

Nowy powiew od Słońca: EJB 3.0



eRudis

Wstęp

Piotr Kochański

p.kochanski@erudis.pl

www.erudis.pl

konsultant firmy **Erudis**

narzędzia:

TestTrack Pro:

śledzenie błędów

SurroundSCM

wersjonowanie

QAWizard

testowanie

Enterprise Architect

modelowanie

szkolenia:

Java

Modelowanie w UML-u

Zarządzanie zmianą

Zbieranie wymagań

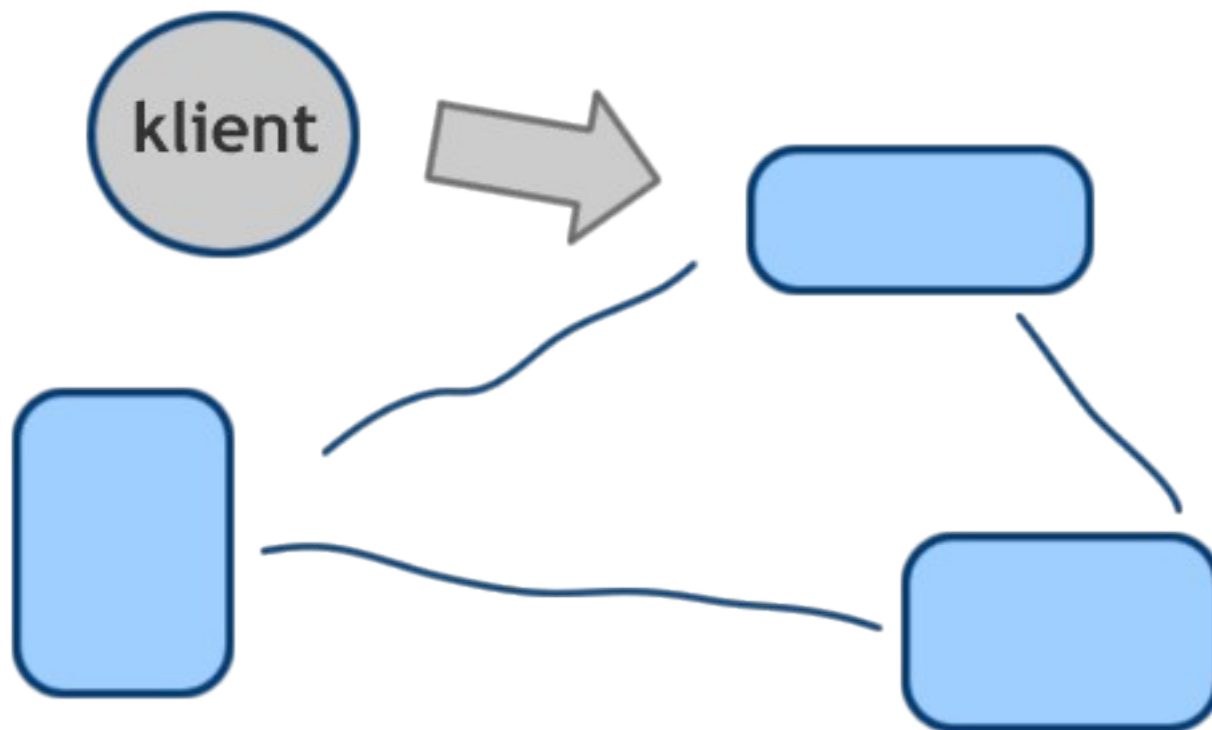
PRINCE2

