Kurs programowania aplikacji bazodanowych

Wykład 4

Paweł Rajba

Instytut Informatyki

Uniwersytet Wrocławski

Plan wykładu

- Wprowadzenie do trwałości
  - Niedopasowanie paradygmatów, architektura warstwowa
- Systemy ORM
- Hibernate
  - Interfejsy
  - Trwałość automatyczna i przezroczysta
  - Definicja metadanych odwzorowujących
  - Odwzorowanie dziedziczenia
  - Mapowanie kolekcji, asocjacje
  - Cykl życia obiektów
  - Trwałość przechodnia
  - Pobieranie obiektów, strategie sprowadzania danych

Czym jest trwałość?

- Składowanie danych utworzonych podczas działania programu
  - Najczęściej obecnie w bazie relacyjnej
- Relacyjne bazy danych są najbardziej rozpowszechnione
  - Były próby stworzenia obiektowych baz danych, ale na razie niewiele z tego wyszło
- „Prosty schemat” uzyskania trwałości w Javie
  - generujemy przygotowane zapytania przy użyciu kodu Javy
  - interfejs JDBC pozwala nam dołączyć parametry do zapytań, wykonać je przejrzeć wyniki

Czym jest trwałość?

- „Prosty” schemat realizuje dostęp niskopoziomowy
  - Zadanie dosyć żmudne
  - Programista powinien się skupić na tworzeniu logiki biznesowej, a nie obsługą trwałości
  - To może zrezygnować z baz relacyjnych?

    — Niestety, jedyne sprawdzone rozwiązanie
- Obiekty trwałe i ulotne
  - Trwałość przechodnia, czyli trwałość przez osiągalność
- Trwałość będziemy rozumieć jako połączenie elementów
  - zapamiętywanie, organizacja i pobieranie danych,
  - współbieżność i integralność danych,
  - współdzielenie danych.

Czym jest trwałość?

- Pojęcie obiektowego modelu dziedziny
  - abstrakcja identyfikująca rzeczywiste byty występujące w danej dziedzinie i powiązania między nimi
  - nie jest przeznaczony dla programistów i nie zawiera szczegółów dotyczących implementacji
- Systemy ORM operują na modelu dziedzinowym
- Uwaga: proste i małe aplikacje czasami łatwiej napisać bez korzystania z modelu dziedzinowego i ORM, tylko po prostu operując na tabelkach

Niedopasowanie paradygmatów

W czym rzecz

- Polega na zupełnie innych filozofii dotyczących modelu relacyjnego i obiektowego
- Niedopasowanie implikuje szereg konkretnych problemów

Problem szczegółowości

- Przyjrzyjmy się mu na przykładzie. Mamy klasy User i Address
- W bazie danych możemy utworzyć dla nich osobne tabele
  - pojawia się problem dużych złączeń
- Albo zapamiętać poszczególne pola adresu w tabeli User: User(ID, Name, A_Street, A_City, A_Code)
  - i wtedy pojawia się problem szczegółowości.
- Problem łatwy do rozwiązania, chociaż często spotykany.

Niedopasowanie paradygmatów

Problem podtypów

- W większości (czyli we wszystkich) DBMS nie jest obsługiwane dziedziczenie
- Z drugiej strony dziedziczenie to podstawowy mechanizm w językach obiektowych
- Zagadnienie asocjacji polimorficznej. Rozpatrzmy przykład:
  - Mamy klasy User $\overset{1..*}{\text{—}}$ Payment, CreditCard $\rightarrow$ Payment, BankAccount $\rightarrow$ Payment
  - Powiązanie User–Payment realizuje asocjację polimorficzną

Niedopasowanie paradygmatów

Problem identyczności

- Jak możemy porównywać elementy:
  - Za pomocą porównania obiektów operatorem ==
  - Za pomocą zdefiniowania metody `equals()`
  - Porównując klucz główny w tabeli relacyjnej

  Oczywiście, wszystkie te sposoby się istotnie różnią
- Pojawia się problem występowania wielu obiektów reprezentujących ten sam wiersz z tabeli relacyjnej
- Pierwsze zalecenie: jako klucz główny powinno być pole niezależne od innych, będące int-em

