

Object Relational Mapping

- Koźmik Ryszard
- Dawidowicz Bolesław

Plan prezentacji

- Świat relacyjny i obiektowy
 - Różnice i problemy
- Środowiska ORM
 - Jak, po co i dlaczego
- Konkretnie rozwiązania

Skąd wynika problem

- Dwie powszechnie przyjęte technologie przy tworzeniu systemów informatycznych są to
 - Relacyjne bazy danych
 - Obiektowe języki programowania
- Model relacyjny i obiektowy mają wiele różnic które uwidaczniają się zarówno na etapie projektowym jak i na etapie integracji systemu

Dwa światy - różnice

- Model relacyjny
 - Oparty ściśle na matematycznych podstawach
 - Dostęp poprzez język zapytań wraz z jego specyfiką:
 - Transakcyjność
 - Współbieżność
 - Spójność danych
 - Bezpieczeństwo

Dwa światy - różnice

- Model obiektowy
 - Nie ma podstaw w postaci teorii matematycznej
 - Odwzorowuje specyficzne relacje i konstrukcje takie jak:
 - Dziedziczenie
 - Polimorfizm
 - Agregacja
 - Klasy abstrakcyjne

Problem ziarnistości

- Załóżmy że mamy klasę 'User' a w niej pole typu 'Address'
- Możemy powiązać je na dwa sposoby:
 - Stworzyć w SQL nowy typ dla adresu i zastosować go w kolumnie tabeli User. Jednakże istnieje spora rozbieżność we wsparciu dla 'User Defined Types' w różnych implementacjach baz
 - Możemy umieścić wszystkie pola związane z adresem jako kolumny tabeli User

Problem ziarnistości

```
create table USER (  
  USERNAME VARCHAR(15) NOT NULL PRIMARY KEY,  
  NAME VARCHAR(50) NOT NULL,  
  ADDRESS_STREET VARCHAR(50),  
  ADDRESS_CITY VARCHAR(15),  
  ADDRESS_STATE VARCHAR(15),  
  ADDRESS_ZIPCODE VARCHAR(5),  
  ADDRESS_COUNTRY VARCHAR(15)  
)
```

- Ziarnistość w klasie:
 - User class
 - Address class
 - String type
- Ziarnistość w bazie:
 - User table
 - Address columns

Problem dziedziczenia

- Główną siłą języków obiektowych jest możliwość budowania hierarchii klas.
- Relacyjny model danych nie oferuje żadnego mechanizmu który by wprost umożliwił deklaracje jednej tabeli jako pod typu innej
- Mechanizmy umożliwiające rozwiązanie tego problemu omówimy w dalszej części prezentacji

Problem tożsamości

- Model obiektowy - tożsame obiekty
 - Położenie w pamięci ($a == b$)
 - Identyczna zawartość ($a.equals(b)$)
- Model relacyjny
 - Wiersz tabeli
- Ogromna przepaść między obydwoma modelami!
- Dodatkowe problemy
 - Zmiana wartości klucza głównego (np. kolumna imię)
 - Obsługa cache
 - Obsługa transakcji

Problem powiązań

- Model obiektowy
 - Referencje do innych obiektów
 - Powiązania są ukierunkowane – dwustronne deklarujemy dwa razy
 - Możliwość powiązań wiele-do-wielu
- Model relacyjny
 - Klucze obce
 - Kierunek powiązania nie ma większego znaczenia (konstrukcje *join* i *projection*)
 - Tylko konstrukcje jeden-do-wielu i jeden-do-jednego (dla wiele-do-wielu konieczna 'przelotka')