Agenda

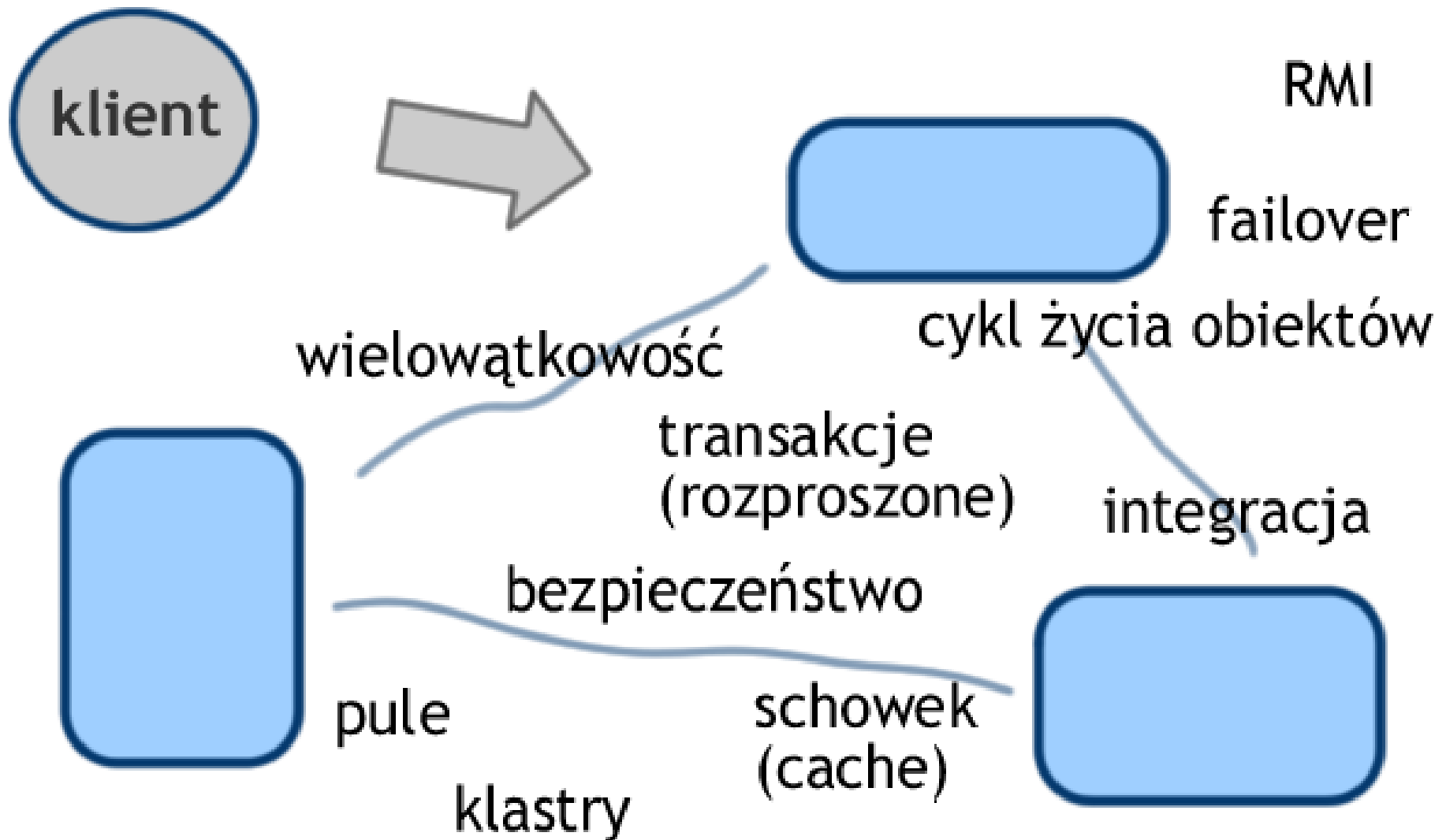
- EJB - podstawowe informacje
- Problemy EJB 1.X i 2.X
- Zmiany wprowadzone w EJB v. 3.0
 - uproszczone API
 - nowe komponenty encyjne
 - wzorce i antywzorce
- Przykład, czyli EJB w 5 min.... *live!*

Co to jest EJB

- Problem: chcemy tworzyć oprogramowanie w architekturze komponentowej?
 - klienci i komponenty muszą się komunikować przez sieć

