą łączącą

- Przykładowy plik mapujący

```
<class name="Person">
    <id name="id" column="personId"><generator class="native"/></id>
    <set name="addresses" table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId" class="Address"/>
    </set>
</class>
<class name="Address">
    <id name="id" column="addressId"><generator class="native"/></id>
    <set name="people" inverse="true" table="PersonAddress">
        <key column="addressId"/>
        <many-to-many column="personId" class="Person"/>
    </set>
</class>
```

- Wygenerowane zostaną tabele:

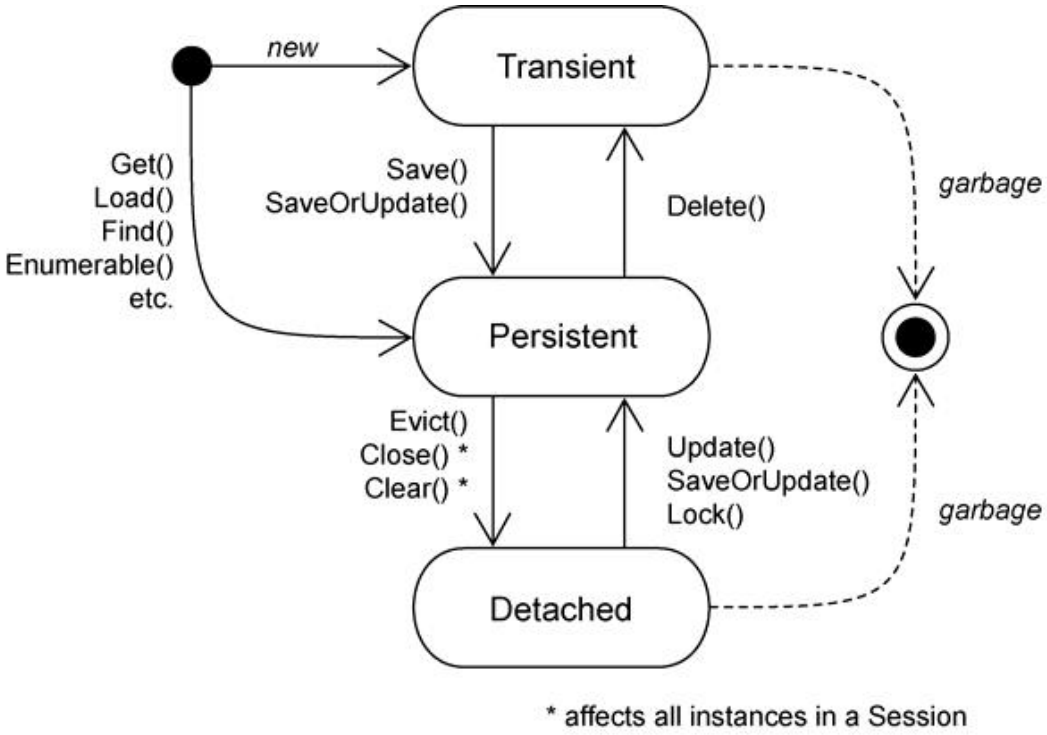
```
Person (personId bigint not null primary key)
PersonAddress (personId bigint not null, addressId bigint not null,
               primary key (personId, addressId))
Address (addressId bigint not null primary key)
```

Przykłady

- `Associations1ManyToOne`
- `Associations2ManyToMany`

Stosowanie obiektów trwałych

Cykl życia obiektu



Źródło: C. Bauer, et al. NHibernate in Action, Manning

Stosowanie obiektów trwałych

Obiekty ulotne

- Obiekt automatycznie po utworzeniu nie znajduje się w bazie danych
  - Jego stan jest ulotny
- Obiekt, do którego referencja jest w innym obiekcie ulotnym, domyślnie również jest ulotny
- Utrwalenie obiektu odbywa się poprzez wywołanie metody `Save()`
  - Analogicznie, wywołanie metody `Delete()` powoduje powrót do stanu ulotności

Stosowanie obiektów trwałych

Obiekty trwałe

- Obiekt trwały to taki o tożsamości bazodanowej
- W jaki sposób obiekt stał się utrwalony
  - Został utrwalony metodą `Save()`
  - Został utrwalony poprzez referencję z innego obiektu trwałego
  - Został wczytany poprzez pewne zapytanie
- Obiekty trwałe zawsze występują w kontekście sesji i transakcji
- Stan obiektów trwałych jest synchronizowany na końcu transakcji
- Obiekt trwały nazywamy nowym, gdy zabezpieczył już identyfikator, ale jego dane nie znalazły się jeszcze w bazie d.