Niedopasowanie paradygmatów

Problemy dotyczące asocjacji

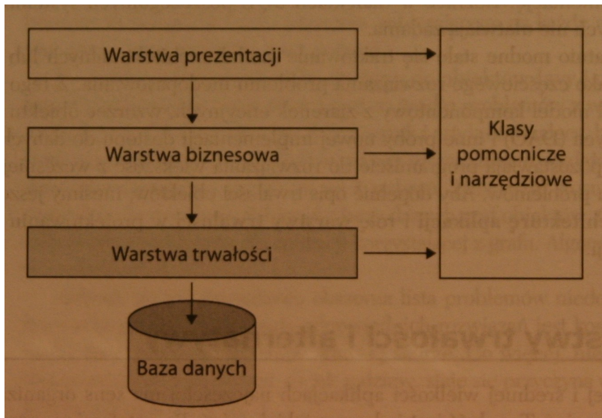
- W systemie relacyjnym mamy związki jeden-do-wielu i jeden-do-jednego
  - Związek wiele-do-wielu jest tak naprawdę połączeniem związków jeden-do-wielu
- W świecie obiektowym poprzez kompozycję możemy z kolei
  - tworzyć asocjacje jednokierunkową
  - tworzyć asocjacje dwukierunkową, definiując odpowiednie elementy w obu obiektach
  - tworzyć asocjacje jeden-do-jednego, jeden-do-wielu, wiele-do-wielu
- Ponieważ w obiektowości można więcej, w systemie relacyjnym trzeba dokonywać symulacji

Niedopasowanie paradygmatów

Problemy nawigacji po grafie obiektów

- Poprzez utworzenie odpowiednich asocjacji, łatwiej nawigować po grafie obiektów
- Mamy daną ścieżkę: jednostka.getOsoba(3).getUlica()
  Jak to pobrać?
  - Efektywnie byłoby wykonać odpowiedni JOIN
  - Metody jednak będą ściągać dane po trochę, czyli bardzo nieefektywnie
- Ogólnie pojawia sie problem odwzorowań języka wewnętrznego systemu ORM na odpowiedni dialekt SQL

Architektura warstwowa



Źródło: C. Bauer, G. King, Hibernate in Action, Manning 2005

Co w warstwie trwałości?

- Serializacja
  - Jest to podstawowy, wbudowany w język mechanizm trwałości
  - Klasa jest zdolna do trwałości, gdy implementuje interfejs Serializable
  - W procesie serializacji są wykorzystywane głównie dwie metody: readObject i writeObject
    - Nie trzeba ich implementować, chociaż można je nadpisać
  - Zapisywane są automatycznie całe grafy obiektów
  - Przydaje się głównie w
    - prostych zastosowaniach np.rejestracji profilu użytkownika,
    - RMI: parametry i obiekty zwracane muszą dać się serializować

Co w warstwie trwałości?

- Dlaczego nie serializacja? Mechanizm ten nie umożliwia
  - Formułowania zapytań
    - Nie możemy wczytać obiektu warunkowo
  - Częściowego odczytu bądź aktualizacji
    - Co jest przydatne przy operowaniu na dużych danych
  - Zarządzania cyklem życia obiektów
    - Nie ma tutaj pojęcia stanu
  - Współbieżności i transakcji
    - Nie można równocześnie czytać i zapisywać z tego samego strumienia (np. poprzez wątki)
    - Pojęcia transakcji w ogóle w przypadku serializacji nie ma

Architektura warstwowa

Co w warstwie trwałości?

- Ręczne tworzenie kodu za pomocą SQL i JDBC
  - Zaleta: mamy dokładnie to co chcemy
  - Wada: kupa roboty i związanych z tym kosztów
    - a po co, skoro są gotowe rozwiązania
- Ziarna encyjne *(entity beans)*
  - Rozwiązania tego typu nie sprawdziły się
  - Popularność tego typu rozwiązań istotnie spada
- Odwzorowanie obiektowo–relacyjne
  - Najlepsze rozwiązanie chociaż nie idealne

Odwzorowanie obiektowo–relacyjne

Czym jest ORM?