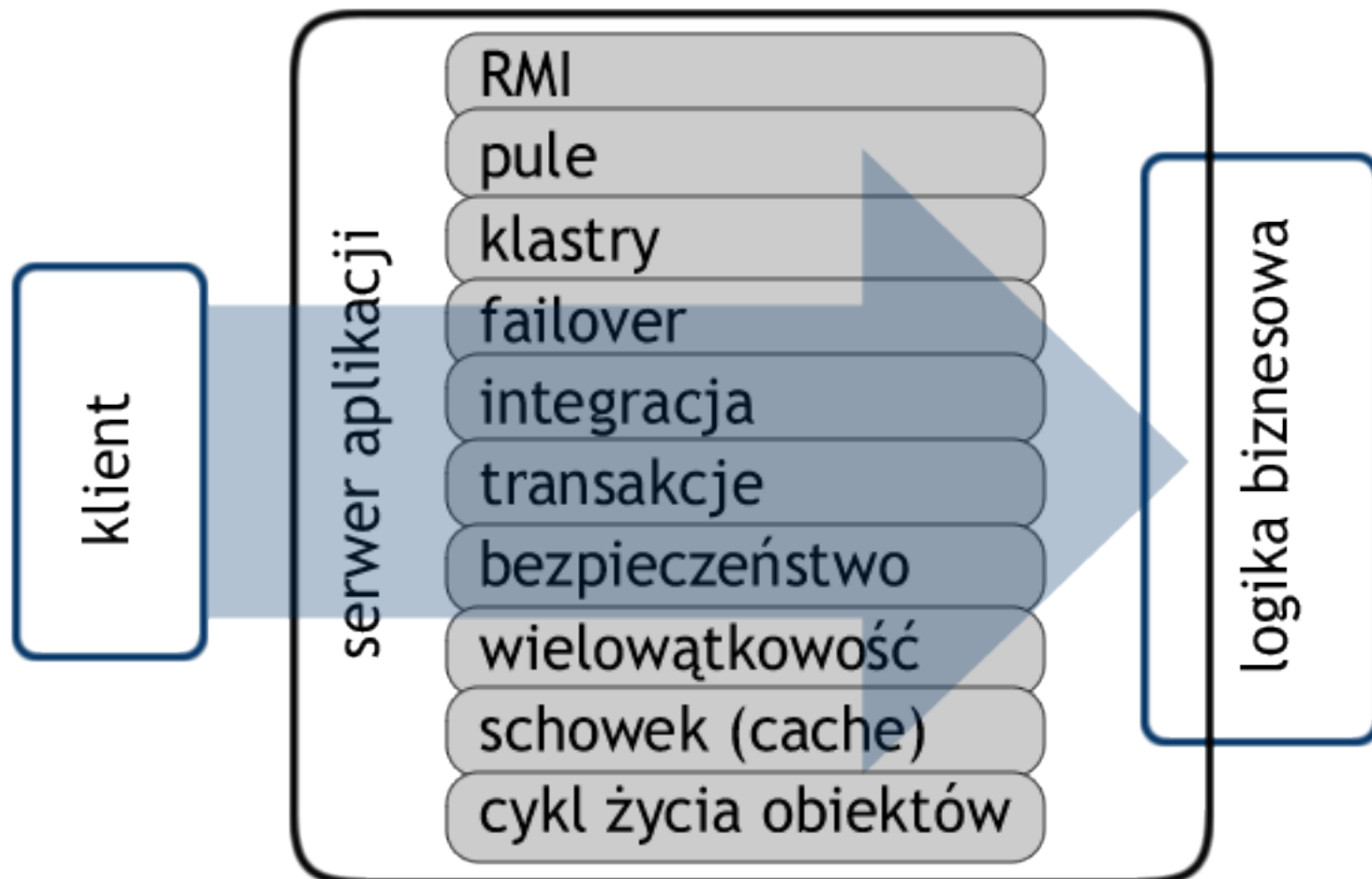


Problemy architektury komponentowej



Middleware

- EJB jest warstwą pośrednią która dostarcza nam potrzebnych usług



EJB jako warstwa pośrednia

- Komponenty EJB żyją na serwerze i są przez niego zarządzane
- my zajmujemy się tylko „logiką biznesową” aplikacji, potrzebnych usług dostarcza serwer
- mamy swobodę wyboru klienta: telefon, WWW, aplikacja GUI, usługa sieciowa

Zalety EJB

- Standard przemysłowy
- Wsparcie producentów serwerów aplikacji (komercyjnych i darmowych, open source)
- niezawodność
- Bezpieczeństwo
- Łatwa obsługa transakcji (także bardzo złożonych)
- Możliwość wykorzystania przez klientów dowolnego typu

Wady EJB 2.X

- Złożona technologia
- Duży nakład pracy przy tworzeniu komponentów i konfiguracji
 - rozwiązanie: XDoclet
- Zbyt mała elastyczność w zakresie komunikacji z bazą danych
 - komponenty encyjne EJB mają niewystarczającą funkcjonalność jako ORM
 - Mało funkcjonalny język kwerend EJB-QL

Wady EJB 2.X, C.D.

- Brak możliwości wykorzystania dziedziczenia, polimorfizmu przez komponenty EJB
- Utrudnione przeprowadzanie testów jednostkowych (np. JUnitem)
- Utrudniona migracja aplikacji między serwerami różnych producentów, ze względu na komponenty encyjne
 - każdy serwer ma swoje własne pliki konfiguracyjne

EJB 3.0

- Radykalna zmiana
- Specyfikacja jest rozbita na dwie części
 - główną (core): komponenty sesyjne, MDB
 - obsługa trwałości - komponenty encyjne (**javax.persistence**)
- Pełna wsteczna kompatybilność
- Wykorzystane doświadczenia
 - mechanizm *Annotations* z XDoclet
 - obsługa trwałości z Hibernate

EJB 2.X - przykład

```
public class CalcBean implements
    javax.ejb.SessionBean {

    //metody obsługi cyklu życia komponentu
    //z interfejsu SessionBean: ejbCreate() ,
    //ejbRemove() , ejbPassivate() , ...

    public double add(double x, double y) {
        return x + y;
    }
}
```

- zależność od zewnętrznego interfejsu
- wymuszona implementacja metod obsługi cyklu życia komponentu
- trzeba utworzyć jeszcze dwa interfejsy

