Numer 4/2009(6)

Java eXPress

CZASOPISMO DLA DEWELOPERÓW JAVA

Gradle: Mocarne narzędzie do budowy projektów

GORM – Grails Object Relational Mapping

Behaviour driven development z easyb

Signal Framework dla platformy Java ME

BONUS: Refaktoryzacja cz.II

>>Wstęp do Log4j >>Express killers cz.V >>Recenzja GroovyMag 11/2009

NEW YEAR(2010).HAPPY = HAPPINESS.HIGH;

Jak przystało na zakończenie roku, parę słów co nam się udało osiągnąć w zeszłym roku i co planujemy na kolejny.

A więc witajcie po raz szósty. Tak, sporo tego się nazbierało. W sumie to prawie 300 stron wiadomości o Javie. I to wszystko dzięki współpracy kilkunastu osób, Waszemu zaangażowaniu i wsparciu sponsorów. A od numeru czwartego zaczęliśmy tłumaczyć Java exPress na język angielski. Sporo pracy, ale satysfakcja większa. Teraz każdy może sięgnąć po Java exPress. Java exPress był obecny na prawie każdej większej imprezie Javowej w Polsce, byliśmy patronem Devoxx i Jazoon, udało nam się zorganizować 3 edycje COOLuarów. Rozpoczęliśmy konkurs Java Geek.

A w przyszłym roku? Dalej będziemy publikować Java exPress co 3 miesiące. Także wersja angielska zacznie się ukazywać regularnie. Podobnie z Java Geek i COOLuarami - planujemy w 2010 roku zorganizować 2 edycje. Jedną w Krakowie, a drugą... sam nie wiem gdzie. Może tym razem Wrocław, a może Warszawa? ;) Pojawią się także nowe inicjatywy, jak na przykład szkolenia dla każdego, czy screencasty. O ile starczy sił i ochotników do pomocy, współpracy i sponsoringu ;)

Na koniec jak zwykle apel. Jeśli chcesz napisać artykuł, pomóc w tłumaczeniach lub tworzeniu stron www, napisz do nas na kontakt@dworld.pl.

A zupełnie na koniec: Wszystkiego dobrego w 2010 roku. Niech Java będzie z Wami ;)

Do zobaczenia w 2010 roku,
Grzegorz Duda

ROZKŁAD JAZDY

NEW YEAR(2010).HAPPY = HAPPINESS.HIGH; 2
NA HORYZONCIE... 3
LOG4J - CZYLI JAK SKUTECZNIE TWORZYĆ LOGI W APLIKACJACH JAVOVYCH 4
GORM – GRAILS OBJECT RELATIONAL MAPPING 10
BEHAVIOUR-DRIVEN DEVELOPMENT Z EASYB 20
SIGNAL FRAMEWORK DLA PLATFORMY JAVA ME 26
GRADLE - MOCARNE NARZĘDZIE DO BUDOWY PROJEKTÓW 33
EXPRESS KILLERS, CZ. V 45
EXPRESS KILLERS, CZ. V - ODPOWIEDZI 46
RECENZJA: GROOVYMAG 11/2009 47
MISTRZ PROGRAMOWANIA: REFAKTORYZACJA, CZ. II 48

Redaktor: Grzegorz Duda
Korekta: Agnieszka Wadowska-Duda
Strony www i teksty html: Marek Podsiadły
Grafika: Jakub Sosiński

Na horyzoncie...

Grzegorz Duda

IntelliJ IDEA 9.0

Długo na to czekałem i w końcu jest. Wczoraj została wypuszczona 9 wersja znakomitego IDE IntelliJ IDEA. Co tu się długo rozpisywać. Ściągać i używać. Szczególnie, że oprócz płatnej wersji Ultimate, jest także darmowa dostępna wersja Communiy Edition. Porównanie funkcjonalności znajdziecie na stronie http://www.jetbrains.com/idea/features/index.html.

W wersji 9.0 zostało dodane wsparcie dla Androida, OSGi, Google ApEngine czy Java-EE 6. Wiele rzeczy zostało poprawionych. Zresztą zerknijcie sami (http://www.jetbrains.com/idea/whatsnew/index.html).

Konferencje w 2010

Rok 2010 zapowiada się bardzo ciekawie od strony konferencyjnej. Dużo i różnorodnie - tak można określić scenę konferencji javowych w Polsce.

Pierwszą konferencją w nowym roku będzie 4Developers (http://2010.4developers.org.pl/), która odbywać się będzie tym razem w Poznaniu, 26 marca. To co ją wyróżnia spośród innych konferencji, to 4 ścieżki: Java, .Net, zarządzanie projektami oraz PHP.

Kolejną konferencją będzie GeeCON, który w dniach 13-14 maja zawita do... Poznania. No właśnie - w tym roku Poznań rządzi konferencyjnie. A GeeCON nikomu nie trzeba chyba przedstawiać. Jedyna w Polsce 2 dniowa konferencja, organizowana przez społeczność JUGową.

W lecie przyjdzie nam odwiedzić Javarsovię (http://javarsovia.pl/). Dokładna data oraz formuła nie są jeszcze znane, ale można liczyć, że koniec czerwca lub początek lipca będzie się działo w Warszawie. To także konferencja organizowana przez społeczność. Tym razem z Warszawa JUG.

Na koniec roku zostaje już klasyczne JDD (http://jdd.org.pl/). Jak zwykle w Krakowie, jak zwykle w październiku, jak zwykle doskonale.

Jeśli do tego dodamy 2 edycje Nie-konferencji COOLuarów, może Warsjawę, może Java4poeple, kilka Eclipse DemoCampów oraz inncy mniejszych spotkań, to rok 2010 wygląda bardzo ciekawie. Nie wspominając o cyklicznych spotkaniach licznych w naszym kraju JUGów.

Konkurs Java Geek

Jeśli konferencje, to nie może zabraknąć konkursu Java Geek. Wystartowaliśmy na COOLuarach, a później JDD z konkursem na zbieranie pieczątek. Kolejne punkty do tego całorocznego konkursu można będzie zdobyć na 4Developers.

Koniec konkursu będzie we wrześniu 2010 roku, a więc jeszcze sporo czasu na zdobycie wielu punktów. A dla najbardziej aktywnych przewidziane są ciekawe nagrody Javowe. Szczegóły na http://www.dworld.pl/page/show/Java_Geek

Log4j - czyli jak skutecznie tworzyć logi w aplikacjach Javovych

Michał Szynkaruk

W życiu każdego programisty JAVA prędzej czy później pojawia się etap, w którym dochodzi on do wniosku, że korzystanie z „System.out.print" w przypadku chęci sprawdzenia wartości danych zmiennych, stanu w którym obecnie znajduje się aplikacja itp. nie jest ani efektywne, ani efektowne. Oczywiście można powiedzieć, że korzystanie ze standardowego wyjścia na konsole jest łatwe i z pozoru szybkie. No tak, ale co w przypadku jeśli chcemy, by zamiast na konsole wynik był zapisywany do pliku? Wówczas trzeba wszystkie te elementy zastąpić odpowiednio innym kodem. Świetnie, ale jeśli chcemy aby wyniki były zapisywane już nie tylko do pliku, ale ponownie na konsole? Uff ... no wymaga to wiele niepotrzebnej pracy. Czy można sobie ułatwić życie? Odpowiedź na to pytanie jest prosta: Log4j! Jest to wieloplatformowa biblioteka napisana w całości w Javie, której autorem jest Ceki Gülcü, a jej początki sięgają jeszcze ubiegłego wieku. Bez obawy, chodzi o lata 90-te :) Umożliwia ona zapisywanie logów w aplikacjach Javovych. Obecnie wspierana jest przez fundację Apache i wchodzi w skład projektu Jakarta. Wyróżnia się trzy wersje: 1.2 która jest uznawana za tę stabilną, 1.3 która została z pewnych względów porzucona oraz 2.0 która jest, nazwijmy to, eksperymentalna.

Co może być zapisywane w logach?

Przede wszystkim szeroko rozumiany przepływ aplikacji. Możemy zapisywać jakie są aktualne wartości poszczególnych zmiennych i obiektów, a także wypisywać komunikaty o różnego typu błędach bądź informować o zdarzeniach biznesowych mających miejsce w trakcie działania aplikacji.

A w jaki sposób?

To już zależy od naszej wyobraźni. Ale od początku. W log4j wyróżniamy trzy typy obiektów:

- Logger'y
- Appender'y
- Layout'y

Loggery posiadają metody, które tworzą logi i ustawiają im odpowiedni priorytet. Miejsca, do których mogą trafić logi są definiowane za pomocą Appender'ów. O tym jaką postać mają mieć komunikaty decydują obiekty typu Layout.
Wyróżniamy następujące Logger'y:

- NOPLogger
- RootCategory
- RootLogger

Najważniejszy jest RootLogger i to właśnie z niego będę korzystać w dalej omówionych przykładach. Pozostałe możemy pominąć. Zwłaszcza, że RootCategory uznawany jest za przestarzały.

Jeśli chodzi o ustawienie priorytetów, to wyróżniamy następujące poziomy:

- **FATAL**: Poważne błędy powodujące przedwczesne zakończenie działania aplikacji
- **ERROR**: Błędy wykonania
- **WARN**: Przy użyciu przestarzałych komponentów, w sytuacjach niespodziewanych nie wpływających na wadliwe działanie

> Z własnego doświadczenia wynika,
> iż najbardziej popularnymi są PatternLayout i HTMLLayout.

- **INFO**: W celu śledzenia wykonania
- **DEBUG**: Szczegółowe info. dotyczące przepływu w działaniu aplikacji, w celach diagnostycznych
- **TRACE**: Bardziej szczegółowe info
- **ALL/OFF**: Wszystkie poziomy / wyłączenie logów

Przy wyborze danego poziomu musimy wziąć pod uwagę to, że wszystkie logi wypisywane na poziomach niższych od tego który wybraliśmy nie będą wyświetlane. Przykładowo, jeśli poziom ustawimy na DEBUG (domyślny) to wówczas wszystkie logi wypisywane na poziomie TRACE nie ujrzą światła dziennego.

Jeśli chodzi o różnego rodzaju Appendery, to jest ich dużo zdefiniowanych, choć nikt nam nie zabrania by tworzyć własne w oparciu o klasę AppenderSkeleton. Wymienię najciekawsze z nich :

ConsoleAppender wypisuje logi na konsolę, domyślnie działa jak `System.out.`, ale może także jako System.Err.

WriterAppender zapisuje logi do pliku korzystając z klasy Writer bądź OutputStream.

JDBCAppender umożliwia zapis logów do bazy danych.

LF5Appender zapisuje logi do konsoli opartej na Swingu.

SMTPAppender pozwala wysłać logi na e-maila, głównie dotyczące poważnych błędów w działaniu aplikacji.

DailyRollingFileAppender rotuje logi do określonego rozmiaru, umożliwiając przy tym zapis z podziałem na lata, miesiące, dni itd.

Mając już wybrany Appender który wie gdzie zapisywać należy ustalić co będziemy zapisywać, a więc czas na Layouty :

HTMLLayout produkuje tabele HTML

PatternLayout umożliwia określenie szablonu wpisu

SimpleLayout powoduje, że logi są wpisywane w postaci: poziom – wiadomość

XMLLayout formuje wpis do formatu xml'owego

DateLayout ułatwia nam zarządzanie czasem umieszczanym we wpisie.

Z własnego doświadczenia oraz wszelkich rozmów użytkowników opisywanej biblioteki wynika, iż najbardziej popularnymi są PatternLayout i HTMLLayout.

Jeśli zdecydujemy się na wykorzystanie PatternLayoutu, to wówczas możemy określić w jaki sposób ma wyglądać dany wpis. Jak to zrobić ? Określamy szablon, w którym stosujemy poniższe elementy:

%% - pojedynczy znak %

%c - kategoria zdarzenia

%C - nazwa klasy, z której został wysłany wpis

%d - data zdarzenia

%m - treść komunikatu

%n - separator linii

%p - priorytet zdarzenia

%r - liczba milisekund, które upłynęły od momentu uruchomienia aplikacji

%t - nazwa wątku, z którego został wysłany wpis

> BasicConfiguratora używamy jeśli na szybko chcemy skonfigurować tworzenie log'ów.

%x - kontekst diagnostyczny powiązany z wątkiem

%M - nazwa metody, z której został wysłany wpis

%L - numer linii, z której pochodzi zdarzenie

%F - nazwa pliku, z którego pochodzi zdarzenie

%l – lokalizacja, z której został wysłany wpis

Zaleca się ograniczenie czterech ostatnich elementów ze względu na spowalnianie aplikacji.

Pozostała nam jeszcze drobna rzecz jaką jest konfiguracja. Możemy tego dokonać na trzy sposoby :

1. przy użyciu obiektu typu BasicConfigurator
2. przy zastosowaniu pliku właściwości
3. przy wykorzystaniu pliku XML

BasicConfiguratora używamy jeśli na szybko chcemy skonfigurować tworzenie log'ów.

Jest to najmniej skomplikowany sposób konfiguracji. Powoduje utworzenie obiektu PatternLayout, którego ConversionPattern ma wartość:

%-4r [%t] %-5p %c %x - %m%n

Klasa BasicConfigurator posiada tylko trzy metody : configure() bezargumentowy i z jednym argumentem (Appender) oraz resetConfiguration().

Konfiguracja za pomocą pliku właściwości jest podobna do konfiguracji przy wykorzystaniu pliku XML, lecz składnia jest inna. Oto przykładowy kod wywołany ze statycznej metody main:

```
BasicConfigurator.configure();
Logger logger = Logger.
  getRootLogger();
logger.debug(„Hello world”);
```

Wynikiem jest:

```
2 [main] DEBUG root  - Hello world
```

Jest to zrozumiałe gdyż korzystamy z domyślnego szablonu o którym nie tak dawno wspomniałem.

Jeśli chcemy zdefiniować własny szablon, to możemy to zrobić przykładowo:

```
Layout lay1 =
  new PatternLayout(„[%p] %c - %m
    - Data wpisu: %d %n”);
Appender app1 =
  new ConsoleAppender(lay1);
BasicConfigurator.configure(app1);
Logger logger =
  Logger.getRootLogger();
logger.debug(„Hello world”);
```

Wynik jaki otrzymałem:

```
[DEBUG] root - Hello world - Data
wpisu: 2009-11-24 15:20:29,049
```

Dlaczego tak się dzieje ? To proste: dekla-

> Decyzja z jakich metod będziecie korzystać zależy od was.

rujemy szablon o danej postaci oraz appender który ma za zadanie wypisać logi na konsole. Następnie umieszczamy appender jako argument funkcji *configure* i to wszystko.

A co jeśli chcemy zmienić poziom wypisywania logów ?

Nie ma problemu, dodajemy na końcu przykładowo:

```
logger.setLevel(Level.TRACE);
```

A jeśli chcemy, aby logi były wypisywane do pliku ?

Nie ma problemu, linijkę `Appender app1 = new ConsoleAppender(lay1);` możemy przykładowo zastąpić w poniższy sposób:

```
Appender app1 = null;
try {
  app1 = new FileAppender(
    lay1,"C:/log_express.txt");
} catch(IOException ex) {
...
}
```

No dobrze, wspomniałem o tym, że można dokonać konfiguracji także przy użyciu pliku properties oraz w formacie xml. Zasada działania jest podobna: umieszczamy dany plik o nazwie log4j.properties lub log4j.xml w dowolnym pakiecie (jeśli będzie to domyślny pakiet, wówczas nie będziemy musieli bawić się w określanie gdzie system ma szukać pliku konfiguracji, gdyż odbędzie się to automatycznie).