- Oprogramowanie zapewniający trwałość automatycznie
- Automat translacji działa na podstawie metadanych opisujących odwzorowanie obiektu na dane
- Translacja jest przezroczysta i działa w obie strony

Cztery główne składowe ORM

- interfejs pozwalający na wykonywanie operacji CRUD na obiektach klas umiejących zapewnić trwałość
- interfejs lub język pozwalający zadawać zapytania
- narzędzia do określania metadanych
- elementy dodatkowe: obsługa transakcji, leniwe pobieranie asocjacji, optymalizacje

Problemy i pytania związane z ORM

- Jak musi wyglądać klasa, żeby można było ją utrwalać?
- Jak definiuje się metadane? Czy są narzędzia, które robią to automatycznie? Czy trzeba je w ogóle definiować?
- W jaki sposób jest odwzorowywana hierarchia dziedziczenia?
- Jak realizowane sa zagadnienia:
  - atrybut „not null”,
  - dostępność pól: public, private, protected,
  - nazewnictwo
    - np. w Oracle nazwy mogą mieć co najwyżej 30 znaków

Problemy i pytania związane z ORM

- W jaki sposób realizowana jest tożsamość obiektów?
- Jak tworzyć obiekt logiki biznesowej (user, payment, itd.)? Czy można to automatyzować?
- Jak wygląda współpraca pomiędzy obiekatmi logiki biznesowej a obiektami oprogramowania ORM?
- Jakie są możliwości języka zapytań?
- Jak wydajne jest pobieranie danych z asocjacji?
  - nasz przykład: jednostka.getOsoba(3).getUlica()

Dlaczego warto użyć ORM?

Rozwiązanie oparte na ORM zapewnia:

- Produktywność
  - programista skupia się na problemie biznesowym, a nie składowaniem obiektów
- Konserwację
  - Mniej kodu przez co łatwiej panować nad aplikacją
  - ORM jest zwykle bardziej elastyczny niż własna warstwa dostępu do danych, przez co łatwiej modyfikować aplikację

Dlaczego warto użyć ORM?

Rozwiązanie oparte na ORM zapewnia c.d.:

- Wydajność
  - Powszechnie panuje przekonanie, że ręcznie napisana trwałość będzie wydajniejsza od tej zautomatyzowanej w ORM
  - Jednak często nie jest to takie proste, bo ORM jest dobrze zoptymalizowany i dopracowany
  - Często też istotną kwestią określającą wydajność ORM jest odpowiednia konfiguracja
- Niezależność od dostawcy
  - Jedna z istotniejszych zalet: zwykle raz napisana aplikacja będzie działać na Oracle, SQL Server, PostgreSQL, itd.

Istotne pytanie: kupić czy napisać?

- Napisanie rozbudowanego systemu ORM nie jest trywialne i jest czasochłonne
- Odpowiedź na pytanie zależy przede wszystkim od
  - Czasu, który jest na „zdobycie" systemu
  - Wymagań, które system powinien spełniać
  - Kosztów, które można ponieść
- Odpowiedź ułatwiają ORM-y udostępniane na licencji GPL
- Zwykle odpowiedź brzmi: kupić lub wykorzystać jeden z darmowych

Jakie ORM będziemy omawiać?

Rozwiązania dla środowiska Java

- Hibernate
- JDO (JPOX)

Rozwiązania dla środowiska .NET

- NHibernate
- eXpress Persistent Objects (XPO) firmy Devexpress

Zaczniemy od prostego przykładu:

- `HelloWorld`

Interfejsy w Hibernate

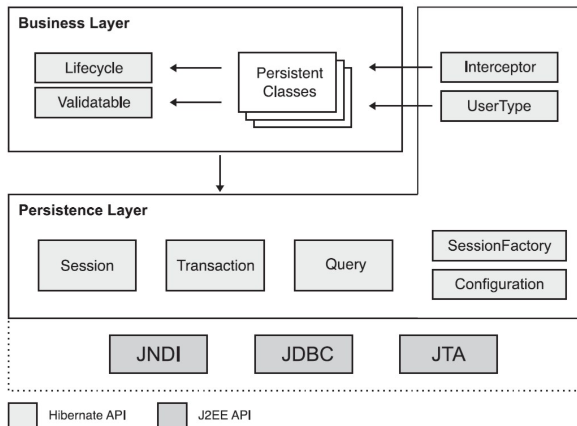
- Możemy podzielić na grupy:
  - Interfejsy do wykonywania operacji CRUD:

    `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`
- Hibernate korzysta także z:
  - JDBC
  - Java Transaction API (JTA)
  - Java naming and Directory Services (JNDI)

Interfejsy w Hibernate



Źródło: C. Bauer, G. King, Hibernate in Action, Manning 2005

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 o 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

Kluczowe interfejsy

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

Kluczowe interfejsy

Typy

- Hibernate ma własny zestaw typów wbudowanych, które dobrze koreluje z typami w języku Java
- Hibernate pozwala także na tworzenie własnych typów poprzez interfejsy `UserType` i `CompositeUserType`

Kluczowe interfejsy

Interfejsy rozszerzeń

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

Co możemy zaimplementować?

- Generowanie wartości klucza głównego (interfejs `IdentifierGenerator`)
- Dialekt SQL-a (klasa abstrakcyjna `Dialect`)
- Strategie buforowania (interfejsy cache i `CacheProvider`)

Kluczowe interfejsy

Interfejsy rozszerzeń. Co możemy zaimplementować? c.d.

- Zarządzanie połączeniami JDBC (interfejs `ConnectionProvider`)
- Zarządzanie transakcjami (interfejsy `TransactionFactory`, `Transaction`, `TrasanctionManagerLookup`)
- Strategie ORM (interfejs `ClassPersister`)
- Strategie dostepu do właściwości (interfejs `PropertyAccessor`)
- Tworzenie pośredników (interfejs `ProxyFactory`)

Trwałość automatyczna i przezroczysta

- Automatyzm oznacza, że ktoś inny zajmuje się warstwą JDBC
- 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`

Podstawowe zasady przy tworzeniu trwałych klas

- Nie trzeba implementować inferfejsu `Serializable`
  - Jednak jest konieczna przy zapisywaniu obiektów w `HttpSession` i przy kontaktach z RMI
- Musi być utworzony kontruktor bezparametrowy
  - nie musi być publiczny
- Należy utworzyć settery i gettery o nazwach zgodnych ze standardową konwencją
  - metody te mogą być nietrywialne
- Można dodawać metody elementów logiki biznesowej

Definicja metadanych odwzorowujących

- Metadane zapisywane są w pliku XML
- Dobry ORM powinien mieć prosty plik konfiguracyjny, możliwy do utworzenia/edycji bez używania narzędzi
- Bardzo przydatne jest korzystanie z narzędzi, które potrafią wygenerować
  - Pliki odwzorowań na podstawie bazy danych
  - Schemat bazy danych na podstawie plików odwzorowań
  - Pliki klas na podstawie plików odzworowań

  Hibernate dostarcza takie narzędzia
- Powinno się tworzyć jeden plik XML dla jednej klasy
  - Można zdefiniować więcej klas w jednym pliku XML
- Zalecane nazewnictwo: {`NazwaKlasy`}`.hbm.xml`
- Pliki z odwzorowaniami są zwykle w pliku `hibernate.cfg.xml`

Struktura pliku konfiguracyjnego

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="helloworld">
  <class name="helloworld.Message" table="MESSAGES">
  ...
  </class>
</hibernate-mapping>
```

Definicja metadanych odwzorowujących

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 Javy
    - 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.

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 `org.hibernate.property.PropertyAccessor`
- 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"‘/>
```

Definicja metadanych odwzorowujących

Podstawy odzworowania właściwości

- Sterowanie wstawieniami i aktualizacjami
  - Można 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

Definicja metadanych odwzorowujących

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

Definicja metadanych odwzorowujących

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 sztucznej kolumny będącej typu int

Definicja metadanych odwzorowujących

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

Definicja metadanych odwzorowujących

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

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

Definicja metadanych odwzorowujących

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:

```
User user = new User();
user.setUsername("jan");
user.setOU("kadry");
session.load(User.class, user)
```

Obiekt sam jest swoim identyfikatorem (w pewnym sensie)

Definicja metadanych odwzorowujących

Tożsamość obiektów — identyfikatory wielokolumnowe

- Drugie rozwiązanie: osobno tworzymy klasę identyfikatora

  ```
  class UserID implements Serializable {
      public String Username; public String OU;
      public UserID() { ... }
      public boolean equals(Object o) { ... }
      public int hashCode() { ... }
  }
  ```