EJB 3.0 - przykład

```
@Stateless
@RemoteBinding (jndiBinding="myapp/calc")
@Remote ({Calc.class})
public class CalcBean implements Calc {
    public double add(double x, double y) {
        return x + y;
    }
}
```

```
public interface Calc {
    public double add(double x, double y);
}
```

- Calc jest zwykłym interfejsem
- Kod komponentu nic nie wie, że będzie udostępniany zdalnie

EJB 3.0 vs. 2.X: najważniejsze zmiany

- Nie jest potrzeby deskryptor wdrożenia `ejb-jar.xml`
 - Konfiguracja odbywa się poprzez mechanizm *Annotations* (`@Stateless`, ...)
- Uproszczony interfejs programistyczny (API)
 - komponenty nie muszą implementować żadnego zewnętrznego interfejsu ani obowiązkowych metod
 - wywołań zwrotnych `ejbCreate()`, `ejbPassivate()` używamy miarę potrzeb
- Nie jest potrzebny interfejs bazowy (home interface)

EJB 3.0: komponenty encyjne

- Komponenty encyjne zostały całkowicie zmodyfikowane
 - są zwykłymi klasami Java, których obiekty mogą być zapisywane w bazie danych
- Możemy używać wszystkich dostępnych mechanizmów OO
 - dziedziczenie,
 - polimorfizm,
 - ...



EJB 3.0: komponenty encyjne, C.D.

- Komponenty encyjne mają pełną funkcjonalność mostu relacyjno- obiektowego (podobnie jak Hibernate)
- EJB-QL - rozbudowany, obiektowy język kwerend (SELECT, DELETE, UPDATE)
- Można używać kwerend SQL
- Komponenty encyjne mogą być używane niezależnie od serwera aplikacji, jak zwyczajny most R-O

Komponent encyjny: przykład

```
@Entity
@Table(name = "notatka")
public class Notatka implements
    Serializable{
    private long id;
    private String tytul;
    private String tresc;

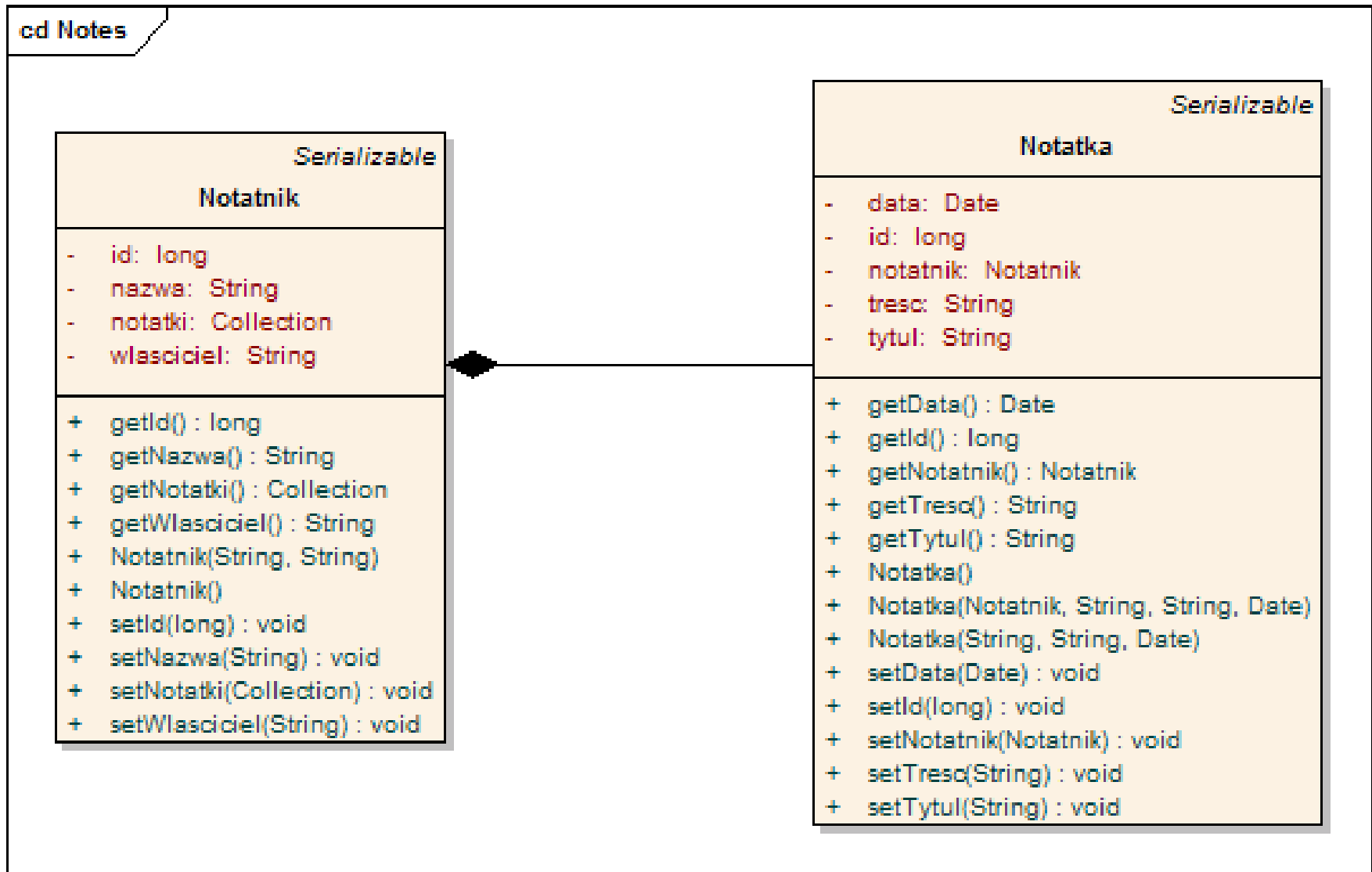
    public Notatka(){}
    public Notatka(String t, String tr) {
        tytul = t; tresc = tr;
    }
    //pozostałe metody get/set pominięte
    @Id @GeneratedValue
    public long getId() {
        return id;
    }
}
```