Przykładowa zawartość pliku właściwości :

```
log4j.appender.C=org.apache.
log4j.ConsoleAppender
log4j.appender.C.layout=
  org.apache.log4j.PatternLayout
log4j.appender.C.layout.
  ConversionPattern=[%p] %c - %m -
  Data wpisu: %d %n
log4j.rootLogger=DEBUG, C
```

Przykładowa zawartość pliku xml:

```
<?xml version="1.0"
      encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration
  SYSTEM „log4j.dtd">
<log4j:configuration xmlns:log4j=
"http://jakarta.apache.org/
log4j/">
  <appender name="console"
    class="org.apache.log4j.Conso-
leAppender">
    <param name="Target"
           value="System.out"/>
     <layout class="org.apache.
log4j.PatternLayout">
      <param name="ConversionPat-
tern" value="[%p] %c - %m - Data
wpisu: %d %n"/>
    </layout>
  </appender>
  <root>
    <priority value ="debug" />
   <appender-ref ref="console" />
  </root>
</log4j:configuration>
```

Konfiguracja w obu przypadkach powoduje takie samo działanie. Różnica jest tylko w sposobie zapisu, a decyzja z jakich metod będziecie korzystać zależy od was.

Warto wspomnieć, że jeśli nie dokonamy konfiguracji Log4j pojawi się stosowne ostrzeżenie, a logi nie będą wpisywane.

Jak skorzystać z Log4j w Netbeans?

Oto przykładowy sposób:

- pobieramy archiwum ZIP ze strony: http://apache.privatejetscharter.net/logging/log4j/1.2.15/apache-log4j-1.2.15.zip
- wyodrębniamy plik *log4j-1.2.15.jar*

> Tworzenie logów w inteligentny sposób znacząco wspomaga tworzenie oprogramowania.

- tworzymy nową bibliotekę przez wybór zakładek Tools → Library
- dodajemy bibliotekę do projektu
- korzystamy z odpowiednich „importów”, np. import org.apache.log4j.*;

Log4j versus Java Logging API

Przy pisaniu wielu programów z wykorzystaniem Log4j spotkałem się z opiniami w stylu:

„Przecież istnieje już coś takiego jak Java Logging API, która jest wspierana przez SUN Microsystems.”

Zgadza się. Nie oznacza to jednak, że jeśli coś jest standardem to znaczy, że jest lepsze, wręcz przeciwnie. Java Logging API powstała później niż Log4j i w dużej mierze się na niej opiera.

Choć różnic jest niewiele to wśród programistów Javy otarła się opinia, że to co możemy wykonać za pomocą Java Logging API jest także możliwe przy wykorzystaniu Log4j, a nawet więcej.

Plusy i minusy obu bibliotek zebrałem w poniższej tabelce.

Log4j	Java Logging API
+	+
Wspierany przez Apache	Wspierane przez SUN
Dłużej na rynku	Nie trzeba dodawać nowych bibliotek
Używany przez ogromną społeczność	-
-	Wybierane przez mniejszą liczbę programistów
Rzadko aktualizowany	

Poza wyraźnymi plusami wspólną wadą może być to, że przy bardzo dużych logach można mieć problemy z dostrzeżeniem tych najistotniejszych dla nas informacji. Na świecie to zjawisko jest określane mianem scrolling-blindness.

Podsumowanie

Możliwości jakie posiada biblioteka Log4j są ogromne. Jest ona łatwa w użyciu i co ważne – na otwartym kodzie. Była optymalizowana pod względem szybkości działania. Nie ma także obaw o to jak będzie funkcjonować w aplikacjach wielowątkowych. Tworzenie logów w inteligentny sposób znacząco wspomaga tworzenie oprogramowania. W następnym numerze umieszczę kilka przykładowych kodów źródłowych pokazujących jak efektownie i efektywnie korzystać z Log4j w różnego typu aplikacjach. Pamiętajcie: nigdy więcej System.out.print :)

O autorze

Michał jest studentem IV roku informatyki na Wydziale Informatyki i Zarządzania Politechniki Wrocławskiej. Jest certyfikowanym programistą Javy (SCJP 6.0).

VRILTEK.COM

Logotypy

Portale

Plakaty

Wizytówki

Strony WWW

DTP

Sklepy internetowe

Ulotki

Banery

GORM – Grails Object Relational Mapping

Mateusz Mrozewski

W tym artykule chciałbym przedstawić podstawowe informacje na temat GORM (Grails Object Relational Mapping) oraz kilka bardziej zaawansowanych zagadnień z nim związanych. Po przeczytaniu artykułu dowiesz się jak działa warstwa persystencji w Grails.

Rozpoczynamy

GORM, warstwa persystencji w Grails zrealizowana jest jako mapowanie obiektowo-relacyjne. Pod maską GORM znajdziemy Hibernate. Możemy z niego również skorzystać bezpośrednio.

Mapowanie OR jest zrealizowane w Grails na tzw. klasach domenowych (Domain class). Klasa domenowa zwana jest w innych kręgach encją. Klasa domenowa utworzona (z linii poleceń lub wyklikana w NetBeans) w najprostszym przypadku jest po prostu pustą klasą, wewnątrz której definiujemy atrybuty, które chcemy przechowywać, np.

```
class Person {
  String firstname
  String lastname
}
```

Gdy utworzymy taką klasę domenową i wystartujemy nasz projekt (z domyślnymi ustawieniami) w NetBeans, zostanie uruchomiona baza Hypersonic i zostanie utworzona nowa tabelka Person. Będzie posiadała 4 kolumny: firstname, lastname, id i version. Grails domyślnie zajmuje się za nas generowaniem identyfikatorów (w najbardziej odpowiedni dla danej bazy sposób) oraz wersjonowaniem.
Podstawowe operacje (CRUD) wykonujemy w następujący sposób:

```
//Utworzenie nowego rekordu CREATE
def p1 = new Person(
  firstname:”Mateusz”,
  lastname:”Mrozewski”);
p1.save()

// Odczytanie rekordu READ
// tu korzystamy z automatycznie
// wygenerowanego id
def p2 = Person.get(1)

// Aktualizacja UPDATE
// modyfikujemy rekord
// odczytany wcześniej
p2.firstname = „Krzysztof”
p2.save();

// Usuwanie DELETE
// usuwamy odczytany wcześniej
// rekord
p2.delete()
```

Jak widać wykonanie podstawowych operacji jest niezwykle łatwe.

Asocjacje

Z GORM robi się zawsze ciekawiej gdy dochodzi do asocjacji. Jednak jest to również rozwiązane w sposób czysty i prosty. Przykłady różnych relacji poniżej:

```
// One-to-one unidirectional -
// jeden do jednego,
// jednokierunkowa
// mając referencję do samochodu,
// możemy pobrać jego silnik,
// ale nie odwrotnie

Class Car {
    Engine engine
}

Class Engine {}

// One-to-one bidirectional -
// jeden do  jednego,
// dwukierunkowa
// Mając referencję do samochodu,
// możemy pobrać jego silnik,
// odwrotnie również
Class Car {
  Engine engine
}
```

> Klasa domenowa zwana jest w innych kręgach encją.

```
Class Engine {
  Car car
}
```

W przypadku obu relacji powyżej operacje nie są kaskadowane. To oznacza, że jeśli skasujemy samochód, to jego silnik pozostanie (usuniemy z bazy, nie mówię o wypadkach). Podobnie ma się zapis zmian, utworzenie nowego rekordu, aktualizacja. Musimy to zrobić ręcznie dla obu klas. Wygodniej było by mieć kaskadowanie. Przykład z kaskadowaniem operacji wygląda tak:

```
// One-to-one unidirectional - jeden do jednego, jednokierunkowa
// mając referencję do samochodu, możemy pobrać jego silnik,
// ale nie odwrotnie
Class Car {
  Engine engine
}

Class Engine {
  static belongsTo = Car
}

// One-to-one bidirectional - jeden do  jednego, dwukierunkowa
// Mając referencję do samochodu, możemy pobrać jego silnik,
// odwrotnie również
Class Car {
  Engine engine
}

Class Engine {
  static belongsTo = [car:Car]
}
```

Jak widać kaskadowanie realizujemy poprzez właściwość belongsTo. Warto zwrócić uwagę na różnicę w deklaracji tej właściwości przy relacji jedno i dwukierunkowej.

Od wersji Grails 1.2-M4 mamy jeszcze jedną możliwość deklarowania relacji jeden do jednego:

```
class Car {
   static hasOne = [engine:Engine]
}
class Engine {
  Car car
}
```

Różnica względem poprzednich przykładów jest taka, że klucz obcy zostanie umieszczony w tabeli engine, a nie w tabeli car.

Relację o krotności większej od jeden tworzymy poprzez wykorzystanie właściwości hasMany. Stosując ją po obu stronach relacji otrzymamy relację wiele do wielu, a tylko po jednej stronie relację jeden do wielu. Tak jak w przypadku relacji jeden do jednego, aby wymusić kaskadowanie należy zastosować właściwość belongsTo. Na przykładzie:

```
//  One-to-many  unidirectional  -
jeden do wielu, jednokierunkowa
class Car {
  static hasMany = [wheels:Wheel]
}

class Wheel {}
```

W związku z relacją jeden do wielu należy pamiętać o kilku ważnych sprawach:

- Grails do zmapowania tej relacji w bazie wykorzysta dodatkową tabelę złączeniową. Można to zachowanie nadpisać i wykorzystać klucze obce.
- Grails pozwala na wykorzystanie dwóch strategii pobierania: lazy i eager. Domyślnie stosowane jest lazy, co w przypadku relacji o krotności większej niż jeden może prowadzić do problemów wydajnościowych (każdy kolejny rekord należąc do relacji jest pobierany oddzielnym zapytaniem w miarę jak

> Grails domyślnie stosuje kolekcję Set
> w przypadku relacji o krotności większej niż jeden.

iterujemy po naszej kolekcji).

- Domyślnie kaskadowane są operacje INSERT i UPDATE, ale do kaskadowania operacji DELETE musimy zastosować właściwość belongsTo.

W przypadku relacji wiele do wielu stosujemy właściwość hasMany po obu stronach relacji:

```
class Book {
 static hasMany = [authors:Author]
}

class Author {
  static hasMany = [books:Book]
}
```

Również w tym przypadku możemy zastosować właściwość belongsTo. Dokumentacja wspomina, że scaffolding nie wspiera jeszcze tego typu relacji.

Grails domyślnie stosuje kolekcję Set w przypadku relacji o krotności większej niż jeden. Domyślne zachowanie możemy nadpisać stosując taką kolekcję, jaka nam się podoba, z więc dowolną implementację Set, List lub Map.

Strategie dziedziczenia

Grails wspiera dwie strategie dziedziczenia: table-per-hierarchy oraz table-per-subclass. Taka jest przynajmniej informacja w oficjalnej dokumentacji. Jak wiadomo pod maską Grails znajdziemy Hibernate'a, a ten wspiera również strategię table-per-concrete-class, więc pewnie i w Grails dałoby się to jakoś wykorzystać.

Czym tak naprawdę są strategie dziedziczenia? Nazwa brzmi dumnie, ale sprawa nie jest zbyt złożona. Wyobraźmy sobie sytuację, w której mamy klasę domenową dziedziczącą po innej klasie domenowej, np.:

```
class Osoba {
  String imie
  String nazwisko
}

class Pracownik extends Osoba {
  String nip
}
```

Strategia dziedziczenia określa nam, jak taka struktura zostanie przedstawiona w bazie danych. Dopóki nie wprowadziliśmy dziedziczenia, każda klasa domenowa miała swoją tabelkę. W tym przypadku, jeśli zastosujemy strategię table-per-hierarchy (domyślne) wszystkie klasy z hierarchii będą umieszczone w tej samej tabeli. Tabelka będzie wyglądała następująco:

```
CREATE TABLE `osoba` (
   `id` bigint(20) NOT NULL auto_
increment,
   `version` bigint(20) NOT NULL,
   `imie` varchar(255) NOT NULL,
  `nazwisko` varchar(255) NOT NULL,
   `class` varchar(255) NOT NULL,
   `nip` varchar(255) NULL,    PRI-
MARY KEY  (`id`)
);
```

Jak widać tabela posiada kolumny „imie" i „nazwisko" zadeklarowane w klasie Osoba oraz pole „nip" z klasy Pracownik. Dodatkowo zawiera automatycznie utworzony przez Grails identyfikator (w moim przypadku auto_increment bo korzystam z MySQLa), kolumnę „version" służącą do wersjonowania rekordów oraz kolumną „class", która pozwala nam określić czy dany wiersz reprezentuje osobę czy pracownika. Plusem tego rozwiązania jest posiadanie w jednej tabeli wszystkiego co potrzeba (kwestia optymalizacyjna - nie potrzeba żadnych operacji złączenia). Minusem jest to, że nasza aplikacja pozwoli teraz zatrudnić pracownika „na czarno" -

> Każda klasa domenowa
> ma automatycznie wygenerowany atrybut „version”,
> który przy każdej operacji UPDATE jest inkrementowany.

kolumna „nip” musi być null (gdyż możemy chcieć dodać osobę, która nie jest pracownikiem i nie ma NIPu).

Drugą dostępną strategią jest table-per-subclass. W tym wypadku każda podklasa będzie posiadała dodatkową tabelkę, która będzie przechowywać tylko dodatkowe pola. Pola odziedziczone będą przechowywane w tabelce klasy nadrzędnej. Jak to zrobić? Wystarczy dodać mały kawałek kodu w klasie nadrzędnej:

```
class Osoba {
  String imie
  String nazwisko

  static mapping = {
    tablePerHierarchy false
  }
}

class Pracownik extends Osoba {
  String nip
}
```

W tym przypadku zostanie utworzona tabela osoba, zawierająca pola „id”, „version”, „imie” i „nazwisko”. Druga tabelka to pracownik, która będzie zawierała pola „id” (pokrywające się z tabelą osoba) oraz pole „nip”. Minusem tego rozwiązania jest potrzeba wykonania złączenia zawsze, gdy będziemy chcieli pobrać rekord pracownika. W zamian za to możemy wymusić „nip” na poziomie bazy danych.

Zagnieżdżone klasy domenowe

Dokumentacja na ten temat nie jest zbytnio rozpisana, ale w sumie nie wiem, czy potrzeba tu dużo tłumaczenia. Chodzi o to, że możemy zagnieździć klasę domenową w innej klasie domenowej. Z punktu widzenia kodu nadal deklarujemy dwie klasy, oznaczamy jedynie, że pewna właściwość jest klasą zagnieżdżoną, np.:

```
class Car {
  Engine engine
  static embedded = [,engine’]
}

class Engine {
  String type
}
```

Z takiej deklaracji otrzymamy jedną tabelę „car”, która będzie posiadała pola „id”, „version” oraz „engine_type”. Pozwala nam to na ładne rozdzielenie modelu obiektowego przy jednoczesnym uproszczeniu zapytań. Zamiast dwóch zapytań lub jednego ze złączeniem mamy po prostu jedno. Należy pamiętać, że obie klasy należy zadeklarować w pliku Car.groovy. Jeśli przeniesiemy klasę Engine do oddzielnego pliku do zostanie wygenerowana oddzielna tabelka dla tej klasy.

Opimistic i Pessimistic Locking

Blokowanie to sposób na zabezpieczenie się przez współbieżnymi zmianami. Zasadniczo są dwa podejścia: optymistyczne, które zakłada, że współbieżne zmiany raczej nie nastąpią (sprawdzenie wykonane jest na koniec operacji/transakcji) oraz pesymistyczne, które zakłada, że współbieżne zmiany są na tyle częste, że lepiej wywłaszczyć rekord przed wykonaniem jakichkolwiek zmian. Są to powszechnie stosowane podejścia, nie tylko w Grails (i nie tylko we framework’ach web’owych).

Blokowanie optymistyczne wykorzystuje wersję rekordu (robi to za nas Grails, a właściwie to Hibernate). Każda klasa domenowa ma automatycznie wygenerowany atrybut „version”, który przy każdej operacji UPDATE jest inkrementowany. Jeśli przy zapisie rekordu GORM wykryje, że wersja została w między czasie zmieniona przez inną transakcję, nasza transakcja zostanie

> Blokownie pesymistyczne działa natomiast wykorzystując operację „SELECT ... FOR UPDATE”.

wycofana i zostanie wyrzucony wyjątek StaleObjectException. Jak to obsłużymy zależy od nas: możemy napisać zmiany lub zrezygnować z zapisu i poprosić użytkownika o ponowne wykonanie zmian na aktualnych danych.