  Należy pamiętać jednak, że klasa ta musi:

  - implementować interfejs `Serializable`
  - posiadać metody `equals()` i `hashCode()`

- Deklaracja pliku metadanych będzie wtedy wyglądać

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

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">
```

Szczegółowe modele obiektów

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

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

Przykłady

- `SimpleMapping`
- `UsingComponents`
- `CompositeIdentifiers`
- `CompositeIdentifiers2`
- `IdentityMap`

Odwzorowanie dziedziczenia

Hibernate oferuje trzy podejścia do tej kwestii:

- Tabela na każdą klasę konkretną
- Tabela na każdą hierarchię klas
- Tabela na każdą podklasę

Odwzorowanie dziedziczenia

Tabela na klasę konkretną

- 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 konkretną

- Problem z asocjacjami polimorficznymi (c.d.)
  - Wróćmy do naszego wcześniejszego przykładu:
    User $\overset{1..*}{\text{---}}$ 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 konkretną

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 każdą 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:

- Pierwsza polega na tym, że we wszystkich klas niebazowych nie można odwzorować właśnośći `NOT NULL`
  - Wszędzie muszą być dopuszczone wartości `NULL`

Odwzorowanie dziedziczenia

Tabela na każdą hierarchię klas

- Drugi 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

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 jaki jest jej dyskryminator (czyli jak będzie rozpoznawana na poziomie tabeli)

Odwzorowanie dziedziczenia

Tabela na każdą 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 każdą 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

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

- `Inheritance1`
- `Inheritance2`
- `Inheritance3`

Mapowanie kolekcji

- Utrwalać można kolekcje typów:
  - `java.util.Set`,
  - `java.util.Collection`,
  - `java.util.List`,
  - `java.util.Map`,
  - `java.util.SortedSet`,
  - `java.util.SortedMap`
    - można też zaimplementować własny typ implementując `org.hibernate.usertype.UserCollectionType`
- Utrwalenie polega na wywołaniu metody `session.persist(collection)`

Mapowanie kolekcji

- Należy uważać z rzutowaniem pobranych obiektów, ponieważ Hibernate przechowuje kolekcje za pomocą własnych reprezentacji:

```
Cat cat = new DomesticCat();
Cat kitten = new DomesticCat();
....
Set kittens = new HashSet();
kittens.add(kitten);
cat.setKittens(kittens);
session.persist(cat);
kittens = cat.getKittens(); // Ok
(HashSet) cat.getKittens(); // Error
```

Mapowanie kolekcji

Deklaracja w pliku mapującym

- 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" cascade="none|all|...">
    <key column="id w tabeli PropertyName" not-null="true"/>
    <one-to-many class="PropertyClass"/>
</set>
```

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.getOddzial().getNazwa()`, ale nie możemy napisać `Oddzial.getPracownicy()`

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 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 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 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 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 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 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:

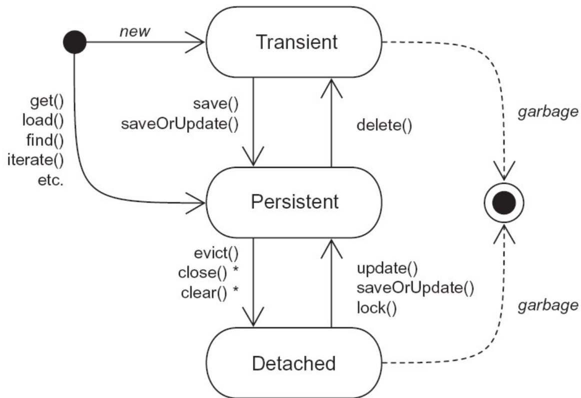
```
Person (personId bigint not null primary key)
PersonAddress (personId bigint not null, addressId bigint not null,
               primary key (personId, addressId))
Address (addressId bigint not null primary key)
```

Przykłady

- `Associations1`
- `Associations2`

Stosowanie obiektów trwałych

Cykl życia obiektu



Źródło: C. Bauer, G. King, Hibernate in Action, Manning 2005

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

Obiekty trwałe

- Obiekt trwały to taki, który obiekt 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
  - hibernate 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`