Problem odwołań

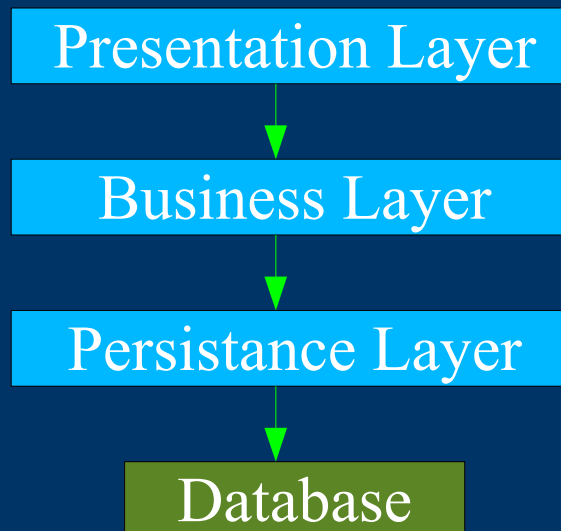
- Model obiektowy
 - Przechodzenie grafu obiektów:
 - `user.getAccountData().getNumber()`
- Model relacyjny
 - Kluczowym zagadnieniem przy efektywności dostępu jest ilość zapytań
 - Dla stworzenia dobrego zapytania musimy więc wiedzieć jaki fragment grafu obiektów chcemy przepytac ZANIM wykonamy zapytanie
- Inne problemy:
 - Jak stworzyć hierarchie obiektów nie odpytując na raz całej bazy lub nie generując dużej liczby zapytań?

Efekt...

- Około 30% kodu aplikacji poświęcone na pogodzenie tych sprzeczność
- Efekt zazwyczaj wysoce niezadowalający...
- Dodatkowe problemy:
 - Nie zgodność różnych RDBMS czyni przenośność trudną

Rozwiązanie

- Wprowadzenie oddzielnej warstwy która zajmie się połączeniem warstwy biznesowej z bazą danych
 - W architekturze warstwowej każda warstwa ma być niezależna od pozostałych



Object Relational Mapping

- Co się składa na rozwiązania ORM
 - API do wykonywania operacji CRUD na obiektach
 - Język zapytań lub API odnoszące się klas lub ich pól
 - Infrastruktura do tworzenia mapowań między modelem obiektywnym i relacyjnym
 - Implementacja obsługi
 - Transakcji
 - 'Dirty checking' – zapisywanie danych całej hierarchii obiektów przy zachowywaniu tylko jednego z nich
 - 'Lazy fetching' – inteligentne i zoptymalizowane pobieranie obiektów z hierarchii dopiero kiedy jest potrzebny
 - Cache

Modele

- Całkowicie relacyjny
 - Koncentracja na modelu relacyjnym i bezpośrednich zapytaniach SQL
 - Duże wykorzystanie procedur składowanych i znaczne przetrzucenie logiki biznesowej na bazę
- Lekki
 - Encje jako klasy mapowane ręcznie na model relacyjny
 - SQL ukryty przed logiką biznesową poprzez wzorce projektowe

Modele

- Średni
 - Koncentracja na modelu obiektowym.
 - SQL generowany dynamicznie
 - Wspiera powiązania między obiektami oraz zapytania w obiektowym języku
 - Raczej bez użycia procedur składowanych
- Pełny
 - Pełne i 'wyrafinowane' odwzorowanie obiektów – łącznie z dziedziczeniem, kompozycją i polimorfizmem.
 - Fetching and caching strategies
 - Pełna przejrzystość dla warstwy biznesowej.

Podstawowe problemy rozwiązań ORM

- 1) Jak wyglądają klasy 'utrwalane' (*persistance*)
 - Klasy czy komponenty – jaka ziarnistość
- 2) W jaki sposób definiujemy mapowanie
 - Metadane
- 3) W jaki sposób mapować dziedziczenie, polimorfizm, klasy abstrakcyjne i interfejsy?
- 4) W jaki sposób mapować tożsamość obiektów na tożsamość w bazie danych (klucze główne)
- 5) Interakcja 'persistance logic' i 'business logic'
- 6) Jaki jest cykl życia utrwalanego obiektu i w jaki sposób powiązać go z wierszem tabeli...