Blokownie pesymistyczne działa natomiast wykorzystując operację „SELECT ... FOR UPDATE”. Wykonanie takiej operacji na bazie danych powoduje zablokowanie rekordu dla innych transakcji (na poziomie bazy danych). Inne transakcje, które spróbują pobrać ten sam rekord zatrzymają się i będą czekać na jego odblokowanie. Należy pamiętać, że jest to mechanizm zależny od danej bazy danych i tak jak wspomina dokumentacja, dostarczony z Grails HSQLDB tego nie obsługuje. Jak wygląda to w kodzie:

```
def car = Car.get(1)
// w tym miejscu inny wątek może pobrać ten sam rekord i blokowanie nic nam nie da
car.lock() // tu blokujemy rekord
car.model = „Audi”
car.save()

// inna wersja rozwiązująca powyższy problem
def car = Car.lock(1)
car.model = „Audi”
car.save()
```

Blokować możemy też od razu w wykonaniu zapytań. Blokady zwalniane są w momencie zatwierdzenia lub cofnięcia transakcji.

Pobieranie danych

GORM pozwala nam na wykonywanie zapytań w kilka różnych sposobów. Możemy do nich zaliczyć podstawowe metody operujące na klasach domenowych, dynamic finders (metody typu find*), kryteria (które pozwalają budować zapytania podobnie jak robi to JaQu) albo wykorzystać stary (dobry) HQL.

Metody klas domenowych

Kawałek kodu chyba najlepiej pokaże, jakie metody możemy wywołać:

```
// Wylistuj wszystkie osoby
def persons = Person.list()
// Pobierz osobę o id=3
def person = Person.get(3)
// Pobierz osoby o id 3,4,5
def persons = Person.getAll(3, 4, 5)
// Pobierz 10 osób z offsetem 100, sortując po nazwisku malejąco
def persons = Person.list(max:10, offset:100, sort:”lastname”, order:”desc”)
```

Dla metody list() możemy dodatkowo określić ilość zwróconych rekordów, od którego rekordu rozpocząć, sortowanie łączenie z kierunkiem, a także w jaki sposób pobrać asocjacje (eager/lazy). Metoda get() po prostu pobiera rekord o zadanym id, a getAll() rekordy o zadanych id. Jeśli któryś z nich nie będzie istniał, lista wyników będzie zawierała null.

Dynamic finders

Dynamic finders to metody tworzone dynamicznie w środowisku uruchomieniowym na podstawie właściwości klasy:

```
class Person {
  String firstname
  String lastname
  Date dateOfBirth
}

def p1 = Person.findByFirstname(„Mateusz”)
def p2 = Person.findByLastname(„Mrozewski”)
def p3 = Person.findByFirstnameAn-
```

> Ale chwileczkę, skąd te wszystkie metody się wzięły?

```
dLastnameLike(„Mateusz”, „M%”)
def p4 = Person.findByDateOfBirthI-
sNull()
```

Wypisane metody to tylko kilka możliwych przykładów. Możemy takie metody wywoływać dla każdej właściwości. Tak jak w przykładzie wyszukania osoby p3 możemy połączyć warunki dla dwóch dowolnych właściwości z operatorem AND lub OR (maksimum dwóch - dla większej ilości powinniśmy wykorzystać kryteria lub HQL). Dwa typy metod to findBy*, który zwraca pierwszy rekord wyniku oraz findAllBy*, który zwraca wszystkie pasujące wyniki. A dozwolone operacje to:

- InList - czy wartość znajduje się na podanej liście
- LessThan - czy wartość jest mniejsza niż podana
- LessThanEquals - czy wartość jest mniejsza lub równa niż podana
- GreaterThan - czy wartość jest większa niż podana
- GreaterThanEquals - czy wartość jest większa lub równa niż podana
- Like - podobne do SQLowego LIKE
- ILike - jak wyżej, tylko case insensitive (to nie produkt apple'a)
- NotEqual - nie równe (nie ma samego equal, bo to równoważne z np. findByName)
- Between - czy wartość jest pomiędzy zadanymi
- IsNotNull - czy wartość nie jest null'em
- IsNull - czy wartość jest null'em

Ten typ zapytań możemy również wykorzystać dla asocjacji. Do tych zapytań możemy też dodać parametry takie jak w metodzie list(). Prawda, że fajne? Osobiście uważam, że przy zastosowaniu już dwóch parametrów przestaje być to czytelne i dużo łatwiej pisze się i analizuje kryteria, ale do prostych zapytań jest to jak najbardziej ciekawe rozwiązanie.

Ale chwileczkę, skąd te wszystkie metody się wzięły? Przecież nic nie dziedziczyliśmy, nic nie implementowaliśmy. Tutaj z pomocą przychodzi nam dynamiczność Groovy oraz to, jak zostało to wykorzystane w Grails. Konkretnie chodzi tu o wykorzystanie metaklas i ich właściwości methodMissing. W książce The Definitive Guide to Grails w dodatku poświęconym Groovy możemy znaleźć nawet prosty przykład implementacji dynamic finder.

Kryteria

Kryteria to sposób na budowanie złożonych zapytań poprzez wykorzystanie składni Groovy. A konkretnie wykorzystana jest tu koncepcja builders.

Nas interesuje jednak jak to wygląda w kodzie. Oto mały przykład z dokumentacji:

```
def c = Account.createCriteria()
def results = c {
  like(„holderFirstName”, „Fred%”)
  and {
    between(„balance”, 500, 1000)
    eq(„branch”, „London”)
  }
  maxResults(10)
  order(„holderLastName”, „desc”)
}
```

W tym przykładzie zostały utworzone kryteria dla klasy domenowej Account. Następnie rozbudowaliśmy je o kolejne warunki - ten prosty przykład chyba dość dobrze pokazuje jakie są możliwości kry-

> W GORM możemy też wykorzystać HQL

teriów.

Co można w takich kryteriach zrobić:

- wykorzystać wszystkie metody z klasy Hibernate Restrictions
- wykorzystać operatory logiczne AND, OR i NOT
- pytać o inne klasy domenowe będące w asocjacji z główną klasą
- wykorzystać wszystkie metody z klasy Hibernate Projections
- wykorzystać wszystkie metody z klasy Hibernate ScrollableResults
- określić ilość pobranych rekordów oraz od którego rekordu zacząć
- określić czy asocjacje będą pobierane w trybie EAGER czy LAZY

Nie wypisuję wszystkich możliwości, gdyż jest tego mnóstwo, a wszystko jest w Javadoc'ach. Jednak ta krótka lista przynajmniej pokazuje jak szerokie mamy możliwości. GORM to nie tylko CRUD.

HQL

W GORM możemy też wykorzystać HQLa (Hibernate Query Language). Jest to język zapytań bardzo podobny do SQLa, ale nie operujemy w nim na tabelkach, tylko na encjach (klasach domenowych). Mamy do dyspozycji trzy metody, które możemy wywołać na naszej klasie domenowej:

- find - zwraca pierwszy wynik pasujący do zapytania
- findAll - zwraca wszystkie wyniki pasujące do zapytania
- executeQuery - wykonuje zapytanie, niekoniecznie zwracające wyniki (np. UPDATE)

Kilka przykładów:

```
// Znajdź pierwszą osobę
// o imieniu Mateusz
def person = Person.find(
  „from Person as p where
   p.firstname = ?”, [‚Mateusz’])
// Znajdź wszystkie osoby
// o imieniu Mateusz
// i nazwisku na M
def results = Person.findAll(
  „from Person as p where
   p.firstname = :firstname and
   p.lastname like :lastname”,
  [firstname:’Mateusz’,
   lastname:’M%’])
```

Niezbędnikiem jest zapoznanie się z rozdziałem dokumentacji Hibernate poświęconej HQL.

Zdarzenia i znaczniki czasowe

Bardzo ciekawą funkcją w GORM są zdarzenia. Zdarzenia to domknięcia wywoływane w odpowiednim momencie cyklu życia instancji klasy domenowej. Możemy je użyć np. do zapisania daty ostatniej aktualizacji, utworzenia encji, zapisania do logu śladu po usuwanej encji, itp. Wyróżniamy następujące typy zdarzeń:

- beforeInsert - wywoływane zanim obiekt zostanie zapisany do bazy po raz pierwszy
- beforeUpdate - wywoływane zanim obiekt zostanie zaktualizowany w bazie
- beforeDelete - wywoływane przed usunięciem rekordu
- afterInsert - wywoływane po zapisaniu obiektu do bazy po raz pierwszy
- afterUpdate - wywoływane po zaktualizowaniu obiektu w bazie

> Własne mapowanie pozwala nam samodzielnie określić jakie będą nazwy tabel i kolumn.

- afterDelete - wywoływane po usunięciu obiektu z bazy
- onLoad - wywoływane po wczytaniu obiektu z bazy

I w przykładzie:

```
class Person {
  Date dateCreated
  Date lastUpdated
  Date loadTime

  def beforeInsert = {
    dateCreated = Date()
  }
  def beforeUpdate = {
    lastUpdated =new Date()
  }
  def onLoad = {
    loadTime = new Date()
  }
}
```

W tym przykładzie naszą klasę domenową rozbudowaliśmy o zapisywanie do bazy daty aktualizacji, daty utworzenia oraz ustawianie czasu wczytania rekordu. Jeśli chodzi o datę utworzenia i aktualizacji, to GORM pozwala nam na małe uproszczenie: wystarczy zadeklarować w klasie domenowej właściwość lastUpdated i dateCreated a będą one automatycznie ustawiane na odpowiednie wartości (konwencja nie tylko ponad konfigurację, ale również ponad kodowanie).

Własne mapowanie

Kolejną cechą GORMa jest własne mapowanie tabel. Własne mapowanie pozwala nam samodzielnie określić jakie będą nazwy tabel i kolumn. Jest to niezwykle przydatne, gdy musimy dostosować się do konwencji przyjętej w firmie/projekcie lub gdy piszemy kod do istniejącej bazy danych. Więcej na ten temat można poczytać oczywiście w dokumentacji, wydaje mi się że ten dział wymaga najmniej dodatkowych opisów i dyskusji.

Wspomnę tylko jeszcze, że możemy również wpłynąć na to, jak generowane będą identyfikatory (domyślnie GORM wykorzystuje ten najodpowiedniejszy dla danej bazy danych). Możemy też wpłynąć na indeksy, jakie zostaną wygenerowane.

Dla zobrazowania możliwości mały przykład:

```
class Person {
  String firstName
  static mapping = {
    table ,people'
    firstName column:'First_Name'
  }
}
```

Cache

Pod maską GORM siedzi Hibernate, nie mogło więc zabraknąć mechanizmu second-level cache z tej biblioteki. Cache możemy włączyć w pliku konfiguracyjnym DataSource.groovy (domyślnie przy wygenerowaniu projektu jest on włączony). Możemy podobnie jak w Hibernate, w pamięci podręcznej (mało zręczne tłumaczenie słowa cache) zapisać encje lub wyniki zapytań. Czy dana klasa domenowa ma być zapisywana do pamięci podręcznej określa się w samej klasie domenowej:

```
class Person {

  static mapping = {
    cache true
  }
}
```

„Keszować" możemy też asocjacje. Możemy ustawić jeden z kilku dostępnych rodzajów pamięci podręcznej. Wpływa to na wydajność naszej pamięci, a to wynika ze strategii obsługi danych oraz dopuszczalnego ryzyka współbieżnych modyfikacji:

> Warstwa persystencji jest niezwykle rozbudowana. Jest dość elastyczna i bogata w funkcje.

- read-only - nasza aplikacja tylko odczytuje dane i nigdy ich nie modyfikuje. Szybkie w działaniu, gdyż nie potrzeba nam żadnych synchornizacji.
- read-write - nasza aplikacja zapisuje i odczytuje dane. Dostęp jest synchronizowany.
- nonstrict-read-write - nasza aplikacja i czyta i pisze, ale prawdopodobieństwo wystąpienia jednoczesnego zapisu jest znikome.
- transactional - przy tej strategi możemy wykorzystać w pełni transakcyjną pamięć podręczną, taką jak JBoss TreeCache.

Ograniczenia (contraints)

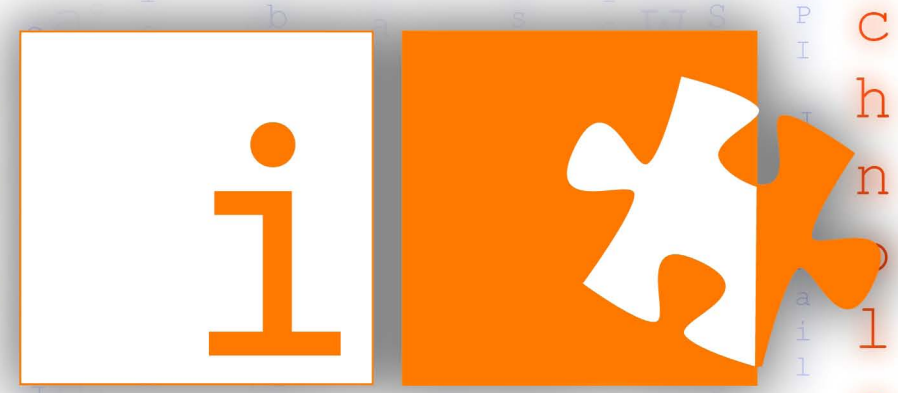
Grails posiada wbudowany mechanizm walidacji danych. Pozwala on nam na bardzo łatwe i deklaratywne określanie, jakie właściwości mogą przyjąć jakie wartości. Dlaczego jest to ważne z punktu widzenia GORM? Ponieważ może to wpłynąć na to, jak zostanie wygenerowany schemat bazy danych. Bardzo mi się to spodobało, że dostajemy z automatu zabezpieczenie na poziomie bazy danych również, a nie tylko na poziomie aplikacji.

Do deklaracji ograniczeń w klasie domenowej wykorzystywana jest sekcja constraints:

```
class Task {
  String name
  String description

  static constraints = {
    description(maxSize:1000)\
  }
}
```

Podsumowanie

Jak widać warstwa persystencji jest niezwykle rozbudowana w Grails. Jest dość elastyczna i bogata w funkcje. Jeśli komuś GORM nie do końca odpowiada, to zawsze można skorzystać z Hibernate bezpośrednio, wykorzystać plugin do JPA lub bezpośrednio łączyć się z bazą danych (JDBC albo pakiet GroovySQL). Nie mniej jednak myślę, że GORM zadowoli większość użytkowników.



Java Team

dołącz do nas www.isolution.pl

Notatki o testowaniu: Behaviour-driven development z easyb

Bartosz Majsak

Groovy to język, który przykuwa coraz większą uwagę entuzjastów technologii Javowych, a artykuły pojawiające się na łamach Java exPress tylko to potwierdzają. Jego niewątpliwymi zaletami są prostota i przejrzystość składni, które pozwalają zdecydowanie zwiększyć produktywność i uwolnić od monotonnych linijek kodu tak dobrze znanych nam z Javy.

W niniejszym artykule chciałbym przedstawić bardzo ciekawe i użyteczne narzędzie bazujące na Groovym. Zachwyca ono swoją prostotą i wszechstronnym zastosowaniem w testach, oraz, co najistotniejsze, pozwala na pisanie testów w myśl koncepcji behaviour-driven development.

Kilka(dziesiąt) słów wprowadzenia

Zanim jednak przejdziemy do przedstawienia *easyb*, warto zacząć od krótkiego wprowadzenia do coraz bardziej popularnej techniki *behaviour-driven development*. Na jakich założeniach się opiera? Czym różni się od *test-driven development*? Jakie narzędzia w Javie pozwalają pisać testy w oparciu właśnie o nią? Po przeczytaniu tego artykułu, poza wzbogaceniem słownika informatycznych żargonów o kolejny modny akronim, z pewnością będziesz znał odpowiedzi na te pytania.