eRudis

Związki między encjami

- Diagram...



Związki między encjami

```
@Entity @Table(name = "notatnik")
public class Notatnik{
    private Collection notatki;

    @OneToMany(... targetEntity=Notatka.class)
    public Collection getNotatki() {
        return notatki;
    }
}
```

```
@Entity @Table(name = "notatka")
public class Notatka{
    private Notatnik notatnik;

    @ManyToOne @JoinColumn(name = "notatnik_id")
    public Notatnik getNotatnik() {
        return notatnik;
    }
}
```

Praca z encjami (CRUD)

```
EntityManager em;
```

```
//dodawanie
```

```
Notatka n = new Notatka("tytul", "tresc");  
em.persist(n);
```

```
//zmiany
```

```
n.setTresc("inna treść");  
em.merge(n);  
em.flush();
```

```
//szukamy notatki o id = 123
```

```
Notatka n = em.find(Notatka.class, 123);
```

```
//usuwanie
```

```
em.remove(n);
```



EJB-QL, co nowego?

- dynamiczne kwerendy:
 - `EntityManager.createQuery()`
 - `EntityManager.createNativeQuery()`
- EJB-QL - praktycznie pełna funkcjonalność SQL-a: group by, having, wewnętrzne i zewnętrzne złączenia, podzapytania, update, delete
- rzutowanie zapytania na dowolny typ

```
SELECT NEW CustomerInfo(c.id, c.info, o.count)
FROM Customer c JOIN c.orders o
WHERE o.count > 100
```

Nowy EJB-QL, przykłady

```
Query q = em.createQuery(  
    "select from Notatka n where  
        n.tytul = :tyt");  
q.setParameter("tyt", "taki tytuł");  
List notatki = q.getResultList();
```

```
Query del = em.createQuery("delete from  
    Notatka n where n.tytul = :tyt");  
del.setParameter("tyt", "taki tytuł");  
del.executeUpdate();
```

```
Query updt = em.createQuery("update Notatka  
    n set n.tresc = 'ala ma kota'  
        where n.tytul = :tyt");  
updt.setParameter("tyt", "taki tytuł");  
updt.executeUpdate();
```

Odwrócenie kontroli

- Inversion of Control/Dependency Injection
- Skąd wziąć referencje do potrzebnych nam zasobów:
 - instancji innych komponentów EJB
 - źródeł danych (DataSource), itp. itd.
- Dwie możliwości
 - wzorzec *Service Locator*, np. wyszukiwanie przez JNDI: `Context.lookup("nazwa")`
 - IoC/DI
- IoC: określamy, jakie zasoby są nam potrzebne, a kontener nam ich dostarcza. Nie szukamy ani nie inicjujemy tych zasobów!

Odwrócenie kontroli: przykład

- Skąd wziąć instancję `EntityManager`-a?

```
@Stateless
@RemoteBinding (jndiBinding="notes/sekretarz")
@Remote ({Sekretarz.class})
public class SekretarzBean
    implements Sekretarz {

    @PersistenceContext
    protected EntityManager em;

    public void dodajNotatke(String tyt,
        String tr) {
        Notatka n = new Notatka(tyt, tr);
        em = new EntityManager(); //nie!
        em = Context.lookup("nazwa_jndi"); //nie!
        em.persist(n);
    }
}
```

Odwrócenie kontroli: inny przykład

```
@Stateless
public class ABean implements A {

    @Resource (mappedName="java:/DefaultDS")
    DataSource myDb;

    @EJB (beanName="CalcBean")
    Calc cal;

    public void metodaDb() {
        Connection conn = myDb.getConnection();
    }

    public double metodaCalc() {
        return cal.add(2, 2);
    }
}
```