Podstawowe problemy rozwiązań ORM

- 7) Infrastruktura dla sortowania, wyszukiwania, agregacji
 - Czy wykonywać te zadania po stronie aplikacji czy po stronie bazy danych – wydajność
- 8) Jak efektywnie pobierać dane i powiązania
 - Minimalizacja liczby zapytań kontra pobieranie zbyt wielu danych na raz

Praktyczne rozwiązania

- Problem dziedziczenia
- Problem powiązań
- Problem tożsamości
- Cykl życia obiektów
- Z życia wzięte

Dziedziczenie

- Istnieją 3 podstawowe rozwiązania:
 - Tabela na każdą klasę
 - Tabela na hierarchie klas
 - Tabela na klasę dziedziczącą

Tabela na klasę(1)

- Nie staramy się przenieść dziedziczenia i polimorfizmu podczas tworzenia schematu BD
- Jedna tabela dla jednej klasy nie abstrakcyjnej.
- Utrudnione pobieranie obiektów implementujących ten sam interfejs/klasę abstrakc.
- Problem w razie zmiany nadklasy

Tabela na klasę(2)

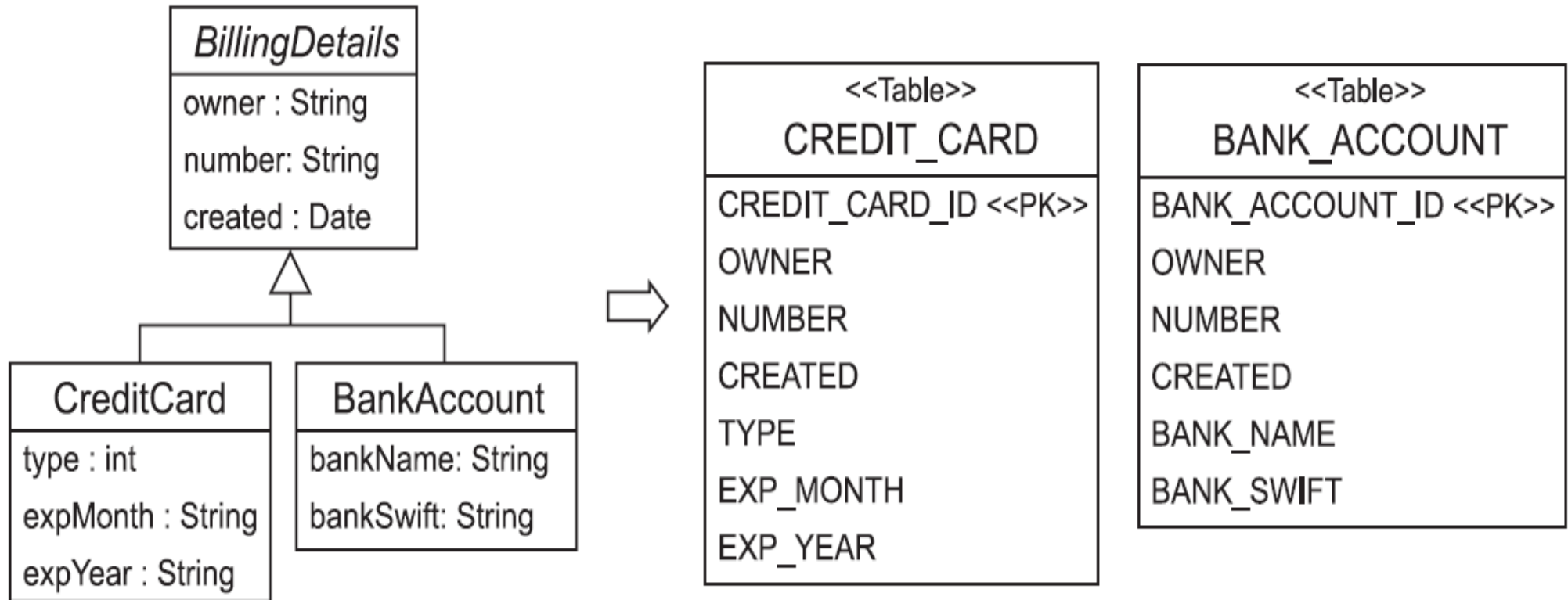


Tabela na hierarchie klas(1)

- Cała hierarchia klas jest obrazowana tylko przez jedną tabelę sumującą wszystkie atrybuty.
- Rodzaj podklasy jest identyfikowany przy pomocy dodatkowego pola.
- Duża rzadkość wypełnienia danymi takiej tabeli
- Jednak to rozwiązanie charakteryzuje się najlepszą wydajnością i prostotą w interpretowaniu modelu obiektowego na relacyjny.
- Wszystkie właściwości specyficzne podklas muszą mieć możliwość wpisania NULL, w innym wypadku trudno jest zaimplementować więzy integralności