Do zilustrowania koncepcji BDD posłuży nam prosty przykład. Załóżmy, że organizujemy konferencję o technologiach związanych z Javą i potrzebujemy stworzyć moduł rejestracji. Oczywiście, aby zachęcić potencjalnych uczestników powinniśmy zaoferować jakieś zniżki. Dla maksymalnego uproszczenia naszego przykładu umówmy się także, że cena wejściówki to 100,00 PLN. A zatem, do dzieła! Zaczynamy (jak zawsze zresztą, prawda?), od testu:

```
@Test
public void testConferenceDiscount() {
  Participant p =
    new Participant();
  p.setJavaUserGroupMember(true);
  RegistrationService
    registrationService =
    new RegistrationService();
  PaymentDetails paymentDetails =
    registrationService.register(
      participant);
  assertEquals(80.0,
    paymentDetails.getPrice());
}
```

Nie trzeba mieć wielkiego doświadczenia w pisaniu testów, aby stwierdzić, że powyższy kod jest daleki od ideału. Przede wszystkim jest nieprecyzyjny. Nazwa metody tak naprawdę nie niesie ze sobą żadnej istotnej informacji. W momencie, gdy w module rejestracji pojawi się jakiś błąd, będziemy zmuszeni do wnikliwej analizy kodu takiej metody testowej, aby określić wymaganie definiujące warunki przydzielania zniżki. W powyższym wypadku nie stanowi to wielkiego wyzwania, ale zapewne wielu z Was musiało nie raz w swojej w karierze analizować bardziej „rozbudowane" testy. Co więcej, pisząc testy, zwłaszcza na początku przygody z test-driven development, bardzo często powraca pytanie *„co ja tak naprawdę powinienem przetestować?"*. Nierzadko zdarza się nam skupiać bardziej na implementacyjnych szczegółach klas, których działanie chcemy weryfikować, niż na faktycznych wymaganiach, które mają spełniać. Między innymi z takimi właśnie problemami próbuje się mierzyć koncepcja *behaviour-driven development*.

Behaviour-driven development, sformułowane po raz pierwszy przez Dana Northa w magazynie *„Better Software"* z marca 2006, to podejście skupiające się przede

> Behaviour-driven development,
> to podejście skupiające się przede wszystkim na wymaganiach
> dotyczących testowanego komponentu aplikacji.

wszystkim na wymaganiach dotyczących testowanego komponentu aplikacji. Jest tak naprawdę delikatną ewolucją *test-driven development*. Zasadniczą różnicą jest położenie nacisku na weryfikację konkretnych wymagań stawianych aplikacji. Każdy test (scenariusz) tworzony jest w taki sposób, aby opisywać pewne zachowanie systemu. Jednym z postulatów, które pojawiły się także w przytoczonym artykule, jest nadawanie metodom testowym nazw, które de facto są zdaniami opisującymi nasze oczekiwania. W przypadku naszego testu moglibyśmy przemianować metodę na:

```
@Test
public void shouldReceiveTwentyPe
rcentOfDiscountIfRecipientIsJUGMe
mber () {
  ...
}
```

Sposób, w jaki piszemy testy, najczęściej powinien przebiegać według następującego schematu:

- definiowanie warunków początkowych
- wykonanie logiki podlegającej weryfikacji
- sprawdzenie zgodności otrzymanych rezultatów z oczekiwanymi

przekładając to na bardziej opisową konstrukcję moglibyśmy napisać:

mając określone warunki początkowe;

gdy wykonamy pewną operację w komponencie testowanym;

wówczas powinniśmy otrzymać oczekiwany rezultat;

Finalna wersja naszego testu, podążająca za konwencją BDD, mogłaby zatem wyglądać następująco:

```
@Test
public void shouldReceiveTwentyPe
rcentOfDiscountIfRecipientIsJUGMe
mber () {
  // given
  Participant p =
    new Participant();
  p.setJavaUserGroupMember(true);
  // when
  RegistrationService
    registrationService =
      new RegistrationService();
  PaymentDetails paymentDetails =
    registrationService.register(
      participant);
  // then
  assertEquals(80.0,
    paymentDetails.getPrice());
}
```

Szablon ten bezdyskusyjnie poprawia przejrzystość testu i ułatwia jego zrozumienie. Po raz pierwszy zobaczyłem go na prezentacji Szczepana Fabera podczas konferencji GeeCON. Jest to jednak tylko konwencja, która bez dyscypliny w zespole może zostać bardzo szybko zapomniana. Na szczęście istnieją biblioteki, które wspierają technikę *behaviour-driven development* i pozwalają nam myśleć o testach jak o realnych wymaganiach stawianych systemowi. Jedną z nich jest właśnie *easyb.*

Czym jest easyb?

Easyb to zrealizowany w Groovym język dedykowany (ang. *DSL; domain-specific language*), oferujący konstrukcje promowane przez BDD. Podstawowymi pojęciami są tu scenariusze (ang. *scenario*) oraz historie (ang. *story*), będące zbiorami scenariuszy. Spójrzmy ponownie na nasz test i spróbujmy określić wymaganie, które stawiamy modułowi do rejestracji uczest-

> *easyb* to tak naprawdę Groovy

ników:

Użytkownicy mający członkostwo w Java User Group powinni otrzymać 20% zniżki podczas rejestracji.

mając użytkownika z członkostwem w JUG

gdy zarejestruje się on na konferencję;

wówczas powinien otrzymać 20% zniżki;

W *easyb* nasz test (scenariusz) wyglądać będzie następująco:

```
scenario "Java User Group members should receive 20% discount", {
  given "participant is a JUG member"
  when "he registers for the conference"
  then "he should receive 20% discount"
}
```

Ponieważ *easyb* to tak naprawdę Groovy, możemy korzystać z dowolnych klas napisanych w języku Java, w tym także je testować. Drugą istotną zaletą tego języka są domknięcia (ang. *closure*), dzięki którym nasz kod staje się bardziej zwięzły, a scenariusz przedstawiony powyżej wykonywalną dokumentacją. Ostateczna postać naszego scenariusza mogłaby zatem mieć taką formę:

```
scenario „Java User Group members should receive 20% discount", {
  given "participant is a JUG member", {
     p = new Participant()
    p.setJavaUserGroupMember(true)
  }

  when "he registers for the conference", {
     registrationService =
       new RegistrationService()
     paymentDetails =
       registrationService.
         register(participant)
  }

  then "he should receive 20% discount", {
     paymentDetails.price.
       shouldBe 80.0
  }
}
```

Co ciekawe, test w niezaimplementowej formie jest również wykonywalny. W raporcie stworzonym przez *easyb* figurował będzie jako "w toku" (ang. *pending*).

```
Running user registration story (UserRegistrationStory.groovy)
Scenarios run: 1, Failures: 0, Pending: 1, Time elapsed: 0.89 sec

1 behavior ran (including 1 pending behavior) with no failures
```

Kolejną ciekawą cechą *easyb* są metody weryfikacji otrzymanych danych, przypominające opis w języku naturalnym. Porównując standardowe podejście:

```
assertEquals(80.0,
  paymentDetails.getPrice());
```

z tym, oferowanym przez *easyb* nie trzeba chyba nikogo przekonywać, które jest bardziej zrozumiałe:

```
paymentDetails.price.shouldBe 80.0
```

Easyb oferuje całą gamę metod do sprawdzania wyników scenariuszy. I tak do porównania dwóch obiektów mamy następujące wyrażenia:

- `shouldBe / shouldNotBe`
- `shouldEqual / shouldNotEqual`

możemy porównywać ich wartości:

- `shouldBeGreaterThan`

> Testy pisane w *easyb*
> mają tak naprawdę formę wykonywalnej,
> choć dość uproszczonej, dokumentacji

- `shouldBeLessThan`
- `shouldStartWith`
- `shouldEndWith`

jak również weryfikować ich typy:

- `shouldBeA / shouldNotBeA`
- `shouldBeAn / shouldNotBeAn`

nie może także zabraknąć wygodnych metod do sprawdzania zawartości kolekcji:

- `shouldHave / shouldNotHave`

Byłoby dużą niesprawiedliwością i nadużyciem przemilczenie istnienia takich bibliotek javowych jak *Hamcrest* czy *FEST-Assert*, które również oferują zbliżoną do języka naturalnego weryfikację wyników testu.

Dodatkowe zalety

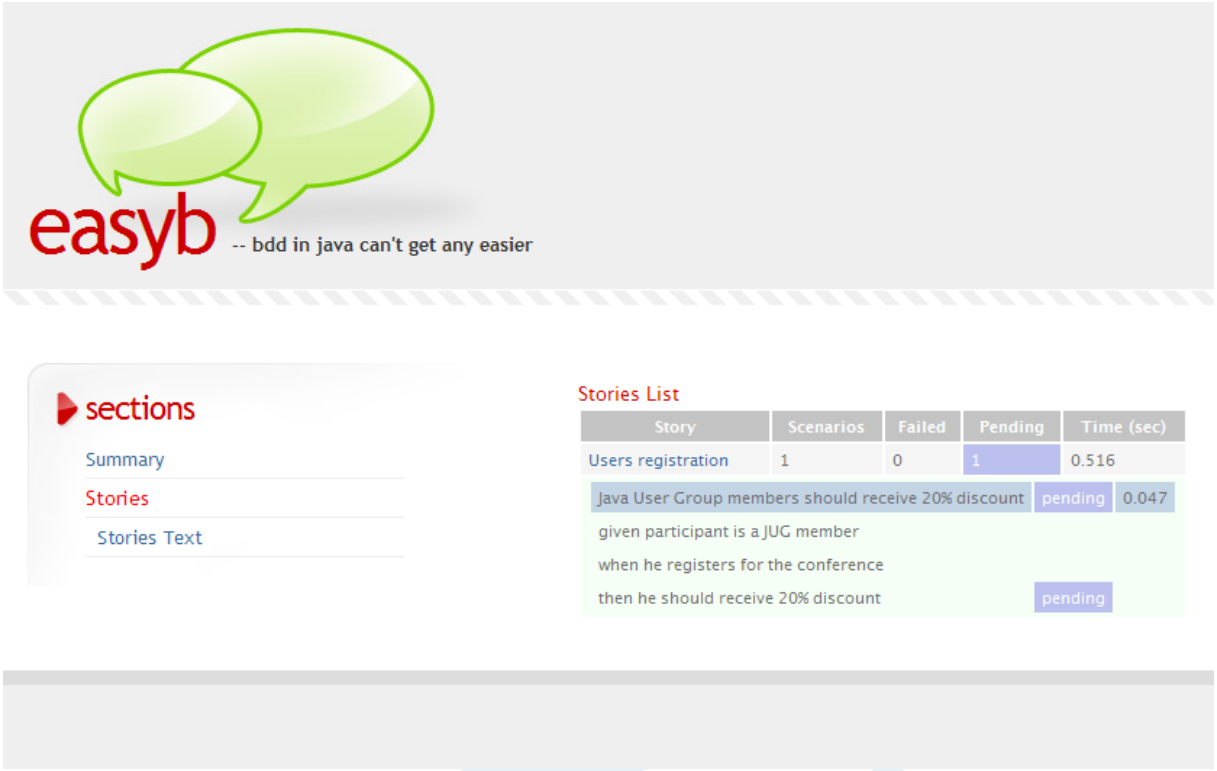
Jak już wspomniałem wcześniej, testy pisane w *easyb* mają tak naprawdę formę wykonywalnej, choć dość uproszczonej, dokumentacji. Stąd wykonując scenariusze, możemy na ich podstawie generować raporty, które traktować możemy jako zręby dokumentacji systemu. W tym momencie oferowane są nam dwie opcje – prosta forma tekstowa oraz raport w postaci strony HTML. Poza informacjami zilustrowanymi rysunkiem 1 i 2, zawierać one mogą także szczegółowe informacje o błędach, które pojawiły się podczas wykonywania scenariuszy.

```
1 scenario executed successfully.

Story: Users registration

scenario Java User Group members
should receive 20% discount
  given participant is a JUG mem-
ber
  when he registers for the con-
ference
  then he should receive 20% di-
scount
```

Scenariusze tworzone w *easyb* cechuje

Rys 1. Podsumowanie scenariuszy w postaci strony HTML

> Oczywiście easyb to nie jedyne narzędzie dla JVM do tworzenia testów w oparciu o behaviour-driven development.



Rys 2. Szczegóły scenariusza będącego w „toku”

prosta i zrozumiała struktura. Pokusić by się można zatem o zaangażowanie do ich tworzenia członków projektu, którzy niekoniecznie mają na co dzień do czynienia z kodem. W dobie narzędzi takich jak *FitNesse* czy *Concordion* wydaje się jednak mało prawdopodobnym, aby osoby nietechniczne chętne były do opisywania wymagań w *easyb*. Od pewnego czasu spotkać jednak można pogłoski o przygotowywaniu aplikacji *Easiness*, która ma wyjść na przeciw oczekiwaniom takich użytkowników.

Integracja

Bez wsparcia dla narzędzi takich jak IDE czy serwer ciągłej integracji nie możemy w pełni wykorzystać zalet, jakie oferują nam solidne testy. W momencie oddawania tego artykułu dla easyb dostępna jest wtyczka do IntelliJ IDEA oraz, od niedawna, do Eclipse. Poza tym możemy uruchomić scenariusze napisane w *easyb* za pomocą Apache Ant, Maven bądź z linii komend.

Alternatywy

Oczywiście *easyb* to nie jedyne narzędzie dla JVM do tworzenia testów w oparciu o *behaviour-driven development*. Oto niektóre z nich:

- Groovy - GSpec
- Java - JBehave, Cuke4Duke, GivWenZen
- Scala - Specs

Konkluzja

W artykule tym zaprezentowana została ciekawa alternatywa pisania testów, jaką jest *behaviour-driven development*, a w szczególności *easyb*. Pozwala ona koncentrować się na wymaganiach, a testy tworzyć w taki sposób, aby zrozumiałe były także dla osób bez doświadczenia informatycznego. Oczywiście możliwości i sposo-

> Bardzo dobrym zastosowaniem, szczególnie uwydatniającym zalety *easyb*, są testy funkcjonalne interfejsu użytkownika.

by implementowania scenariuszy w *easyb* są dużo szersze. W artykule tym przedstawione zostały jedynie najistotniejsze jego cechy i funkcjonalności. Gorąco zachęcam do odwiedzenia stron podanych w sekcji **Źródła**, ale przede wszystkim do poeksperymentowania. Bardzo dobrym zastosowaniem, szczególnie uwydatniającym zalety *easyb*, są testy funkcjonalne interfejsu użytkownika. Opis interakcji użytkownika z naszą aplikacją, dzięki scenariuszom, wygląda bardzo naturalnie. W tandemie z wygodną biblioteką emulującą zachowanie użytkownika w przeglądarce tworzy naprawdę ciekawe rozwiązanie, ale to temat na kolejny artykuł.

Źródła

1. http://dannorth.net/introducing-bdd - wprowadzenie do *behaviour-driven development*
2. http://easyb.org/ - strona główna projektu easyb
3. http://parleys.com/display/PARLEYS/Home#talk=28573704 – prezentacja o easyb z konferencji Devoxx
4. http://fitnesse.org/ - strona projektu FitNesse
5. http://code.google.com/p/givwenzen/ - strona projektu GivWenZen
6. http://www.concordion.org/ - strona projektu Concordion
7. http://code.google.com/p/specs/ - strona projektu Specs
8. http://groovy.codehaus.org/Using+GSpec+with+Groovy - wprowadzenie do Spec
9. http://jbehave.org/ - projekt JBehave
10. http://wiki.github.com/aslakhellesoy/cuke4duke - strona projektu Cuke4Duke
11. http://code.google.com/p/hamcrest/ - biblioteka Hamcrest
12. http://fest.easytesting.org/assert/wiki - biblioteka FEST-Assert

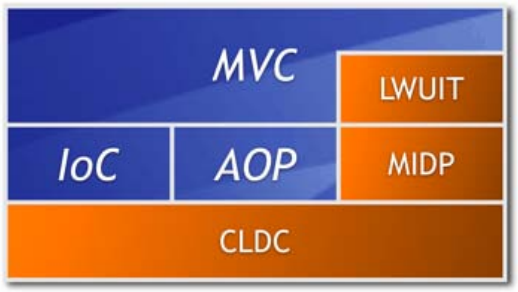
Signal Framework dla platformy Java ME

Marek Wiącek

Wprowadzenie

Signal Framework to open-source'owa biblioteka IoC, AOP oraz MVC przeznaczona dla Javy ME (J2ME) oraz oparta o Springa. Framework został zaprojektowany w taki sposób, aby przezwyciężyć ograniczenia CLDC, które uniemożliwiają uruchamianie kontenerów IoC opartych o Jave SE na platformie J2ME. Signal Framework używa zwykłych XMLowych plików konfiguracyjnych Springa, pozwalając programistom na wykorzystanie istniejących narzędzi i umiejętności.