Wzorce i anty-wzorce

- Odwrócenie kontroli zamiast Service Locator (tam, gdzie warto)
- Nie ma potrzeby stosować Data Objects/Value Objects
 - komponenty encyjne w wersji 2.X z lokalnym interfejsem były „przyklejone” do serwera, nie można było ich zwracać klientowi
 - w wersji 3.0 nie ma tego problemu, encje mogą być
 - złączane z warstwą trwałości
 - odłączane od niej i modyfikowane przez klienta
 - ponownie przyłączane do warstwy trwałości

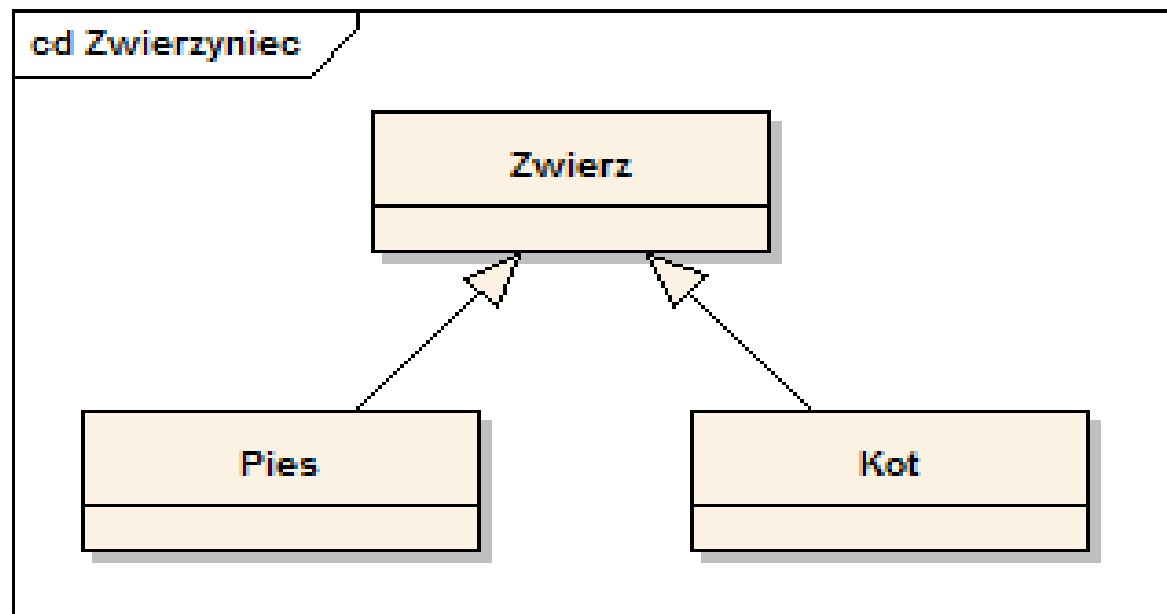
Model obiektowy

- Komponenty EJB pozwalają na tworzenie poprawnego modelu obiektowego.
- Nie jest to proste, zwłaszcza dla komponentów encyjnnych
 - trzeba utworzyć przejście pomiędzy światem obiektów a światem SQL-a
- EJB 3.0 mają potrzebne do tego mechanizmy



Dziedziczenie

- Encje mogą dziedziczyć po sobie
 - hierarchia dziedziczenia jest automatycznie odwzorowana w bazie danych
 - kwerendy EJB-QL mogą być polimorficzne
 - przykład...



Dziedziczenie, przykład

```
@Entity
@Inheritance(strategy =
    InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "TYP_ZWIERZA",
    discriminatorType =
    DiscriminatorType.STRING)
public class Zwierz implements
    Serializable{..}
```

```
@Entity
@Inheritance(strategy =
    InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(discriminatorType =
    DiscriminatorType.STRING)
@DiscriminatorValue("PIES")
public class Pies extends Zwierz{...}
```



Dziedziczenie, szczegóły

- Strategie odwzorowania dziedziczenia w bazie
 - **SINGLE_TABLE** - jedna tabela z polem, które określa klasę konkretną (wymagane przez EJB 3.0 spec.)
 - **JOINED** - tabela złączona z bazową na każdą klasę dziedziczącą
 - **TABLE_PER_CLASS** - oddzielna tabela dla każdej klasy dziedziczącej
- Dla **SINGLE_TABLE** tabela wygląda następująco