Tabela na hierarchie klas(2)

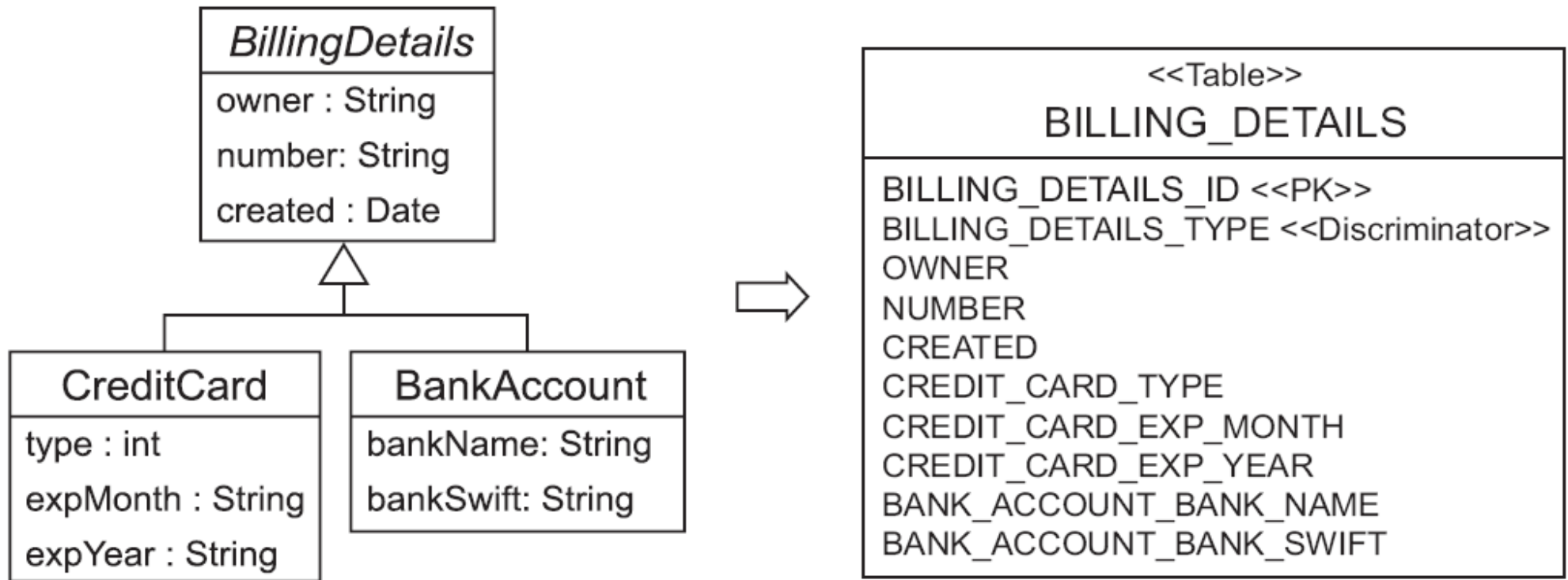
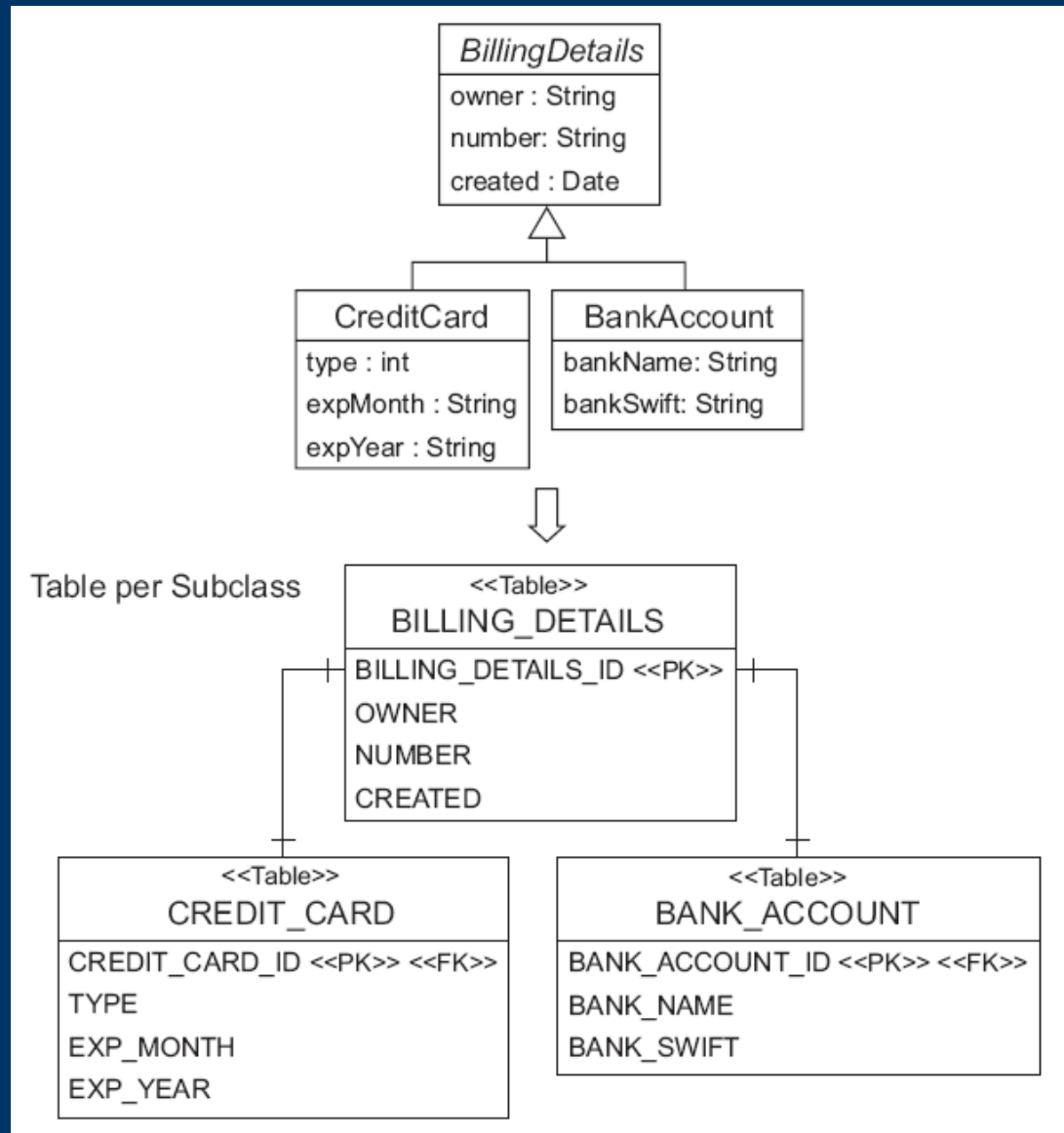