Poniższy diagram przedstawia architekturę biblioteki:



Inspiracją dla nazwy frameworka jest biblioteka Antenna (http://antenna.sourceforge.net/) oraz, oczywiście, Spring Framework.

Kontener IoC

Wsparcie dla refleksji w CLDC jest bardzo ograniczone w porównaniu do Javy SE. API pozwala jedynie na tworzenie obiektów za pomocą domyślnych (bezargumentowych) konstruktorów. Refleksja nie pozwala na przekazywanie argumentów do konstruktorów, wywoływanie metod, dostęp do pól, ani tworzenie dynamicznych proxy.

Aby przezwyciężyć te ograniczenia framework IoC przetwarza pliki konfiguracyjne kontekstu podczas kompilowania aplikacji oraz generuje kod Javy odpowiedzialny za tworzenie kontekstu po uruchomieniu aplikacji. Kiedy aplikacja J2ME jest uruchamiana, wykonuje ona wygenerowany kod zamiast przetwarzania plików konfiguracyjnych. Dzięki temu kontekst może zostać utworzony bez parsowania plików XML lub używania zaawansowanej refleksji. Rozmiar wygenerowanego kodu jest bardzo mały, ponieważ biblioteka IoC zawiera jedynie około 10 klas. Wygenerowany kod *nie* zależy od bibliotek Springa.

Mimo tego, że framework został stworzony na potrzeby platformy J2ME, kontener IoC może być używany w dowolnych aplikacjach Java, które potrzebują wykorzystać funkcjonalność Springa bez polegania na refleksji lub parsowania plików XML (np. platformy Android oraz GWT).

Signal Framework wspiera następujące funkcje kontenera IoC zaimplementowanego w Springu:

- Pliki konfiguracyjne XML
- Komponenty (beans) typu singleton
- Wstrzykiwanie zależności poprzez argumenty konstruktorów oraz własności (properties) obiektów
- Autowiring
- Inicjalizacja na żądanie (lazy initialization)
- Przetwarzanie komponentów (bean post-processors) (`com.auroraosftworks.signal.runtime.core.context.IBeanProcessor` – odpowiednik interfejsu `org.springframework.beans.factory.config.BeanPostProcessor`)
- Lekkie AOP oparte o automatycznie generowane klasy proxy

> Signal Framework to
> open-source'owa biblioteka IoC, AOP oraz MVC
> przeznaczona dla Javy ME

- `com.aurorasoftworks.signal.runtime.core.context.IInitializingBean` - odpowiednik interfejsu `org.springframework.beans.factory.InitializingBean`
- `com.aurorasoftworks.signal.runtime.core.context.IContextAware` – odpowiednik interfejsu `org.springframework.beans.factory.BeanFactoryAware`

Generator kodu jest na ogół uruchamiany jako wtyczka Mavena, która wymaga 2 parametrów: nazwy pliku konfiguracyjnego Springa oraz nazwy klasy Java, która zostanie wygenerowana. Generator obsługuje znacznik `<import resource="…">`, jest więc możliwe przetworzenie wielu plików konfiguracyjnych poprzez jedno uruchomienie wtyczki.

Poniżej przedstawiono przykład pliku konfiguracyjnego Springa oraz odpowiadającą mu wygenerowaną klasę:

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
  <bean id="greeter"
  class="com.aurorasoftworks.signal.examples.context.core.Greeter">
    <constructor-arg ref="msgSource" />
  </bean>
  <bean id="msgSource"
    class="com.aurorasoftworks.signal.examples.context.core.
      MessageSource">
    <constructor-arg>
      <map>
        <entry key="greeting" value="Hello!" />
      </map>
    </constructor-arg>
  </bean>
</beans>

public class ApplicationContext extends
  com.aurorasoftworks.signal.runtime.core.context.Context
{
  public ApplicationContext() throws Exception
  {
    /* Begin msgSource */
    ApplicationContext.this.registerBean(„msgSource",
      new com.aurorasoftworks.signal.examples.context.core.
        MessageSource(new java.util.Hashtable(){{
          put(„greeting", „Hello!"); }}));
    /* End msgSource */
    /* Begin greeter */
    ApplicationContext.this.registerBean(„greeter",
      new com.aurorasoftworks.signal.examples.context.core.
        Greeter(((com.aurorasoftworks.signal.examples.context.core.
          MessageSource) GreeterContext.this.getBean(„msgSource"))));
    /* End greeter */
    }
}
```

> Implementacja AOP dostarczona przez Signal Framework ma niewielkie wymagania pamięciowe

Biblioteka obecnie nie obsługuje narzędzia Ant, ale zadania Anta (tasks) mogą być łatwo zaimplementowane jako lekka nakładka na istniejący generator.

Framework AOP

Oprócz utrudniania implementacji IoC, ograniczenia API CLDC uniemożliwiają również użycie istniejących frameworków AOP na urządzeniach J2ME. API AOP Alliance oraz popularne biblioteki AOP, takie jak AspectJ oraz JBoss AOP, zależą od typów zawartych w pakiecie `java.lang.reflect.*`, które nie są zaimplementowane w Javie ME. Dodatkowo implementacje AOP często wykorzystują własne implementacje class-loaderów i/lub dynamiczne proxy które nie są obsługiwane przez implementacje CLDC.

Implementacja AOP dostarczona przez Signal Framework została zaprojektowana na potrzeby urządzeń J2ME: ma niewielkie wymagania pamięciowe, używa jedynie klas obecnych w API CLDC oraz alokuje podczas działania najmniejszą możliwą ilość obiektów. To podejście pociąga jednak za sobą pewne ograniczenia: framework nie jest tak rozbudowany jak jego odpowiedniki dla aplikacji desktopowych i klasy enterprise oraz obsługuje jedynie przechwytywanie metod wywoływanych na interfejsach.

Framework AOP przezwycięża ograniczenia Javy ME poprzez generację kodu. Kiedy kontekst aplikacji jest przetwarzany podczas kompilacji, framework identyfikuje komponenty implementujące interfejs `com.aurorasoftworks.signal.runtime.core.context.proxy.IProxy`, a następnie tworzy dla nich klasy proxy. Instancje klas proxy są stosowane do przechwytywania wywołań metod oraz wywoływania interceptorów.

Ta koncepcja jest podobna do mechanizmu dynamicznych proxy wspieranych przez Jave SE. Większość klas zawartych we frameworku AOP ma swoje odpowiedniki w Javie SE, co przedstawiono w Tabeli 1.

Obiekty proxy są na ogół tworzone przez klasę `ProxyFactory`. Najbardziej powszechną metodą tworzenia instancji proxy jest przekazanie listy interceptorów do metody `IProxyFactory#createProxy(IProxyTarget target, IMethodInterceptor [] interceptors)`. Zwracany obiekt implementuje te same interfejsy, co przekazany argument i może być bezpiecznie rzutowany na te interfejsy.

Przechwytywanie metod zostało przedstawione na Diagramie 1.

Z uwagi na swoją prostotę biblioteka posiada pewne ograniczenia. Obiekty proxy stworzone przez bibliotekę wielokrotnie

Typ w bibliotece Signal AOP (com.aurorasoftworks.signal.runtime.core.*)	Odpowiednik w Javie SE
`context.proxy.ProxyFactory`	`java.lang.reflect.Proxy`
`context.proxy.IInvocationHandler`	`java.lang.reflect.InvocationHandler`
`context.proxy.IMethodInterceptor`	`org.aopalliance.intercept.MethodInterceptor`
`context.proxy.IProxy`	wartość zwracana przez `java.lang.reflect.Proxy.newInstance`
`context.proxy.IProxyTarget`	dowolny obiekt
`context.proxy.IProxyClass`	`java.lang.Class`
`context.proxy.IMethodHandler`	`java.lang.reflect.Method`

Tabela 1. Odpowiedniki klas Signal AOP w Java SE.

> Framework MVC wspiera zarówno API MIDP jak i LWUIT

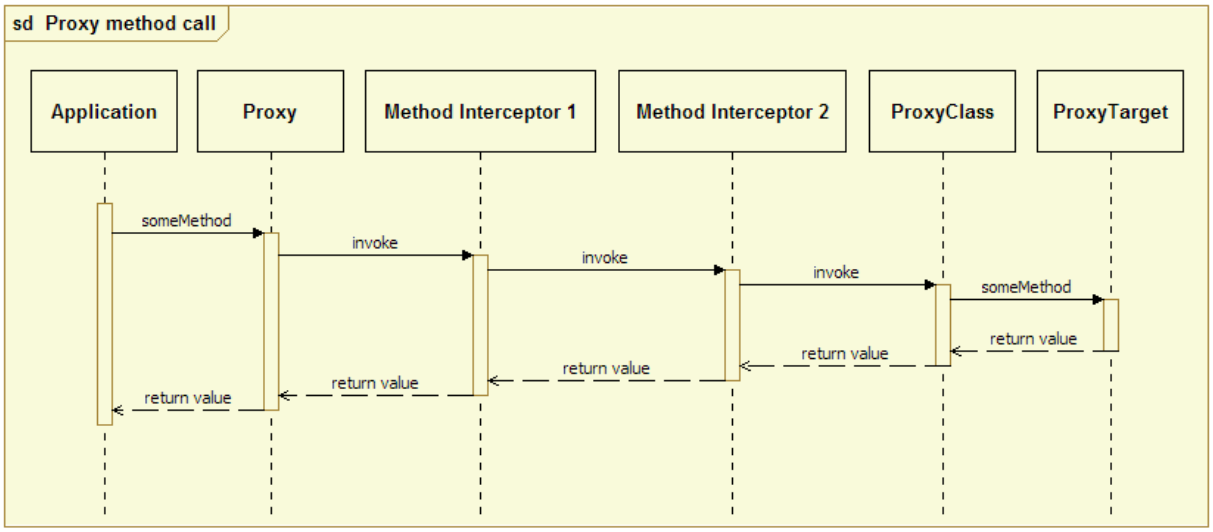


Diagram 1. Przechwytywanie metod

wykorzystują instancje tablic przeznaczone do przekazywania argumentów do interceptorów aby uniknąć tworzenia tymczasowych obiektów. To jednak oznacza, że kod klas proxy musi być synchronizowany (odpowiada za to generator kodu). W wielowątkowych aplikacjach może to prowadzić do problemów z wydajnością. Kod proxy jest synchronizowany na instancji proxy, co oznacza że wiele instancji proxy tego dla samego obiektu może być używanych współbieżnie bez blokowania żadnych wątków.

Kiedy typy podstawowe są przekazywane do lub zwracane przez metodę proxy muszą one zostać opakowane za pomocą typów obiektowych takich jak `java.lang.Integer`. Jest to jedyny scenariusz, który wymaga tworzenia tymczasowych obiektów. W pozostałych przypadkach biblioteka AOP nie potrzebuje tworzyć żadnych tymczasowych obiektów i może być bezpiecznie używana niezależnie od jakości garbage collectora.

Framework MVC

Framework MVC jest oparty o funkcjonalność IoC oraz AOP opisaną w poprzednich sekcjach. Framework wspiera zarówno API MIDP jak i LWUIT oraz w razie potrzeby może być łatwo rozszerzony o wsparcie dla innych technologii prezentacji.

Framework nie nakłada żadnych ograniczeń na model domenowy, ani widoki, pod warunkiem że używany jest LWUIT lub MIDP. Zamiast tego, biblioteka jest zaprojektowana tak, aby ułatwić tworzenie kontrolerów. Najważniejsze funkcje zaimplementowane w warstwie kontrolera to inicjalizacja kontrolerów (oraz powiązanych widoków, jeśli istnieją) na żądanie (lazy initialization) oraz deklaratywne reguły nawigacji zdefiniowane w pliku konfiguracyjnym IoC.

Kontroler to zwykły komponent (bean) zdefiniowany w kontekście IoC aplikacji. Wszystkie kontrolery muszą implementować interfejs `com.aurorasoftworks.signal.runtime.ui.mvc.ICotroller` lub interfejsy dziedziczące po nim. Kontroler zarządza częścią interfejsu aplikacji, którą może być pojedynczy widok lub skomplikowany kreator (wizard) składający sie z wielu kroków.

Najbardziej powszechym rodzajem kon-

> Najważniejszym komponentem w aplikacji MVC jest dyspozytor (dispatcher).

trolera jest kontroler widoku. Kontrolery widoku dla API MIDP oraz LWUIT powinny implementować odpowiednio interfejsy `com.aurorasoftworks.signal.runtime.ui.mvc.midp.IViewController` oraz `com.aurorasoftworks.signal.runtime.ui.mvc.lwuit.IViewController`. Framework automatycznie przekazuje komendy (`javax.microedition.lcdui.Command` i `com.sun.lwuit.Command`) do kontrolera widoku, który je wygenerował. Wiele kontrolerów może być powiązanych z widokiem.

Drugim typem kontrolera jest kontroler przepływu (flow controller), który zarządza reużytkowalnym procesem, na ogół kreatorem składającym się z wielu widoków. Po zakończeniu działania przepływ powraca do miejsca, w którym został rozpoczęty, podobnie jak wywołanie metody. Kontrolery przepływu powinny implementować interfejs `com.aurorasoftworks.signal.runtime.ui.mvc.IFlowController`. Przepływy można ze sobą łączyć: jeden przepływ może startować inne przepływy, w tym nowe instancje tej samej klasy kontrolera przepływu.

Zależności między kontrolerami są definiowane jako zależności między komponentami, dzięki czemu kontrolery mogą „generować" zdarzenia wywołując metody interfejsów. Takie rozwiązanie skutkuje luźnym powiązaniem kontrolerów, deklaratynymi definicjami reguł nawigacji i silnym typowaniem. Wywołania metod interfejsów są przechwytywane przez framework aby w przezroczysty sposób zrealizować niezbędne przetwarzanie takie jak wyświetlanie właściwego widoku, rejestrowanie obserwatorów dla komend oraz inicjaowanie i kończenie przepływu.

W większości przypadków pożądane jest aby kontrolery i widoki były inicjalizowane na żądanie. W takim wypadku kontekst aplikacji musi być odpowiednio skonfigurowany:

```
<beans xmlns=
  "http://www.springframework.org/
schema/beans"
xmlns:xsi=
  "http://www.w3.org/2001/XMLSche-
ma-instance"
   xsi:schemaLocation="http://www.
springframework.org/schema/beans
http://www.springframework.org/
schema/beans/spring-beans.xsd"
  default-lazy-init="true">
  <!-- ... -->
</beans>
```

Najważniejszym komponentem w aplikacji MVC jest dyspozytor (dispatcher). Dyspozytor to obiekt dostarczany przez bibliotekę, który przechwytuje wywołania metod i realizuje niezbędne przetwarzanie, takie jak wyświetlanie odpowiedniego widoku lub rejestracja obserwatora komend:

```
<bean id="dispatcher"
  class=
     "com.aurorasoftworks.signal.
runtime.ui.mvc.midp.Dispatcher"
/>
```

Istnieją dwie implementacje koncepcji dyspozytora: `com.aurorasoftworks.signal.runtime.ui.mvc.midp.Dispatcher` oraz `com.aurorasoftworks.signal.runtime.ui.mvc.lwuit.Dispatcher`, używane odpowiednio w aplikacjiach MIDP oraz LWUIT.

Po definicji dyspozytora należy umieścić definicje kontrolerów oraz widoków, co pokazano poniżej. Zależności między kontrolerami są definiowane jako zależności między komponentami IoC.