Stosowanie obiektów trwałych

Obiekty trwałe

- Podczas synchronizacji aktualizowane są tylko zmodyfikowane (brudne) obiekty
  - NHibernate dokonuje tzw. automatycznego sprawdzania zabrudzenia
- Domyślnie aktualizowane są wszystkie pola obiektu
  - Jeśli aktualizować tylko zmienione, w odwzorowaniu klasy należy dodać `dynamic-update="true"`
  - Włączenie powyższego powoduje dynamiczne generowanie kwerend typu `UPDATE`

Stosowanie obiektów trwałych

Obiekty odłączone

- Obiekty związane z sesją, po jej zamknięciu stają się odłączone
  - ich dane są trwałe, ale już nie zarządzane przez Hibernate
- Obiekt odłączony można ponownie związać z nową sesją
- Obiekt można odłączyć jawnie metodą `Evict()`, ale raczej się tego nie robi

Stosowanie obiektów trwałych

Zasięg identyczności obiektów

- Mamy trzy poziomy zasięgu identyczności:
  - Bez zasięgu identyczności
    - dwukrotne pobranie tego samego wiersza spowoduje utworzenie dwóch różnych instancji obiektów
  - Transakcyjny zasięg identyczności
  - Identyczność o zasięgu procesu
- NHibernate realizuje transakcyjny zasięg identyczności
  - Działanie oglądaliśmy w przykładzie `IdentityMap`
- Istotną kwestią jest obsługa identyczności obiektów odłączonych
  - tutaj istotna jest odpowiednia implementacja metod `Equals()` i `GetHashCode()`, które powinny być spójne

Stosowanie obiektów trwałych

Zarządca trwałości

- Każdy zarządca trwałości przezroczystej powinien udostępniać funkcjonalność:
  - operacje CRUD
  - wykonywanie zapytań
  - sterowanie transakcjami
  - zarządzanie buforowaniem na poziomie transakcji
- W Hibernate zarządca trwałości jest realizowany przez interfejsy `ISession`, `IQuery`, `ICriteria` i `ITransaction`
- Warstwą pomiędzy aplikacją a NHibernate jest obiekt `ISession`

Przykład

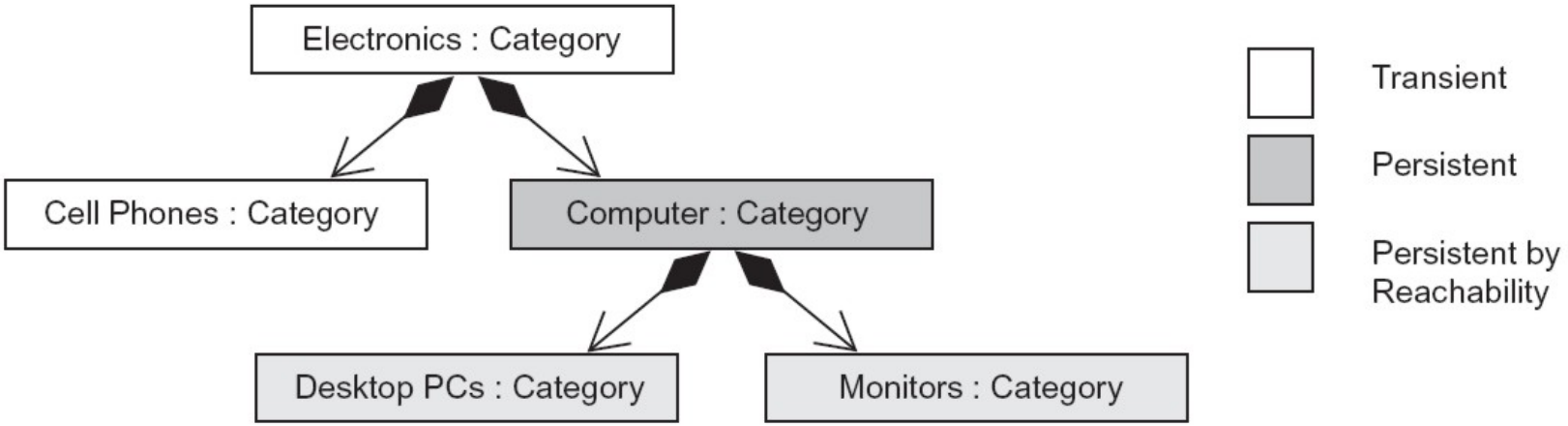
- `UsingObjects`

Trwałość przechodnia w Hibernate

Trwałość przez osiągalność

- Ma miejsce wtedy, gdy z obiektu trwałego jest referencja do innego obiektu
  - wtedy ten inny obiekt realizuje trwałość przez osiągalność
- Jest rekurencyjna
- Zapewnia integralność więzów referencyjnych
  - graf obiektów można odtworzyć wczytując jego korzeń
- Teoretycznie istnieje obiekt korzenia, z którego da przejść do dowolnego innego obiektu trwałego
  - w szczególności nieosiągalne obiekty powinny być z bazy usunięte (niewydajne)
- NHibernate nie implementuje tego modelu