Tabela na klasę dziedziczącą(1)

- Każda podklasa reprezentowana jest przez swoją własną tabelę.
- Dziedziczenie obrazowane jest przez klucze obce
- Tabela zawiera kolumny z wartościami specyficznymi dla danej subklasy i klucz główny, który jest jednocześnie kluczem obcym do tabeli nadklasy.
- Dobra normalizacja schematu bazy danych.
- Łatwość wprowadzania modyfikacji do schematu.
- Łatwość stworzenia więzów integralności danych.
- Kiepska wydajność

Tabela na klasę dziedziczącą(2)



Co wybrać?

- Brak ścisłych reguł.
- Dla prostych odwzorowań należy starać się stosować model jedna tabela na jedną klasę.
- W razie potrzeby odwzorowania bardziej skomplikowanej struktury, zwłaszcza gdy podklasy deklarują sporo danych, należy zazwyczaj stosować model jedna tabela na jedną podklasę.

Powiązania

- Rodzaje:
 - Jeden-do-jednego
 - Jeden-do-wielu(wiele-do-jednego)
 - Wiele-do-wielu
- Wszystkie te powiązania mogą być jedno i dwustronne co daje w efekcie 8 rodzajai.
- Problem jest taki, że schemat relacyjny bazy danych nie odwzorowuje krotności powiązań pomiędzy tabelami, które są potem odwzorowywane na klasy.

