Projekt BD

ORM, JUnit, JavaDoc, log4j

Co powinniśmy mieć

- Aplikacja używająca hibernate
- Encje – dla każdej tabeli
  - Asocjacje pomiędzy encjami
- Warstwa DAO umożliwiająca CRUD dla każdej tabeli
- Pytania i wątpliwości?

Mój przykład

- Encje
  - Car
  - Owner
  - Equipment
- Car ma "w sobie"
  - Obiekt Owner
  - Listę obiektów Equipment

Bean Car

```
public class Car {
    int idc;
    String make;
    String model;
    String regNum;
    Double price;
    Owner owner;
    Set<Equipment> eqs = new HashSet<Equipment>();
}
```

Kod wydobywający samochód

- Kod wydobywający samochód wraz z obiektami powiązanymi

```
Car car = session.find(Car.class,1)
System.out.println(car.getModel());
System.out.println(car.getOwner().getName());
for(Equipment eq:car.getEqs()) {
    System.out.println(eq.getIde()+" "+eq.getDesc());
    }
```

- Działa to tylko w ramach sesji!

Kontrola zapytań

- Sprawdzenie, jakie zapytania do bazy wysyła Hibernate:
- Wpis do hibernate.cfg.xml

```
<property name="hibernate.show_sql"
   value="true"/>
```

Po zamknięciu sesji

```
Session session = HibFactory.getInstance().openSession();
Car car = session.find(Car.class,1)
session.close();
System.out.println(car.getModel());
System.out.println(car.getOwner().getName()); // to działa
for(Equipment eq:car.getEqs()) { // to nie działa (LazyInitializatioExc)
    System.out.println(eq.getIde()+" "+eq.getDesc());
    }
```

- Automatycznie wyciągane są tylko obiekty ManyToOne
- Inne trzeba dociągnąć samemu w ramach sesji

FetchType

- LAZY
  - Ze źródła danych wyciągany jest tylko obiekt
  - Przy wywołaniu obiektów powiązanych (car.getEquipment()) są one „dociągane” na życzenie
  - Jeśli sesja zamknięta, wywołanie obiektów powiązanych powoduje LazyInitializationException
- EAGER
  - Wyciąga z bazy obiekt i od razu wszystkie powiązane
  - Łatwo wyciągnąć zbyt dużo danych na raz!
- Ustawienie:
  - @ManyToOne(fetch=FetchType.LAZY|EAGER)

Ćwiczenie

- Przetestować działanie FetchType na własnym modelu danych
- Zastanowić się gdzie zastosować EAGER a gdzie LAZY

Problemy z testowaniem

- Jak testować wydzielone fragmenty programu
- Za każdym razem konieczność przygotowania parametrów – wartości początkowych do testu
- Kolejne zmiany aplikacji mogą powodować, że poprzednie – przetestowane – elementy przestają działać
- Testowanie jest żmudne i czasochłonne
- Rezultat: Wiele jest programów testowanych przez użytkowników końcowych

Usprawnienie testowania

- Łatwe tworzenie testów
- Możliwość ich zapisywania i wielokrotnego uruchamiania
- Tworzenie zestawów testów i ich wspólne uruchamianie
- Ułatwienie przygotowania danych dla testów
- Najpopularniejsze narzędzie: JUnit

Użycie JUnit

- Narzędzie do tworzenia testów jednostkowych
- Budowa testu – metody z adnotacjami
- @BeforeClass, @AfterClass – przed i po wszystkich testach
- @Before, @After – przed i po każdym teście
- @Test – metoda testująca, bezparametrowa, zwracająca void

Przykładowy test JUnit4

```
import static org.junit.Assert.*;
public class CarTest {
    CarDAO carDAO;
    @BeforeClass
    public void init() { carDAO = new CarDAO();   }
    @Test
    public void testGetClient() {
        Car car = carDao.getCar(1);
        Assert.assertEquals("Zwrocono nieprawidlowy samochód",
                1, cln.getNo());
        car = carDao.getCar(-1);
        Assert.assertNull("Zwrocono samochód którego nie ma", car);
    }
}
```

Asercje

- assertEquals([String message,]Object o1, Object o2)
- assertTrue
- assertFalse
- assertNull
- assertNotNull
- assertSame
- assertNotSame
- fail

Zalecane praktyki

- Testy w osobnym folderze test
- Struktura pakietów jak w folderze src
  - aby był dostęp do metod protected
- Osobne testy dla każdej z klas i każdej metody w klasie
- UWAGA! Testować tylko to, czego działania nie jesteśmy w 100% pewni (nie np. gettery i settery)
- Zasada generalna:
  - **`Test until fear turns to boredom`**