Architektura warstwowa



Źródło: C. Bauer, et al. NHibernate in Action, Manning

Trwałość przechodnia w Hibernate

Trwałość kaskadowa

- Model realizowany w NHibernate
- Koncepcja podobna do trwałości przez osiągalność
- Powiązania są odtwarzane na podstawie asocjacji
  - domyślnie, hibernate nie dokonuje analizy asocjacji
- Kaskady zwykle używa się do relacji jeden-do-jednego i jeden-do-wielu
  - używanie kaskady w przypadkach wiele-do-jednego i wiele-do-wielu jest raczej bez sensu
- Wartości kaskady można łączyć, np.

  `cascade="save-update, delete"`

Trwałość przechodnia w Hibernate

Trwałość kaskadowa

- Do sterowania przechodniością trwałości mamy atrybut `cascade`
  - `cascade="none"` — ignorowanie asocjacji
  - `cascade="save-update"` — asocjacja zostanie wykorzystana, gdy obiekt został przekazany metodzie `Save()` lub `Update()`
    - skutkiem będzie utrwalenie obiektów „potomnych”
  - `cascade="delete"` — asocjacja zostanie wykorzystana, gdy obiekt został przekazany metodzie `Delete()`
    - skutkiem będzie „ulotnienie” obiektów „potomnych”
  - `cascade="all"` — jak w przypadkach `save-update` i `delete` oraz obsługa metod `Evict()` i `Lock()`
  - `cascade="delete-orphan"` — stosuje się do relacji jeden-do-wielu i określa wykonanie metody `Delete()` na wszystkich obiektach „potomnych”
  - `cascade="all-delete-orphan"` — jak w przypadku `all` i `delete-orphan`

Przykład

- `PersistenceByReachability1`
- `PersistenceByReachability2`

Pobieranie obiektów

Hibernate udostępnia następujące metody pobierania obiektów:

- Nawigacja po grafie obiektów,

  np. `user.Adres.Miasto`
- Pobranie obiektu na podstawie jego identyfikatora
- Zastosowanie języka HQL
- Wykorzystanie interfejsu `Criteria`, który umożliwia zadawanie zapytań w sposób „obiektowy”
- Przekazanie zapytań SQL

Pobieranie obiektów

Na podstawie identyfikatora

- Najszybszy sposób pobrania obiektu
- Typowa konstrukcja
  - `var user = session.Get<User>( userID )`
- W przypadku nieznalezienia obiektu, metoda `Get()` zwróci wartość `null`
  - dostępna także w tym celu metoda `Load()` w przypadku nieznalezienia obiektu rzuci odpowiednim wyjątkiem

Pobieranie obiektów

Za pomocą języka HQL

- SQL służy tylko do pobierania danych
- Przykładowy kod:

```
IQuery q = session.CreateQuery("from User u where u.firstname = :fname");
q.SetString("fname", "Arnold");
IList<User> result = q.List<User>();
```

- Niektóre z możliwości tego języka:
  - Określanie ograniczeń dla właściwości obiektów powiązanych
    - nawigacja po grafie obiektów za pomocą języka zapytań
  - Pobranie podzbioru właściwości zamiast wszystkich
    - może zwiększyć wydajność jeśli jedno z pól to BLOB
  - Sortowanie wyników
  - Dzielenie wyników na strony
  - Agregacja z użyciem `group by` i `having`, stosowanie funkcji agregujących `sum`, `max`, `min`
  - Podzapytania (zapytania zagnieżdżone)

Pobieranie obiektów

Zapytanie przez określenie kryteriów

- Zapytanie budujemy poprzez wywołania metod odpowiednich obiektów (zwane też QBC — *Query By Criteria*)
- Sprawdzanie zapytania jest na etapie kompilacji, a nie uruchomienia, jak w przypadku HQL
- Przykładowy kod:

```
ICriteria criteria = session.CreateCriteria( typeof( User ) );
criteria.Add( Expression.Like("firstname", "Alfred") );
IList<User> result = criteria.List<User>();
```

Zapytanie przez przykład

- Polega na utworzeniu obiektu i wypełnieniu tylko wybranych pól (zwane też QBE — *Query By Example*)
  - Zwracane są obiekty, których pola są równe tym ustawionym, a pozostałe mają dowolne wartości
- Przykładowy kod:

```
User exampleUser = new User { FirstName = "Alfred" };
ICriteria criteria = session.CreateCriteria( typeof( User ) );
criteria.Add( Example.Create(exampleUser) ); IList<User> result = criteria.List<User>();
```