Jeden-do-jednego(one-to-one)

```
@Entity
public class CreditCard java.io.Serializable {
    private int id;
    private Date expiration;
    private String number;
    private String name;
    private String organization;
    private Customer customer;
    ...

    @OneToOne(mappedBy="creditCard")
    public Customer getCustomer() {
        return this.customer;
    }
    public void setCustomer(Customer customer) {
        this.customer = customer;
    }

    ...
}
```

```
@Entity
public class Customer java.io.Serializable {
    private CreditCard creditCard;
    ...

    @OneToOne(cascade={CascadeType.ALL})
    @JoinColumn(name="CREDIT_CARD_ID")
    public CreditCard getCreditCard() {
        return creditCard;
    }
    public void setCreditCard(CreditCard card) {
        this.creditCard = card;
    }
    ...
}
```

Jeden-do-wielu(Wiele-do-jednego) one-to-many(many-to-one)

```
@Entity
public class Customer implements java.io.Serializable {
    ...
    private Collection<Phone> phoneNumbers = new ArrayList<Phone>();
    ...
    @OneToMany(cascade={CascadeType.ALL})
    @JoinColumn(name="CUSTOMER_ID")
    public Collection<Phone> getPhoneNumbers() {
        return phoneNumbers;
    }
    public void setPhoneNumbers(Collection<Phone> phones) {
        this.phoneNumbers = phones;
    }
}
```

```
CREATE TABLE PHONE
(
    ID INT PRIMARY KEY NOT NULL,
    NUMBER CHAR(20),
    TYPE INT,
    CUSTOMER_ID INT
)
```

Wiele-do-wielu(many-to-many)

```
@Entity
public class Reservation java.io.Serializable {
    ...
    private Set<Customer> customers = new HashSet<Customer>();
    ...
    @ManyToMany
    @JoinTable(table="RESERVATION_CUSTOMER",
              joinColumns={@JoinColumn(name="RESERVATION_ID")},
              inverseJoinColumns={@JoinColumn(name="CUSTOMER_ID")})
    public Set<Customer> getCustomers() { return customers; }
    public void setCustomers(Set customers);
    ...
}
```

```
CREATE TABLE RESERVATION_CUSTOMER
(
    RESERVATION_ID INT,
    CUSTOMER_ID INT
)
```

```
@Entity
public class Customer java.io.Serializable {
    ...
    private Collection<Reservation> reservations = new ArrayList<Reservation>();
    ...
    @ManyToMany(mappedBy="customers")
    public Collection<Reservation> getReservations() {
        return reservations;
    }
    public void setReservations(Collection<Reservation> reservations) {
        this.reservations = reservations;
    }
    ...
}
```