Ćwiczenie

- Przygotowanie testów JUnit dla swojej aplikacji

Dokumentowanie Javadoc

- Aplikacja Javadoc (element JDK) tworzy dokumentcję na podstawie komentarzy w źródłach
- Wywołanie: javadoc <parametry>
- Przetwarza komentarze przed klasami i metodami
- Specjalna składnia:

```
/**
 * Dodanie nowego samochodu
 * @param car samochód do dodania
 */
public void add(Car car) {
```

Niektóre adnotacje

- @param
- @return
- @author
- @deprecated
- @throws

```
/**
 * Wydobycie z bazy listy samochodów o podanej marce
 * @param make poszukiwana marka zamochodu
 * @return lista znalezionych samochodów
 */
```

Przetwarzanie komunikatów

- Często podczas testowania w kodzie umieszczamy logowanie komunikatów na temat pracy aplikacji
- Po ukończeniu i przetestowaniu aplikacji to logowanie ograniczamy
- Biblioteka Log4J bardzo ułatwia nam to zadanie
- Pozwala na
  - wpisywanie w kod komunikatów o różnych poziomach
  - konfiguracje miejsca przesyłania komunikatów

Logger

- Logger – obiekt, do którego przesyłane są komunikaty
- Loggery ustawione są w hierarchii – najwyżej jest RootLogger
- Loggery dziedziczą po sobie kaskadowo właściwości
- Użycie loggera w programie:

```
Logger logger = Logger.getLogger(Start.class);
...
logger.debug("Start aplikacji");
```

Poziomy logowania

- trace
- debug
- info
- warn
- error
- fatal

Appender

- Każdy Logger ma przypisany Appender
- Rodzaje appenderów:
  - ConsoleAppender
  - FileAppender
  - SocketAppender
  - JMSAppender
  - NTEventLogAppender
  - SyslogAppender
  - ...
- Dla appendera możliwe ustawienie layoutu

Konfiguracja

- Domyślnie w pliku log4j.properties
- Przykład:

```
# rootLogger ma poziom DEBUG appender A1
log4j.rootLogger=DEBUG, A1
# Ustawienie A1 na ConsoleAppender
log4j.appender.A1=org.apache.log4j.ConsoleAppende
   r
# Ustawienie PatternLayout dla A1
log4j.appender.A1.layout=org.apache.log4j.Pattern
   Layout
log4j.appender.A1.layout.ConversionPattern=%-4r
   [%t] %-5p %c %x - %m%n
```

ConversionPattern

- **%C** nazwa klasy z której zostało wysłane zdarzenie
- **%d** data zdarzenia (przykład: %d{HH:mm:ss,SSS})
- **%m** komunikat wysłany z aplikacji
- **%n** separator linii
- **%p** priorytet zdarzenia
- **%%** pojedynczy znak %
- **%M** nazwa metody z której zostało wysłane zdarzenie
- **%L** numer linii z której zdarzenie pochodzi
- **%F** nazwa pliku z którego pochodzi zdarzenie
- **%r** ilość milisekund, które zdążyły upłynąć od startu aplikacji
- **%t** nazwa wątku z którego zostało wysłane zdarzenie

Przykładowe modyfikatory

- **%20c** dodaje spacje z lewej strony jeśli nazwa kategorii jest krótsza niż 20 znaków
- **%-20c** dodaje spacje z prawej strony jeśli nazwa kategorii jest krótsza niż 20 znaków
- **%.30c** przycina kategorię pozostawiając ostatnie 30 znaków
- **%20.30c** wypełnia z lewej strony nazwę kategorię spacjami do 20 znaków oraz przycina początkowe znaki jeżeli jest dłuższa on 30 znaków

Przykładowy plik

```
# rootLogger ma poziom DEBUG appender A1
log4j.rootLogger=DEBUG, A1
# Ustawienie A1 na ConsoleAppender
log4j.appender.A1=org.apache.log4j.ConsoleAppender
# Ustawienie PatternLayout dla A1
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-4r [%t] %-5p %C - %m%n

log4j.logger.kurs.test=INFO, test
log4j.appender.test=org.apache.log4j.FileAppender
log4j.appender.test.File=test.log
log4j.appender.test.layout=org.apache.log4j.PatternLayout
log4j.appender.test.layout.ConversionPattern=%-20.20C %d{HH:mm:ss,SSS}
    %M[%L] (%m)%n
```

Ćwiczenie

- Dodanie komentarzy JavaDoc
- Zamiana wszystkich System.out.println na log.[poziom]