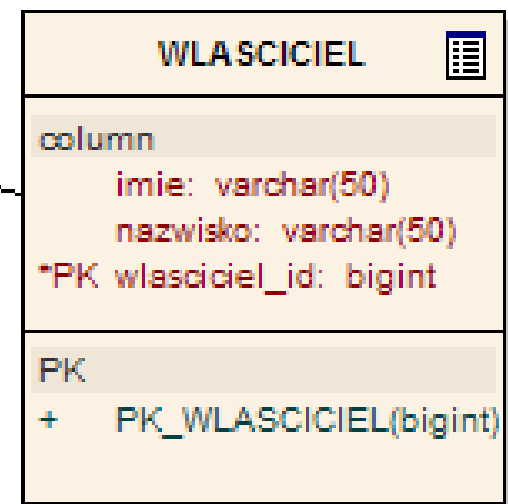
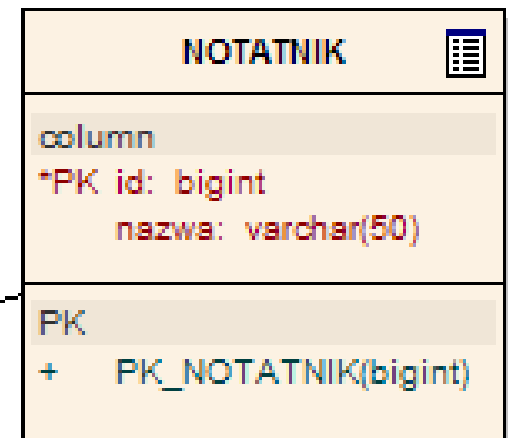
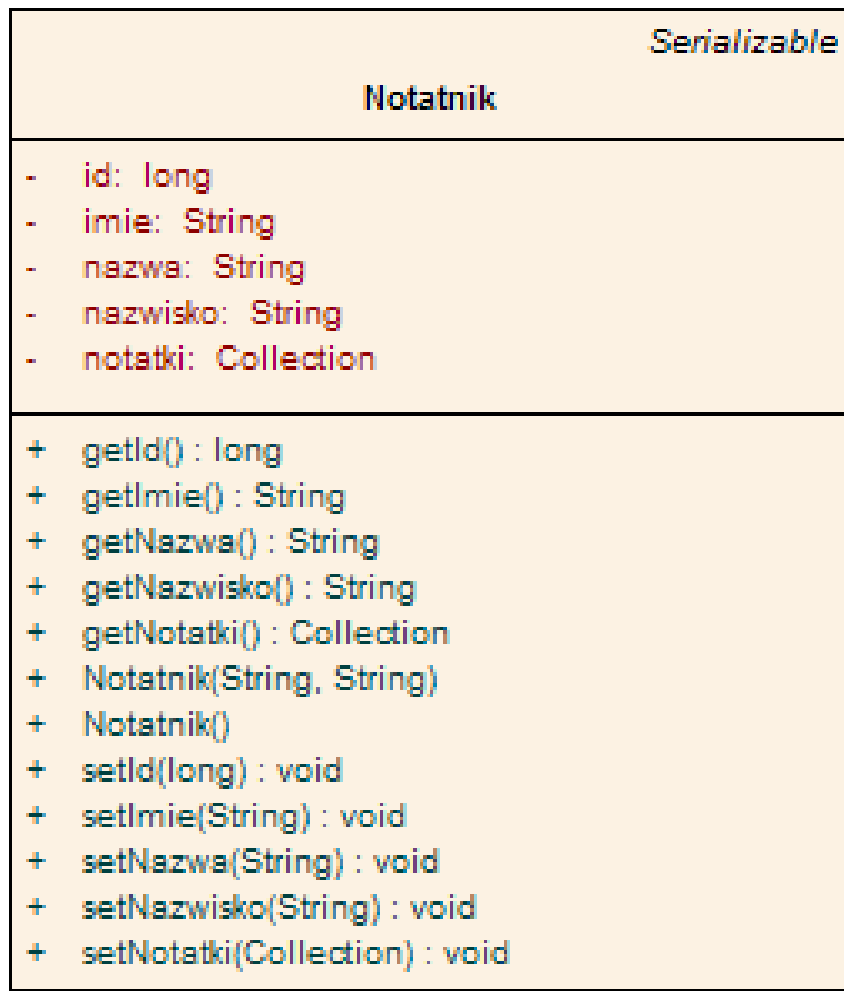
```
create table ZWIERZ ( ID integer primary key,  
                    TYP_ZWIERZA varchar,  
                    NAZWA varchar,  
                    LICZBA_LAP ind) ;
```

Model obiektowy

- Komponenty encyjne pozwalają na tworzenie poprawnego i wygodnego modelu obiektowego
- Można odwzorować **wiele komponentów** encyjnych na **jedną tabelę** w bazie danych
- Można odwzorować **jeden komponent** na **wiele tabel** w bazie danych