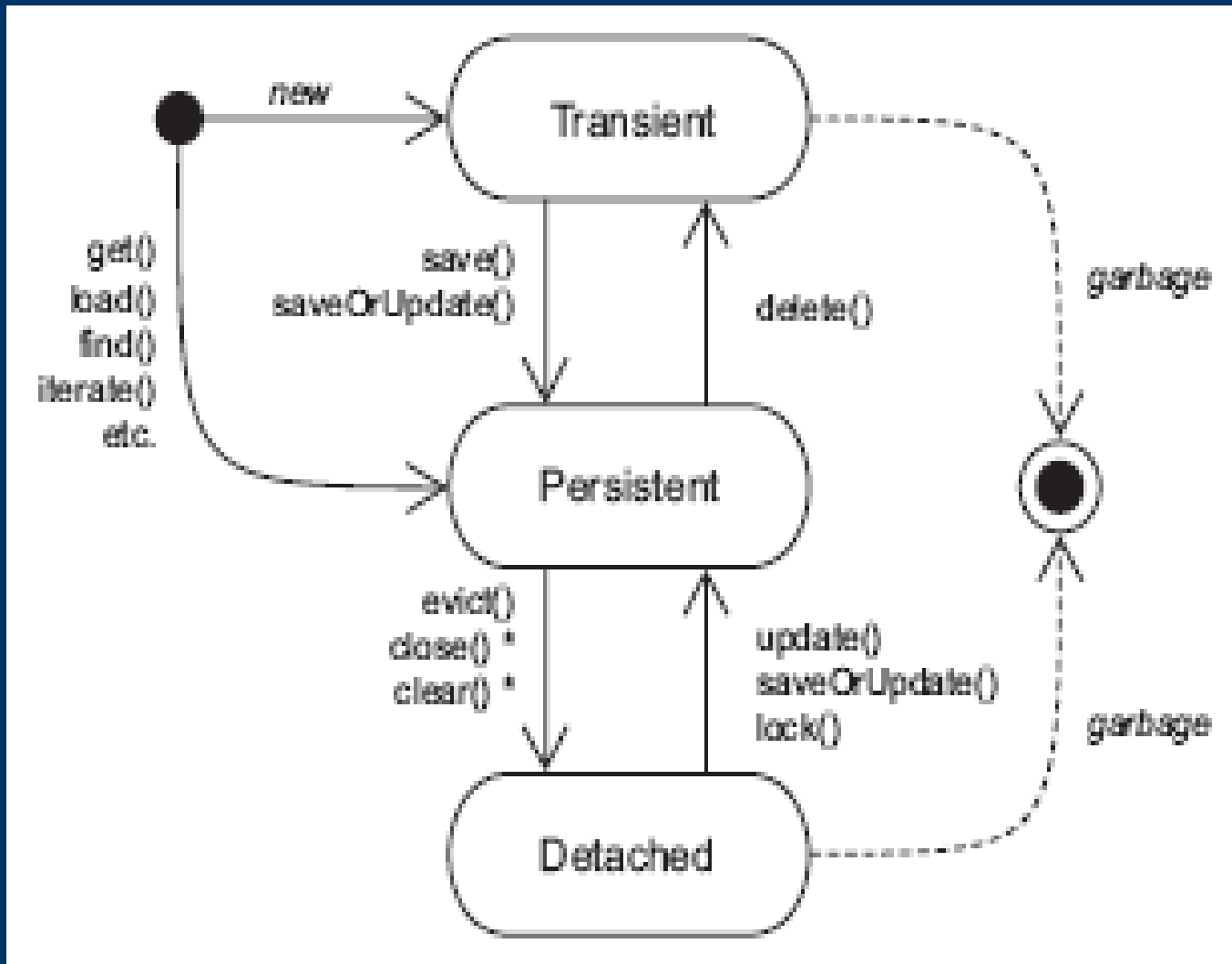
Tożsamość

- Łatwo doprowadzić do sytuacji kiedy dwie różne instancje obiektu wskazują na ten sam wiersz
- Strategie
 - Implementacja metod `.equals()` i `.hashCode()`
 - Porównywanie obiektów po ID związanym z PK
 - Porównywanie obiektów po wartości
 - Business key – zestawienie kilku kluczowych pól obiektu

Pobieranie ('fetching')

- Istnieje wiele strategii - m. in.:
 - Immediate fetching – powiązany obiekt jest pobierany od razu poprzez sekwencje operacji *'read'*
 - Lazy fetching – powiązany obiekt jest pobierany przy jego pierwszym użyciu – skutkuje to nowym zapytaniem chyba że znajdował się w cachu
 - Eager fetching – powiązane obiekty są pobierane razem z rodzicem poprzez konstrukcje *'outer join'*
 - Batch fetching – usprawnienie *'lazy fetching'* polegające na pobraniu większej ilości obiektów danego typu – poprzez dołożenie więcej ID w klauzuli *'where'*

Cykl życia obiektów



Cykl życia cd.

Applica

Z życia wzięte

- <http://www.iem.pw.edu.pl/~dawidowb/ZSBD/>
- <http://www.iem.pw.edu.pl/~dawidowb/ZSBD/DzienniczekDB.JPG>
- <http://www.iem.pw.edu.pl/~dawidowb/ZSBD/DzienniczekEJB%20Diagram%20ver2.png>
- <http://www.iem.pw.edu.pl/~dawidowb/ZSBD/ForumsDatabase.png>
- <http://www.iem.pw.edu.pl/~dawidowb/ZSBD/ForumsDatabase-inherit.png>