```
<bean      id="accountListViewCtl"
class="com.aurorasoftworks.sig-
nal.examples.ui.mvc.midp.Accoun-
tListViewController">
```

> Dystrybucja kodu źródłowego frameworka zawiera przykładową aplikację

```
  <constructor-arg
    ref="accountListView" />
  <constructor-arg
    ref="accountService" />
  <property name="newAccountEvent"
ref="newAccountCtl" />
  <property name="editAccountEvent"
ref="editAccountCtl" />
</bean>
```

W powyższym przykładzie kontroler nazwany `accountListViewCtl` obsługuje dwa rodzaje zdarzeń: `newAccountEvent` oraz `editAccountEvent`, które są powiązane z dwoma innymi kontrolerami jako własności komponentów. Te zdarzenia to zwykłe referencje do interfejsów zdefiniowanych poniżej. Kontrolery „generują" zdarzenia poprzez wywoływanie metod interfejsów, które są z kolei przechwytywane przez dyspozytora.

```
public interface INewAccountEvent
{
  void onNewAccount();
}

public interface IEditAccountE-
vent
{
  void onEditAccount(
    IAccount account);
}
```

Diagram 2 przedstawia sekwencję wywołań niezbędną do obsłużenia akcji użytkownika w scenariuszu opisanym powyżej.

Komenda wybrana przez użytkownika jest wysyłana do dyspozytora (`CommandListener.commandAction`), który z kolei przekazuje ją do aktywnego kontrolera (`ICommandHandler.handleCommand`). Instancja `AccountListViewController` reaguje na komendę generując zdarzenie `INewAccountEvent.onNewAccount`, które jest przechwytywane przez dyspozytora. Dyspozytor deaktywuje instancję `AccountListViewController`, zmienia bieżący widok na ten, który jest powiązany z instancją `NewAccountViewController` oraz aktywuje go. Przejście z jednego kontrolera na drugi jest prawidłowo odzwierciedlone w stanie aplikacji: `NewAccountViewController` staje się aktywnym kontrolerem, a jego powiązany widok jest wyświetlony na ekranie.

Dystrybucja kodu źródłowego frameworka zawiera przykładową aplikację zaimplementowaną w technologiach MIDP oraz LWUIT, która demonstruje prawidłowe użycie biblioteki MVC.

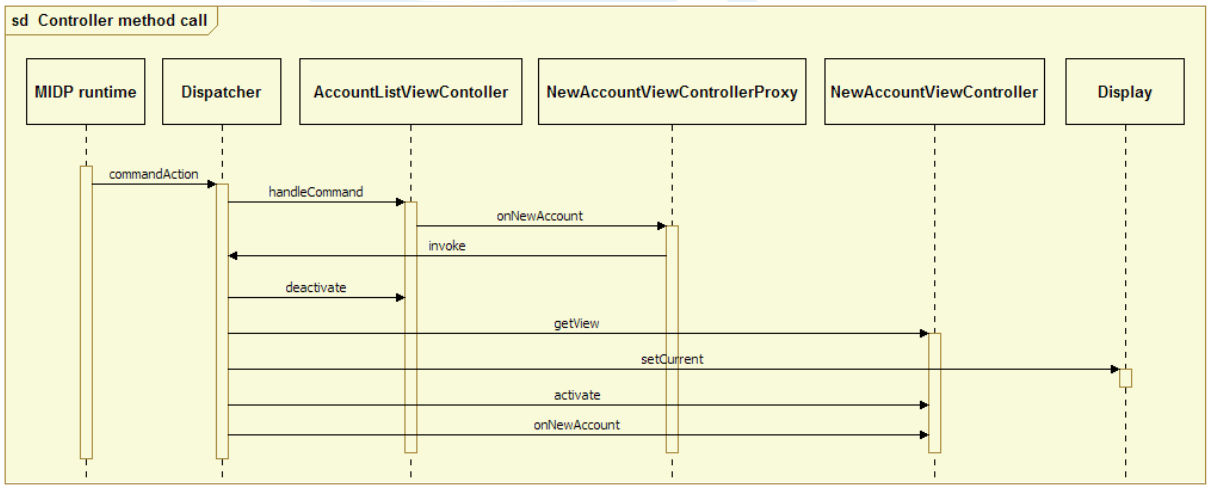


Diagram 2. Sekwencja wywołań obsługi akcji

> Biblioteka została zaprojektowana
> aby pogodzić dwa sprzeczne wymagania

Podsumowanie

Biblioteka została zaprojektowana aby pogodzić dwa sprzeczne wymagania: wykorzystać jak największą część funkcjonalności Springa oraz ułatwić dodanie wsparcia dla innych kontenerów IoC w przyszłości, jeśli będzie to potrzebne. Funkcjonalność opisana w tym artykule została zrealizowana dość małym kosztem: około 10 000 linii kodu nie licząc przykładowych aplikacji.

Pewne uproszczenia musiały zostać zaakceptowane z uwagi na ograniczenia platformy Java ME; przede wszystkim framework nie wspiera annotacji i zamiast tego wymaga w pewnych sytuacjach aby klasy aplikacji implementowały interfejsy biblioteki. Z tego samego powodu szablony klas (generics) nie są stosowane w API frameworka.

Po kilku wersjach beta framework stał się dość dojrzały i zawiera wszystkie znaczące funkcje, które były planowane. Wersja produkcyjna powinna być dostępna najpóźniej w styczniu 2010.

Dodatkowe informacje można znaleźć w następującej lokalizacji: http://www.aurorasoftworks.com/products/signalframework.

Gradle - mocarne narzędzie do budowy projektów

Tomasz Kaczanowski

Wstęp

„Jakiego narzędzia używasz do budowy projektu i dlaczego?" W świecie Javy, na tak zadane pytanie można spodziewać się jednej z dwóch odpowiedzi:

> *Używam Mavena. Świetne pluginy pozwalają mi w trzech liniach uzyskać efekt, na który potrzebowałbym kilkadziesiąt linii XMLa, gdybym używał Anta. Czasami przeszkadza mi brak elastyczności Mavena, ale radzę sobie wówczas używając plugina AntRun.*
>
> *Użytkownik Mavena*

> *Używam Anta i dzięki jego elastyczności mogę zrobić wszystko - co tylko zechcę. Maven zdobył popularność dzięki zarządzaniu dependencjami, ale ja mam Apache Ivy, który potrafi o wiele więcej.*
>
> *Użytkownik Anta*

Nie mam zamiaru powtarzać tu argumentów, którymi zwolennicy Anta i Mavena wymieniali się wielokrotnie w niezliczonych dyskusjach na czatach, forach i blogach. Po pierwsze dlatego, że nie pojawiły się żadne nowe argumenty na rzecz któregokolwiek z tych dwóch rozwiazań. Po drugie dlatego (i to jest główny powód), że na scenie narzędzi do budowy projektów pojawił się nowy zawodnik - *Gradle*. I jest to gracz tak silny, że ma wszelkie szanse by usunąć w cień zarówno Anta jak i Mavena, a dyskusję o wyższości jednego nad drugim przesunąć do historii komputerowych „flame wars".

Nim zajmiemy się Gradlem, przyjrzyjmy się najbardziej znanym (czyt. uciążliwym) słabościom Anta i Mavena:

- ciężko jest zaimplementować jakikolwiek algorytm w skrypcie budującym; trudno wyrazić nawet tak podstawowe konstrukcję jak `if` czy `for`,
- ciężko jest wykonać nawet pozornie proste taski, których wykonywanie nie zostało przewidziane przez developerów Anta/Mavena, i które nie są dostępne w postaci gotowych tasków lub pluginów,
- potrzebna jest dogłębna znajomość tych narzędzi, by zrozumieć złożony skrypt budujący,
- podejście „build by convention" nie jest wspierane (Ant), albo skutecznie utrudnia konfigurację (Maven),
- wsparcie dla budowy projektów wielomodułowych jest niewystarczające,
- skrypty budujące są niepotrzebnie duże i nieczytelne z uwagi na narzut samego języka (XML),

Aby móc w pełni skorzystać z lektury tego artykułu, zachęcam do instalacji Gradle. Instrukcja instalacji znajduje się na stronie projektu. Dla Twojej wygody do artykułu dołączone są kody źródłowe. Zarówno artykuł jak i kod źródłowy jest zgodny z wersją 0.8 Gradle (najnowszą w momencie pisania artykułu).

Celem tego artykułu jest przedstawienie projektu Gradle i sprawienie, byś zapra-

> Jakiego narzędzia używasz do budowy projektu i dlaczego?

gnął poznać go lepiej. Nie zamierzam konkurować z wyczerpującą dokumentacją ani z dołączonymi do dystrybucji Gradle przykładami.

Gradle - krótkie wprowadzenie

W artykule przedstawię Gradle prezentując kilka z życia wziętych przykładów jego zastosowania. Jednak nim to nastąpi, pozwolę sobie pokrótce scharakteryzować ten projekt, tak byś wiedział czego możesz oczekiwać.

Gradle jest bardzo elastycznym narzędziem do budowy projektów, w szczególności projektów Javy. Umożliwia pisanie skryptów budujących w opartym na Groovym DSLu, co sprawia, że wyrażenie w nich algorytmów staje się banalnie proste. Gradle pozwala na tworzenie zależności między zadaniami (taskami) i opiera się na paradygmacie "convention over configuration" (z tym że konfiguracja jest zawsze możliwa i łatwa do przeprowadzenia). Gradle wykracza poza możliwości innych narzędzi, jednocześnie stosuje się do przyjętych przez społeczność Javy niepisanych standardów, obniżając w ten sposób koszt związany ze zmianą narzędzia budującego.

Gradle jest nadal w trakcie developmentu, osiągnął już jednak stabilność, pozwalającą na jego produkcyjne użycie. Lista zaimplementowanych funkcjonalności jest już obecnie imponująca (na uwagę zasługuje m.in. zaawansowane wsparcie dla projektów wielomodułowych), a do realizacji czeka jeszcze wiele wartościowych pomysłów (m.in. możliwość pisania skryptów budujących w DSLach opartych o inne niż Groovy języki JVM - np. Scala, JRuby).

Pierwszy skrypt budujący

Obiecałem zaprezentować Gradle poprzez przykłady jego użycia, zajmijmy się więc napisaniem kawałka kodu. Zaczniemy od bardzo prostej aplikacji webowej. Następnie rozszerzę ten przykład, co pozwoli mi zaprezentować różne cechy Gradle - przede wszystkim dynamiczną naturę jego skryptów oraz wsparcie dla projektów wielomodułowych.

Najmniejsza na świecie aplikacja webowa ;)

Layout naszego projektu prezentuje się następująco:

```
myWebApp
`- src
   `- main
      |- java
      |  `- org
      |     `- gradle
      |        `- sample
      |           `- Greeter.java
       `- webapp
          `- index.jsp
```

Jak widać projekt jest prosty aż do bólu. Ma tylko jedną klasę (`Greeter.java`), która wykorzystuje metody z biblioteki Commons Lang ze zbioru Apache Commons, oraz jedną stronę JSP - `index.jsp` - która używa tejże klasy.

build.gradle

Czas dodać skrypt budujący - nadamy mu domyślną nazwę `build.gradle`.

> Gradle jest bardzo elastycznym narzędziem do budowy projektów, w szczególności projektów Javy.

Skrypt ten powinien wykonać następujące zadania:

- załadować wymagane dependencje,
- skompliować klasy Javy,
- zbudować plik WAR.

A więc proszę - przed nami pierwszy skrypt budujący napisany z użyciem Gradle.

```
usePlugin ‚war'
version = 0.1

repositories {
  mavenCentral()
}

dependencies {
  compile
   „commons-lang:commons-lang:2.4"
}
```

Umieść ten plik w głównym folderze projektu myWebApp main i uruchom pisząc w linii komend **`gradle war`**. Wynikowy plik `myWebApp-0.1.war` znajdziesz w folderze `build/libs`. Zdeployuj go w dowolnym kontenerze webowym (np. Tomcatcie) i rozkoszuj się aplikacją działającą pod adresem `http://YourHostName/myWebApp-0.1`.

Podejrzewam, że w powyższym skrypcie Twoją uwagę przykuło kilka spraw:

- skrypty Gradle mogą być bardzo zwięzłe,
- nie ma XMLa, jest DSL,
- Gradle potrafi „rozmawiać" z repozytoriami Maven.

Tak, to wszystko prawda, ale zapewniam że to dopiero początek. Póki co, przyjrzyjmy się dokładniej temu, co (i jak) robi ten skrypt.

Informacje o procesie budowania

Używając Gradle możesz z łatwością dowiedzieć się wielu użytecznych informacji na temat procesu budowania projektu. Wykonaj komendę **`gradle --tasks`** aby poznać dostępne zadania (te napisane przez Ciebie i dostarczone przez zadeklarowane w skrypcie budującym pluginy). Wykonaj **`gradle --properties`** żeby poznać wszystkie propertiesy i **`gradle --dependencies`** by dowiedzieć się o wszystkich zależnościach projektu. Ale przede wszystkim wykonaj komendę **`gradle -?`** by poznać mnóstwo innych przydatnych komend.

DSL i „convention over configuration"

Prawdopodobnie zauważyłeś, że zaprezentowany plik `build.gradle` jest bardzo zwięzły. Jest to możliwe dzięki dwóm cechom Gradle:

- Gradle używa DSL, który jest o wiele bardziej zwięzły niż XML (łatwiej też go czytać i zrozumieć),
- Gradle działa zgodnie z paradygmatem "convention over configuration", co pozwala zminimalizować wielkość pliku budującego, pod warunkiem że używane są domyślne wartości (takie jak layout projektu, nazwy plików wynikowych itp.).

> Gradle stosuje rozpowszechnioną przez Mavena strukturę projektu

Powiedz to tak po prostu

Chciałbyś użyć w skrypcie plugina War ? Chciałbyś ustawić wersję na 1.0 ? Po prostu to powiedz:

```
usePlugin ‚war’
version = 0.1
```

Chciałbyś wypisać komunikat jeżeli zajdzie pewien warunek ? Po prostu to powiedz:

```
if (someCondition) {
  println “some message”
}
```

Jeżeli zastanowisz się nad tymi dwoma przykładami, dojdziesz do wniosku, że nie ma w nich nic niezwykłego. „Tak to powinno wyglądać”. Masz rację. A jednak, wyrażenie tych jakże prostych rzeczy jest bardzo trudne w Antcie i Mavenie. Gradle, używając mieszanki DSL i Groovyego, daje możliwość bezpośredniego wyrażania tego, co pragniesz powiedzieć.

Przykładowo, Gradle stosuje rozpowszechnioną przez Mavena strukturę projektu, i domyślnie zakłada, że pliki źródłowe znajdują się w folderach jak poniżej.[1]

Inna konwencja sprawia, że stworzony przez Gradle plik WAR domyślnie nosi nazwę tworzoną według wzorca `nazwa-Projektu-wersja.war` (z kolei nazwa projektu jest identyczna z nazwą folderu, w którym projekt się znajduje, o ile nie wyspecyfikowano inaczej). Inną domyślną wartością jest użycie kompilatora Javy w wersji 1.5.

Konfiguracja jest możliwa i łatwa do wykonania

Jeżeli zajdzie taka potrzeba, wówczas wszystkie powyższe domyślne ustawienia można z łatwością zmodyfikować. Przykładowo, załóżmy że chcemy, by wszelkie artefakty (pliki JAR i WAR) trafiały do katalogu `build/artifacts` zamiast do domyślnego `build/libs`. Wystarczy ustawić wartość zmiennej `libsDirName` na „artifacts”:

```
libsDirName = „artifacts”
```

Aby zmienić nazwę generowanego pliku JAR czy WAR należy z kolei ustawić wartość zmiennej `archivesBaseName`:

```
archivesBaseName = „i-dont-like-the-default-name”
```

Ponieważ prezentowana aplikacja myWebApp wykorzystuje domyślne ustawienia, nie ma więc potrzeby modyfi-

```
`-- src
    |-- main
    |   |-- groovy    // kod źródłowy pisany w Groovym
    |   |-- java      // kod źródłowy pisany w Javie
    |   |-- resources // zasoby aplikacji
    |   `-- webapp    // kod źródłowy aplikacji webowej
    `-- test
        |-- groovy    // testy napisane w Groovym
        |-- java      // testy napisane w Javie
        `-- resources // zasoby testowe
```

> Gradle pluginami stoi.

kowania wartości tych zmiennych, skutkiem czego plik budujący jest bardzo krótki.

W pluginach siła

Zgodnie z aktualną modą, Gradle pluginami stoi. :) Każdy plugin może rozszerzyć zasób funkcji oferowanych przez Gradle na trzy sposoby. Dla przykładu, użycie w skrypcie budującym pluginu Java niesie za sobą następujące konsekwencje:

- dostępnych staje się prawie 20 nowych tasków (np. `clean`, `compileJava`, `test`, `jar`),
- stworzone zostają zależności między taskami, które wymuszają rozsądną kolejność ich wykonywania (np. task `test` wykonywany jest po taskach `compileJava` i `compileJavaTests` tasks),
- do konfiguracji projektu dołączany jest tzw. „convention object", co skutkuje ustawieniem wielu domyślnych wartości, np.:
- `javadoc.destinationDir= file-(,build/docs/javadoc')`

  `archivesBaseName = projectName`

W zaprezentowanym skrypcie użyty został plugin War, co ma następujące konsekwencje:

- zaimportowany zostaje plugin Java (plugin War jest od niego zależny), a wraz z nim wszystkie jego zadania i ustawienia konfiguracyjne,
- dodany zostaje task `war`, który robi to, czego należy oczekiwać po tasku `war`:
  - kopiuje pliki z `src/main/webapp` do głównego katalogu archiwum,
  - kopiuje skomplikowane klasy do `WEB-INF/classes`,
  - umieszcza wszelkie zależności runtime w katalogu `WEB-INF/lib`,
  - tworzy plik archiwum WAR w katalogu `build/libs`

Obecnie Gradle oferuje 9 pluginów (Java, Groovy, War, OSGi, Eclipse, Jetty, Maven, Project-reports, Code-quality), a jeżeli nie znajdziesz wśród nich potrzebnej Ci funkcjonalności, zawsze możesz napisać swój własny (jest to bardzo proste, acz wykracza poza ramy tego artykułu - zerknij proszę do dokumentacji Gradle).

Repozytoria i zarządzanie zależnościami

Mam dla Ciebie dobrą wiadomość - jak już przepiszesz swoje skrypty budujące na Gradle, będziesz mógł nadal używać tych samych repozytoriów artefaktów, których używasz obecnie. Gradle opiera się na Apache Ivy, i potrafi „dogadać się" z każdym możliwym do wyobrażenia typem repozytorium - od „standardowych" repozytoriów Mavena, poprzez staroświeckie foldery `libs` używane tradycyjnie w buildach Anta, aż po niemal dowolne repozytorium o zdefiniowanej

> Osiągnięcie rzeczy prostych
> jest w Gradle nie tylko łatwe,
> ale i daje się elegancko wyrazić.

przez programistę strukturze.

Jak widać na przykładzie pierwszego skryptu, osiągnięcie rzeczy prostych jest w Gradle nie tylko łatwe, ale i daje się elegancko wyrazić. Aby Gradle przeszukiwał centralne repozytorium Mavena w poszukiwaniu wymaganych artefaktów, wystarczy dodać takie oto linijki do skryptu budujacego:

```
repositories {
  mavenCentral()
}
```

Deklaracje dependencji są o wiele bardziej zwięzłe, niż te, którymi posługujesz się w Mavenie 2[2] lub Ivy. Zamieszczony poniżej kawałek kodu dodaje JAR biblioteki Commons Lang do classpatha komplikacji:

```
dependencies {
  compile
   „commons-lang:commons-lang:2.4”
}
```

Pisanie własnej logiki

Jak dotąd polegaliśmy wyłącznie na funkcjonalnościach oferowanych przez Gradle. Najwyższy czas, by dorzucić nieco własnej wymyślnej funkcjonalności. W ten sposób poznamy też klika nowych możliwości Gradle.

Liczenie sum kontrolnych pliku (z użyciem Anta)

Napiszemy teraz taska, który policzy sumy kontrolne plików z katalogu `build/libs` (domyślny katalog, do którego trafiają stworzone pliki JAR i WAR). Ponieważ wierzymy mocno w reużycie kodu, wykorzystamy w tym celu task checksum dostępny w Antcie. Dodaj poniższy kawałek kodu do pliku `build.gradle` i wykonaj polecenie **`gradle clean checksum`**.

Ant i Gradle

Gradle zapewnia świetną integrację z Antem. Możesz zaimportować Antowy plik `build.xml` bezpośrednio do skryptu budującego Gradle. Dzięki temu wszelkie targety z pliku Anta, staną się dostępne jako taski Gradle. Inny typ integracji zapewnia dostaczana przez Groovyego klasa AntBuilder. Obiekt `ant` jest dostępny w każdym skrypcie budującym Gradle, dzięki czemu możesz używać całego bogactwa tasków Anta - począwszy od prostych takich jak `javac`, `copy`, `jar`, aż po bardziej złożone - np. `selenese` (do uruchamiania testów Selenium) czy `findbugs`(do generowania raportów ze statycznej analizy kodu narzędziem Findbugs).

To wszystko czyni migrację z Anta do Gradle możliwie bezbolesną... a gdy już jej dokonasz, to nie spodziewam się byś kiedykolwiek zechciał do Anta powrócić. :)

```
task checksum (
  dependsOn: assemble) << {
  def files = file(libsDir).
    listFiles()
    files.each { File file ->
      ant.checksum(file: file,
        property: file.name)
       println „$file.name  Check-
sum:
${ant.properties[file.name]}”
    }
}
```

W tych kilku linijkach kodu dzieje się cał-

> Gradle zapewnia świetną integrację z Antem.

kiem sporo - widać użycie skryptów Groovyego, definiowanie zależności miedzy taskami i użycie tasków Anta.

Po pierwsze, tworzony jest nowy task o nazwie `checksum`, który zależy od taska `assemble` dostarczanego przez plugin Java (ten task odpowiedzialny jest za tworzenie plików JAR i WAR). Następna linijka, napisana w Groovym, sprawia że wszystkie pliki z katalogu `libsDir` (wartosc tego property jest domyślnie ustawiana przez plugin Java i wskazuje na katalog`build/libs`) trafiają do listy. W końcu na każdym z plików z listy wywoływany jest Antowy task `checksum`, a wynik wypisywany jest na standardowe wyjście.

Snapshoty i wersje stabilne

W świecie Javy przyjęło się, że nazwy artefaktów odzwierciedlają fakt, czy artefakty te są stabilne (tzw. releasy) czy też nie (snapshoty). Snapshoty zwyczajowo otrzymują suffix „-SNAPSHOT”, co sprawia że łatwo odróżnić je od wersji stabilnych. Zaimplementujmy teraz w naszym skrypcie budującym ten schemat nazewniczy artefaktów.

Poniższy fragment pliku `build.gradle` pokazuje bardzo interesującą cechę Gradla. Otóż wywołanie skryptu budującego podzielone jest na dwie fazy. W pierwszej konstruowany jest acykliczny graf zależności miedzy taskami (DAG, ang. dependency acyclic graph). W drugiej fazie następuje właściwe wywołanie tasków. Poniższy prosty przykład pokazuje, że jest możliwe „wejście” pomiedzy te dwie fazy i wykonanie pewnych operacji na grafie tasków, nim nastąpi ostateczne wykonanie tasków. W tym prostym przykładzie, graf tasków jest analizowany i na podstawie tej analizy podejmowana jest decyzja o nazwie wynikowego pliku:

```
task release(dependsOn: assemble)
<< {
  println 'We release now'
}

build.taskGraph.whenReady {
  taskGraph ->
  if(taskGraph.
    hasTask(':release')) {
      version = '1.0'
    } else {
        version = '1.0-SNAPSHOT'
    }
}
```

Wykonaj skrypt budujący dwukrotnie - pierwszy raz wywołując komendę **`gradle clean assemble`** a następnie **`gradle clean release`**. Sprawdź w katalogu `build/libs` jakie nazwy otrzymały wynikowe pliki.

Tworzenie raportu z testów

Wyobraź sobie, że wykorzystujesz Gradle do umieszczenia bundli z testami integracyjnymi (napisanymi z użyciem TestNG) w środowisku OSGi, w którym działają moduły testowanej aplikacji. Modułów tych jest ponad 10, i ulegają one częstym zmianom (jako że wszystkie są jeszcze we wczesnym stadium developmentu). Chciałbyś wygenerować raport z testów, który zawierałby:

- dane na temat środowiska (wersja Javy, system operacyjny itp.),
- lista bundli zdeployowanych w

> Dużo lepszym rozwiązaniem jest stworzenie własnego taska i wywołanie go z odpowiednimi parametrami z poziomu skryptu budującego.

środowisku OSGi (wraz z numerem wersji każdego z bundli),

- wyniki testów.

Możliwe jest upchnięcie całej logiki tworzenia raportu bezpośrednio do skryptu budującego, ale wówczas ten urósłby nieprzyzwoicie i stałby się nieczytelny. Dużo lepszym rozwiązaniem jest stworzenie własnego taska i wywołanie go z odpowiednimi parametrami z poziomu skryptu budującego. Plik `build.gradle` wyglądałby w zarysie tak:

```
import org.gradle.sample.report.
ReportTask
...
dependencies {
  compile
  'commons-lang:commons-lang:2.4'
  testRuntime
  'org.easymock:easymock:2.5.2'
  // much more dependencies here
}
...
task report(type: ReportTask,
  dependsOn: itest) {
  reportDir = 'build/report'
  testNgResultsDir =
    'build/itest/test-result'
  serverLogDir=
    'build/itest/server-log'
  jars = configurations.testRuntime
}
```

A tak wygladałaby klasa tasku (z pominięciem nieistotnych szczegółów):

```
public class ReportTask extends
DefaultTask {

  def String reportDir
  def String testNgResultsDir
  def String serverLogDir
  def FileCollection jars

  @TaskAction
  def createReport() {
    def text = new StringBuilder()
    ["os.name", "os.version",
     "java.version",
     "java.vm.version",
     "java.runtime.version",
     "user.language",
     "user.name"].each {
       ext.append(it +
":\t${System.getProperty(it)}\n")
     }
    text.append("gradle -v".
      execute().text)
    jars.each {
      text.append("$it.name\n")
    }
  ...
    ant.zip(
      destfile: zipReportFile,
      basedir: tmpDir)
    }
    ...
}
```

Te dwa kawałki kodu demonstrują kilka godnych uwagi rzeczy. Po pierwsze, wszelkie typowe operacje na plikach (kopiowanie, tworzenie archiwum ZIP, usuwanie itd.) są bardzo łatwe do wykonania, gdy można użyć skryptów Groovyego (w tym klasy AntBuilder). Po drugie, stworzenie własnego, reużywalnego taska, a następnie skonfigurowanie go i wywołanie, jest bardzo proste. W tym przypadku task `ReportTask` przyjmuje cztery parametry, których dostarcza mu skrypt budujący (taki podział pozwala na wyrzucenie prawdziwej logiki biznesowej poza skrypt). Po trzecie w końcu, przykład ten pokazuje, że dependencje w Gradle można przetwarzać w najróżniejszy sposób - w powyższym przykładzie wykorzystujemy ten fakt do banalnego zadania wypisania ich do pliku raportu.

Budowa projektów wielomodułowych

Nasza aplikacja webowa potrzebuje kolejnego interfejsu użytkownika - napisze-

> Gradle oferuje dużą elastyczność,
> jeżeli chodzi o layout projektów wielomodułowych.

my go w Swingu. Jest to dobry moment, by podzielić projekt na trzy części - przy okazji prezentując fragment tego, co Gradle ma nam do zaoferowania w kontekście budowy projektów wielomodułowych:

- *core* - projekt zawierający logikę biznesową. Napisany w Javie, zależny od biblioteki Commons Lang. Oba projekty interfejsu użytkownika używają jego klas.
- *swing* - odpowiedzialny za interfejs użytkownika dla aplikacji desktopowej. Napisany w Groovym.
- *web* - tworzy webowy interfejs użytkownika. Napisany w JSP. Wynikowy WAR musi zawierać wszystkie wymagane w runtimie biblioteki - nie tylko JAR z projektu *core*, ale również jego dependencje (tj. bibliotekę Commons Lang).

Layout projektów wielomodułowych

Gradle oferuje dużą elastyczność, jeżeli chodzi o layout projektów wielomodułowych. Na potrzeby tego artykułu, zastosuję layout hierarchiczny (na schemacie pominąłem zawartość katalogów `src`):

```
.
|-- build.gradle
|-- settings.gradle
|-- core
|   `-- src
`-- ui
    |-- swing
    |   `-- src
    `-- web
        `-- src
```

W głównym katalogu znajdują się dwa podkatalogi - na projekt *core*, i drugi nazwany `ui`. W podkatalogu `ui` znajdują się wszystkie projekty związane z budową interfejsu użytkownika - obecnie są to dwa projektu *swing* i *web*.

Jak widać cały projekt wielomodułowy posiada tylko jeden skrypt budujący `build.gradle`. Można zastanawiać się, czy dobrym pomysłem jest zawarcie całej logiki budowania w jednym miejscu. Jeżeli uznamy, że nam to nie odpowiada, Gradle spełi naszą zachciankę i pozwoli na stworzenie oddzielnych skryptów budujących dla każdego podkatalogu.[3]

Jeden czy wiele skryptów budujących ?

Najważniejsze jest to, że można uzyskać ten sam efekt stosując oba podejścia. Dla przykładu, jeżeli chcemy dodać do projektu*core* dependencję w postaci biblioteki Commons Lang, to możemy albo dodać ten kawałek kodu do pliku `build.gradle` w głównym katalogu:

```
project (':core') {
  dependencies {
  compile "commons-lang:commons-lang:2.4"
  }
}
```

albo dodać tę linijkę do pliku `build.gradle` w podkatalogu `core`:

```
dependencies {
  compile "commons-lang:commons-lang:2.4"
}
```

Jak widzisz, jest to jedynie kwestia smaku.

> Zanim przyjrzymy się skryptowi budującemu, zastanówmy się jakie zadania powinien on wykonywać.

settings.gradle

Po piersze przyjrzyjmy się plikowi, którego dotąd jeszcze nie napotkaliśmy: `settings.gradle`. W nim właśnie znajdują się informacje o layoucie projektu[4]. Jak widać poniżej, wskazuje on Gradleowi w jakich podkatalogach znajdują się poszczególne moduły (podprojekty):

```
include "core", "ui:swing", "ui:web"
```

build.gradle

Zanim przyjrzymy się skryptowi budującemu, zastanówmy się jakie zadania powinien on wykonywać. Wydaje się, że są one następujace:

- projekt *core* - kompilacja klas Javy i budowa archiwum JAR,
- projekt *web* - kompilacja klas Javy i plików JSP, i budowa archiwum WAR,
- projekt *swing* - kompilacja klas Groovyego i budowa archiwum JAR,
- umieszczenie wszystkich stworzonych artefaktów (plików JAR i WAR) do wspólnego zasobu (podkatalogu `repo` w głowym katalogu projektu)

Plik budujący `build.gradle` przedstawiony jest poniżej.