Jeden komponent, wiele tabel

cd Multiple Mapping



Jeden komponent, wiele tabel

```
@Entity
@Table(name = "NOTATNIK")
@SecondaryTable(name = "WLASCICIEL")
@JoinColumn(name = "WLASCICIEL_ID")
public class Notatnik implements
    java.io.Serializable{

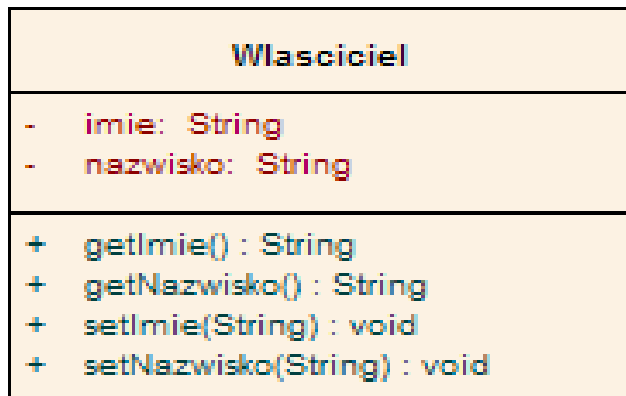
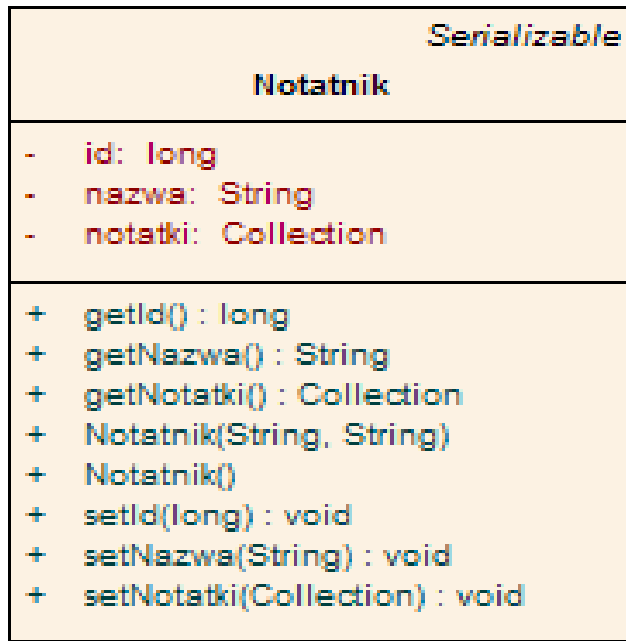
    public String getNazwa(){
        return nazwa;
    }

    @Column(name = "IMIE",
            table = "WLASCICIEL")
    public String getImie(){
        return imie;
    }
}
```



Wiele klas, jedna tabela

cd Embeddable



Wiele klas, jedna tabela

- Zwykła klasa Java, która będzie zawarta w klasie komponentu EJB

```
@Embeddable  
public class Wlasciciel{  
    private String imie;  
    private String nazwisko;  
  
    //metody get/set ...  
}
```

Wiele klas, jedna tabela, C.D.

```
@Entity
@Table(name = "NOTATNIK")
public class Notatnik{
    private int id;
    private Wlasciciel wlasciciel;
    //pozostałe pola

    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name = "imie",
            column = @Column(name = "IMIE")),
        @AttributeOverride(name = "nazwisko",
            column = @Column(name = "NAZWISKO"))
    })
    public Name getWlasciciel(){
        return wlasciciel;
    }
}
```



Model obiektowy - podsumowanie

- EJB 3.0 pozwala tworzyć obiektowy model aplikacji bez żadnych ograniczeń
 - dla komponentów sesyjnych/MDB jest to oczywiste
 - dla komponentów encyjnych mamy most relacyjno-obiektowy, pozwalający mapować dowolnie obiekty na model relacyjny danych

Message Driven Beans

- Komponenty zorientowane na komunikaty mają uproszczone API

```
@MessageDriven(activationConfig =
    { @ActivationConfigProperty
      (propertyName="destinationType",
       propertyValue="javax.jms.Queue"),
      @ActivationConfigProperty(
       propertyName="destination",
       propertyValue="queue/testmdb") })
public class TestMDB implements
    MessageListener{
    public void onMessage(Message msg){
        System.out.println("jest wiadomość");
    }
}
```

Przykład na żywo

- Serwer aplikacji: JBoss 4.0.4RC1
- Eclipse + JBossIDE
- Komponent sesyjny, który zwraca aktualną datę i godzinę

Podsumowanie

- EJB 3.0
 - łatwiejsze w przyswojeniu
 - większa funkcjonalność (komponenty encyjne)
 - szybsze tworzenie oprogramowania
- Prezentacja + przykładowy projekt:
<http://www.erudis.pl/sc/ejb3>
- Kontakt: p.kochanski@erudis.pl