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
    - dla całej JVM istnieje jeden obiekt reprezentujący dany wiersz
- Hibernate realizuje transakcyjny zasięg identyczności
  - Działanie oglądaliśmy w przykładzie `IdentityMap`
- Istotną kwestią jest obsługa identyczności obiektów odłączonych
  - tutaj istotna jest odpowiednia implementacja metod `equals()` i `hashCode()`, 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 `Session`, `Query`, `Criteria` i `Transaction`
- Warstwą pomiędzy aplikacją a hibernate jest obiekt `Session`

Przykład

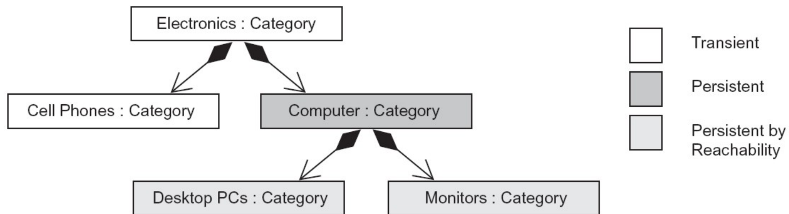
- `UsingObjects`

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)
- Hibernate nie implementuje tego modelu

Architektura warstwowa



Źródło: C. Bauer, G. King, Hibernate in Action, Manning 2005

Trwałość przechodnia w Hibernate

Trwałość kaskadowa

- Model realizowany w Hibernate
- 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

- `PersistenceByReachability`
- `PersistenceByReachability2`

Rozróżnienie obiektów ulotnych i odłączonych

Obiekt jest traktowany jako niezapisany obiekt ulotny, jeśli

- właściwość identyfikatora wynosi `null`
- właściwość wersji wynosi `null`
- ustawiono w pliku mapującym klasy właściwość `unsaved-value` i wartość właściwości identyfikującej jest jej równa
- ustawiono w pliku mapującym klasy właściwość `unsaved-value` i wartość właściwości wersji jest jej równa
- przekazano obiekt implementujący interfejs `Interceptor`, który zwrócił `Boolena.TRUE` po wywołaniu metody `Interceptor.isUnsaved()` z argumentem będącym testowanym obiektem

Pobieranie obiektów

Hibernate udostępnia następujące metody pobierania obiektów:

- Nawigacja po grafie obiektów,

  np. `user.getAddress().getCity()`
- 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
  - `User user = (User) session.get(User.class, 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

- Służy tylko do pobierania danych
- Przykładowy kod:

```
Query q = session.createQuery("from User u where u.firstname = :fname");
q.setString("fname", "Arnold");
List result = q.list();
```

- 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:

```
Criteria criteria = session.createCriteria(User.class);
criteria.add( Expression.like("firstname", "Alfred") );
List result = criteria.list();
```

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(); exampleUser.setFirstname("Alfred");
Criteria criteria = session.createCriteria(User.class);
criteria.add( Example.create(exampleUser) ); List result = criteria.list();
```

Strategie sprowadzania danych

Hibernate imlementuje następujące strategie sprowadzania danych:

- Sprowadzanie natychmiastowe
  - Pobierany jest automatycznie cały graf powiązanych ze sobą obiektów
  - Raczej mało wydajne rozwiązanie
    - chyba, że wiemy, iż będziemy korzystać ze wszystkich pobranych obiektów
- Sprowadzanie leniwe
  - Pobranie obiektów jest maksymalnie odwlekane
  - Wadą jest generowanie wielu zapytań (po każdy obiekt osobne)

Strategie sprowadzania danych

Strategie sprowadzania danych c.d.:

- Sprowadzanie wyprzedzające
  - Polega na pobraniu obietków powiązanych z pobieranym obiektem, ale stosując złączenia
  - Jest wydajniejsze od sprowadzania natychmiastowego
- Sprowadzanie wsadowe
  - Zamiast podawać w klauzuli `WHERE` pojedynczy identyfikator, hibernate zbierze ich więcej i poda razu cały zestaw identyfikatorów
  - Sprowadzanie wyprzedzające będzie w większości wypadków szybsze

Przykład

- `QueryingDatastore`