Dokumentacja

- HQL

  https://www.hibernate.org/hib_docs/nhibernate/1.2/reference/en/html/queryhql.html

- Criteria

  https://www.hibernate.org/hib_docs/nhibernate/1.2/reference/en/html/querycriteria.html

- SQL

  https://www.hibernate.org/hib_docs/nhibernate/1.2/reference/en/html/querysql.html

NHibernate Query Analyzer

- Narzędzie do zadawania zapytań HQL
  - Strona główna

    http://www.assembla.com/wiki/show/NHibernateQueryAnalyzer
  - Poprzednia strona główna:

    http://ayende.com/projects/nhibernate-query-analyzer.aspx
- Pozwala także tworzyć pliki konfiguracji i mapowania w sposób „wizualny”
- Niestety dosyć ciężko zmusić to do działania

NHibernate Query Analyzer

- Jak mi się udało z tego narzędzia skorzystać? Mamy przykład QueryingDatastore
  - Uruchamiamy NQA
  - Tworzymy nowy projekt: File → New → Project
  - Dodajemy plik głównej konfiguracji: hibernate.cfg.xml
  - Do katalogu z programem NQA kopiujemy
    - pliki mapowania: Koszyk.hbm.xml, Produkt.hbm.xml
    - pliki klas: Koszyk.cs, Produkt.cs
    - skompilowany QueryingDatastore.exe oraz plik NHibernate.ByteCode.LinFu.dll
  - Klikamy w *Build Project*
  - Wybieramy File → New → Query i możemy zadawać zapytania

Przykład

- *NHibernate Query Analyzer* online demo

Przykład

- `QueryingDatastore`

Strategie sprowadzania danych

Hibernate imlementuje następujące strategie sprowadzania danych:

- Sprowadzanie natychmiastowe, ang. *immediate fetching*
- Sprowadzanie leniwe, ang. *lazy fetching*
- Sprowadzanie wyprzedające, ang. *eager (outer join) fetching*
- Sprowadzanie wsadowe, ang. *batch fetching*

Ciekawy wpis na blogu:
http://ayende.com/Blog/archive/2009/04/13/nhibernate-mapping-ltsetgt.aspx

Strategie sprowadzania danych

Sprowadzanie natychmiastowe

- Po wysłaniu zapytania pobierającego obiekt, od razu wysyłane są dalsze zapytania do pobrania obiektów zależnych (powiązanych asocjacjami)
- Mało efektywne rozwiązanie
- Ma sens wtedy, gdy wiemy, że pobierane obiekty są już w cache'u

Strategie sprowadzania danych

Sprowadzanie leniwe

- Pobranie obiektów jest maksymalnie odwlekane
- Jest podstawową własnością systemów ORM (domyślna strategia NHibernate)
  - zagrożenie: strategia natychmiastowa może prowadzić do pobrania do pamięci całej bazy
- Spostrzeżenia
  - Trzeba uważać na liczbę generowanych pod-zapytań (po każdy pod-obiekt osobne)
  - Warto stosować, gdy „dociąganych” jest niewiele obiektów
    - np. w reakcji na kliknięcie użytkownika w aplikacji

Strategie sprowadzania danych

Sprowadzanie wyprzedzające

- Polega na pobraniu obiektów powiązanych z pobieranym obiektem, stosując złączenia
- Jest wydajniejsze od sprowadzania natychmiastowego i czasami bywa bardziej optymalne od sprowadzania leniwego
- Do pobrania obiektów wykorzystywany jest OUTER JOIN
- Można włączyć na poziomie transakcji, chociaż częściej określa się w pliku mapującym
- Poprzez parametr *max_fetch_depth* w pliku *hibernate.cfg.xml* możemy określić, ile tabel może być maksymalnie złączanych (domyślnie: 1)

Strategie sprowadzania danych

Sprowadzanie wsadowe

- Nie jest to osobna stategia sprowadzania, tylko mechanizm przyspieszający działanie sprowadzania leniwego i natychmiastowego.
- Zamiast podawać w klauzuli `WHERE` pojedynczy identyfikator, NHibernate zbierze ich więcej i poda razu cały zestaw identyfikatorów

Mapowanie XML

- Do określenia strategii sprowadzania danych mamy atrybuty: *lazy*, *fetch*, *batch-size*

Przykład

- `FetchingStrategies`

NHibernate to LINQ

- Obecnie jest dostępna tylko jedna implementacja
- Obecnie dostępnych jest tylko część funkcjonalności
- Trzeba pobrać osobną DLL, do pobrania ze strony:

  http://sourceforge.net/projects/nhibernate/files/
- Takie trochę wprowadzenie:

  http://blogs.hibernatingrhinos.com/nhibernate/archive/2008/11/26/linq-to-nhibernate.aspx

Przykład

- `NHibernateLinq`