```
subprojects {
  usePlugin 'java'
  group = 'org.gradle.sample'
  version = '1.0'

  repositories {
    mavenCentral()
    flatDir(name: 'fileRepo', dirs:
"./repo")
  }

  uploadArchives {
    repositories {
     add
      project.repositories.fileRepo
    }
  }

}

project (':core') {
  dependencies {
    compile "commons-lang:commons-
lang:2.4"
  }
}

project (':ui') {
  subprojects {
    dependencies {
     compile project (':core')
    }
  }
}

project (':ui:web') {
  usePlugin 'war'
}

project (':ui:swing') {
  usePlugin 'groovy'
  dependencies {
    groovy "org.codehaus.groovy-
:groovy-all:1.6.5"
  }
}
```

Nawet jeżeli pierwszy raz w życiu widzisz plik `build.gradle` opisujący budowę projektu wielomodułowego, nie powinieneś mieć problemów z jego zrozumieniem. Po pierwsze, korzystając z property `subprojects`, skrypt ustawia pewne własności wspólne dla wszystkich podprojektów (*core*, *swing* i *web*). W ten sposób wszystkie podprojekty będą:

> Gradle umożliwia wykonywanie częściowych buildów, a nam nie pozostaje nic innego jak z tej możliwości ochoczo korzystać

- używały pluginu Java,
- miały własność `group` ustawioną na `org.gradle.sample` a wersję na 1.0,
- szukały dependencji w centralnym repozytorium Mavena i w lokalnym katalogu `repo`,
- składowały stworzone artefakty w katalogu `repo`.

Po ustawieniach wspólnych dla wszystkich podprojektów, każdy z podprojektów konfigurowany jest osobno. Projekt *core* otrzymuje zależność od biblioteki Commons Lang. Oba projekty interfejsu użytkownika są zależne od projektu *core*. Następnie do projektu *web*dodany jest plugin War. W końcu, projekt *swing* zostaje skonfigurowany jako projekt Groovyego (co sprowadza się do dodania do niego pluginu Groovy i dopisania zależności od biblioteki Groovyego).

Budowa projektu - pełna i częściowa

W przypadku projektu wielomodułowego pojawia się pytanie “ale co właściwie chciałbyś zbudować?”. A więc, chciałbyś zbudować oba projekty interfejsu użytkownika, czy może tylko jeden z nich?

Aby zbudować wszystkie projekty, wykonaj komendę **`gradle build`** z poziomu głównego katalogu projektu. Zauważysz, że w procesie budowania powstaną trzy artefakty: `core-0.1.jar`, `swing-0.1.jar` i `web-0.1.war`.

Jeżeli chcemy wybudować wyłącznie jeden z interfejsów użytkownika, możemy użyć tasku `buildNeeded`[5]. Na przykład, wykonanie polecenia **`gradle buildNeeded`** w podkatalogu `ui/swing`, poskutkuje zbudowaniem wyłącznie projektów *core* i *swing* oraz stworzeniem dwóch artefaktów: `core-0.1.jar` i `swing-0.1.jar`.

Wniosek z tego wszystkiego taki, że Gradle umożliwia wykonywanie częściowych buildów, a nam nie pozostaje nic innego jak z tej możliwości ochoczo korzystać (w końcu funkcjonalność ta pozwala znacząco skrócić czas budowy projektu).

Podsumowanie

...ah, czyżby kolejny projekt-zbawca ? Kolejne narzędzie, które obiecuje rozwiązanie wszelkich problemów związanych z budowaniem projektów ? Nie całkiem. Gradle nie obiecuje aż tak wiele. Za to próbuje

uczynić niemożliwe możliwym, możliwe łatwym, a łatwe eleganckim

Moshé Feldenkrais

I nawet już teraz, przed wydaniem wersji 1.0, Gradle oferuje mnóstwo interesujących i przydatnych możliwości. W tym artykule zaprezentowałem kilka z nich, i nie ukrywam, że jest to jedynie wierzchołek (wciąż rosnącej) góry lodowej. :)

Chciałem zachęcić cię do zainwestowania odrobiny czasu by lepiej poznać Gradle. Początkowo jego elastyczność i brak sztywnych reguł może sprawić, że

> Przekonasz się że w tym projekcie tkwi potęga, o jakiej nawet nie marzyłeś.

będziesz czuł się nieco nieswojo (szczególnie jeżeli jesteś doświadczonym użytkownikiem Mavena). W moim przypadku, nawet po kilku miesiącach pracy z Gradle, mam poczucie, że wciąż nie dokonałem jeszcze całkowitego "przeskoku myślowego". Wciąż też zadziwia mnie łatwość i elastyczność, z jaką Gradle pozwala mi tworzyć skrypty budujące projekt. Musisz dać sobie nieco czasu, a przekonasz się że w tym projekcie tkwi potęga, o jakiej nawet nie marzyłeś. Koszt przejścia (z Anta lub Mavena) jest minimalizowany przez to, że Gradle wykorzystuje wiele rozwiązań, które są Ci dobrze znane z innych projektów (np. standardowy layout projektu, taski Anta, repozytoria Mavena itd.).

Na stronie Gradle znajdziesz bardzo rozbudowany podręcznik użytkownika, a w kodzie źródłowym mnóstwo przykładów, które pomogą Ci rozpocząć pracę z Gradle (zajrzyj też do CookBook po różnorakie pomocne wskazówki). Możesz też liczyć na wsparcie członków społeczności (dostępna jest lista mailingowa i wiki), a w przypadku gdybyś potrzebował profesjonalnych szkoleń, zapoznaj się z ofertą na stronie Gradle Inc.

Linki

- http://gradle.org – strona projektu Gradle
- http://maven.apache.org/plugins/maven-antrun-plugin/ - Maven AntRun Plugin
- http://ant.apache.org - Ant
- http://maven.apache.org - Maven
- http://groovy.codehaus.org/Using+Ant+from+Groovy - użycie Anta ze skryptów Groovy

[1] Dla ścisłości dodam, że tak naprawdę to pluginy Java i War narzucają taki układ projektu jako domyślny.

[2] Maven 3 pozwala na użycie równie zwięzłej składni, ale jak zobaczymy nieco później, w przypadku dependencji, Gradle oferuje o wiele więcej.

[3] W załączonym do artykułu kodzie źródłowym znajdziesz obie wersje - sam oceń, która wydaje Ci się bardziej odpowiednia.

[4] Plik `settings.gradle` może też służyć innym celom.

[5] Inne taski o zbliżonym działaniu opisane są w podręczniku użytkownika Gradle.

Express Killers, cz. V

Damian Szczepanik

Przykład pierwszy

W dzisiejszym cyklu zaczniemy od krótkiego kodu dla spostrzegawczych. Pytanie brzmi: zakładając, że poniższa klasa się kompiluje, a jej uruchomienie nie spowoduje rzucenie wyjątkiem, jakiej klasy jest referencja `obj`?

```
public class ClassType
{
  // definicja typu obj

  public static void main(String[]
args)
  {
    Object pattern = ClassType.
class;
    for (int i = 0; i < obj.
length(); i++)
    {
      if (pattern.equals(obj))
        obj += 5;
      else
        obj += -5;
      System.err.println(obj);
    }
  }
}
```

Jeśli zadanie jest zbyt trudne, skopiuj klasę do swojego IDE. Pomoże Ci to znaleźć rozwiązanie.

Przykld drugi

W drugiej części (już bez pomocy IDE!) przypomnijmy sobie, czym grozi brak enkapsulacji oraz przesłanianie zmiennych. Co będzie wynikiem uruchomienia poniższej metody main()?

```
public class AccessingToOverride-
nAttribute
{
  public int a;

  public void add()
  {
    a = 10;
  }

  public static void main(String[]
args)
  {
    AccessingToOverridenAttribute
f = new Bar();
    f.add();
    System.err.println(++f.a);
    System.err.println(++((Bar)
f).a);
  }
}


class Bar extends AccessingToOver-
ridenAttribute
{
  public int a = 2;

  public void add()
  {
    a = 5;
  }
}
```

Express Killers, cz. V - odpowiedzi

Damian Szczepanik

Przykład pierwszy:

Kluczem do zagadki jest metoda length() i operacje dodawania. Po pierwsze obj nie może być tablicą, gdyż w pętli użyto funkcji length(), a nie length. Oznacza to, że obj jest referencją do klasy, która implementuje metodę length(). Żaden wrapper typów prymitywnych jak Integer czy Double nie implementuje tej metody. Ma ją natomiast String. Czy jest możliwe, by była to inna klasa, którą użytkownik sam zaimplementował? Metoda length() nie stanowiłaby problemu, gdyż można ją zaimplementować. Natomiast operator dodawania jest dozwolony tylko dla typów prymitywnych, wraperów oraz wspomnianej klasy String. Java nie udostępnia mechanizmów przeciążania operatorów, jak to ma miejsce np w języku C++. Klasa String posiada modyfikator final, zatem nie można jej także rozszerzyć. Wobec powyższego jedyną możliwością, by powyższy kod się kompilował jest, by obiekt obj był referencją do klasy String. I nie był wartością null, gdyż to spowoduje wygenerowanie wyjątku podczas uruchomienia.

Przykład drugi:

Po pierwsze metoda add() zostanie wywołana zależnie od typu obiektu f, a nie deklaracji tej zmiennej. Zatem sterowanie zostanie przekazane do klasy Bar. Oznacza to, że to wartość atrybutu a klasy bazowej nie ulegnie zmianie.

Po drugie użycie wartości f.a spowoduje wydrukowanie atrybutu wskazanego typem zmiennej niezależnie od tego, na co wskaże referencja. Zostanie to określone na etapie kompilacji i nie ma znaczenia, że klasa pochodna deklaruje atrybut o tej samej zmiennej. Nie warto zatem deklarować zmiennej o takiej samej nazwie, jak w klasie nadrzędnej i nie należy pozostawiać dostępu do tej zmiennej w sposób, który umożliwi użycie konstrukcji, jak we wspomnianym przykładzie.

Po trzecie wydrukowanie zmiennej poprzez wywołanie ((Bar) f).a zadziała wg powyższej zasady: kompilator na etapie kompilacji wskaże atrybut klasy pochodnej, gdyż jawnie tego oczekiwano poprzez rzutowanie.

Wnioski: atrybuty powinny być niewidoczne poza klasą i dostępne poprzez metody get i set, co skutecznie minimalizuje opisane powyżej problemy oraz ułatwia debugowanie.

Recenzja: GroovyMag 11/2009

Krzysztof Konwisarz

Groovy Under the Hood – Groovy Collective Types - Ranges

Autor: Kirsten Schwark

Zakresy to jedna z konstrukcji ułatwiająca programowanie w Groovy, nieobecna w Javie. Przydają się one choćby w pętlach for i w instrukcjach switch. W obszernym artykule można przeczytać o ich implementacji, rodzajach i charakterystykach. Nie brakuje też przykładowych zastosowań.

Groovy around the Globe – Australia & New Zealand

Autor: Steve Dalton

Debiutujący cykl artykułów ma za zadanie przybliżać nam środowiska Groovy i Grails z całego świata, zaczynając od Australii i Nowej Zelandii. Znajdziemy tu informacje o konferencjach i grupach użytkowników na kontynencie, poznamy też nazwiska i osiągnięcia ludzi z tamtych okolic zajmujących się Groovy i Grails.

Enterprise Development with Groovy and Grails

Autor: Jason Warner

Jason Warner przedstawia historię niewielkiej firmy migrującej z ColdFusion na Groovy i Grails. W żadnym miejscu nie ukrywając wad Grails pisze o powodach, wyzwaniach i korzyściach wynikających z takiego kroku.

Grails and Maven – Differing Opinions, Same Goal

Autor: Michael Wall

Istnieją sytuacje, w których Maven przydaje się przy pracy z aplikacją Grails. Okazuje się, że nic nie stoi na przeszkodzie, aby przy użyciu grails-maven-plugin dodać go do nowego lub istniejącego projektu i zarządzać nim przy pomocy mavenowskich ekwiwalentów poleceń Grails.

Plugin Corner – Bean-Fields Plugin

Strony GSP czasami robią się mało czytelne. Powyższa wtyczka pozwala skrócić ilość kodu potrzebną do wyświetlenia choćby pola formularza. Znaczniki HTML, miejsce wstawienia etykiety oraz samego pola zawarte są w szablonie, a my w naszym widoku w ich miejsce umieszczamy jedną linijkę. Poza istniejącymi szablonami mamy możliwość tworzenia własnych.

O autorze

Student Informatyki na Wydziale Fizyki Uniwersytetu im. Adama Mickiewicza w Poznaniu.

Kod promocyjny

Dla czytelników JAVA exPress przygotowaliśmy specjalny kod promocyjny „**poland1**”, dzięki któremu możecie pobrać jeden dowolny numer GroovyMag. Zapraszamy do zapoznania się z tym czasopismem.

Mistrz Programowania: Refaktoryzacja, cz. II

Mariusz Sieraczkiewicz

Dzisiaj kolejny odcinek książki-niespodzianki o refaktoryzacji. Mariusz Sieraczkiewicz zgodził się opublikować w odcinkach na łamach JAVA exPress swoją książkę "Jak całkowicie odmienić sposób programowania używając refaktoryzacji". Jest to pierwsza książka z serii Mistrz Programowania i dotyczy... no tak - refaktoryzacji.

Pierwsza część książki jest dostępna za darmo na stronie http://www.mistrzprogramowania.pl/. Tam także możesz zakupić pełną wersję, bez konieczności czekania 3 miesięcy na kolejną część w JAVA exPress. No i będziesz miał całość w jednym pdf-ie.

W każdym razie zapraszam nawet jeśli możesz czekać. Wspomagajmy samych siebie. Może jutro Ty będziesz chciał coś sprzedać...

A książka Mariusza jest warta swej ceny ;)

Grzegorz Duda

Refaktoryzacja: Zmiana algorytmu na pisany ludzkim językiem

Algorytm poszukiwania słowa można zapisać z użyciem następujących kroków:

1. inicjacja zmiennych i pobranie zawartości strony z tłumaczeniem
2. dla każdego znalezionego słowa będącego częścią tłumaczenia
   (a) znajdź polski odpowiednik
   (b) znajdź angielski odpowiednik
   (c) zapamiętaj tłumaczenie
   (d) wypisz tłumaczenie na ekranie

Powyższy zapis algorytmu odpowiada ludzkiemu rozumowaniu, zaś obecna wersja kodu jest zapisem programistycznego myślenia. Spróbujmy zatem zrefaktoryzować kod, aby jego struktura odzwierciedlała przebieg opisany powyżej. Pierwszy krok, który zrobimy w tym kierunku, to wydzielenie metody odpowiedzialnej za część *inicjacja zmiennych i pobranie zawartości strony z tłumaczeniem*. Wynikiem tej metody będzie strumień `BufferedReader` reprezentujący analizowaną stronę HTML. Strumień jest obiektem potrzebnym w całym algorytmie, zatem będzie polem w klasie `SearchWordService`. Kod po refaktoryzacji wygląda następująco:

```
package pl.bnsit.webdictionary;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class SearchWordService {

    private String command = null;
    private BufferedReader bufferedReader = null;

    public SearchWordService(String command) {
```

```
        this.command = command;
    }

    public List<DictionaryWord> search() {
        List<DictionaryWord> result = new ArrayList<DictionaryWord>();

        String polishWord = null;
        String englishWord = null;
        int counter = 1;

        try {
            bufferedReader = prepareBufferedReader();

            boolean polish = true;
            String line = bufferedReader.readLine();

            // .. dalej kod bez zmian

        return result;
    }

    private BufferedReader prepareBufferedReader() throws IOException,
            MalformedURLException {
        String[] commandParts = command.split(" ");
        String wordToFind = commandParts[1];

        String urlString = "http://www.dict.pl/dict?word="
                + wordToFind + "&words=&lang=PL";

        return new BufferedReader(new InputStreamReader(
                new URL( urlString).openStream()));
    }

    private boolean hasNextLine(String line) {
        return (line != null);
    }
}
```

W ten sposób złożona metoda `search` uprościła się nieco poprzez wydzielenie metody składowej `prepareBufferedReader`. Metoda ta zwraca jako wynik stworzony strumień do odczytu zawartości strony, jednocześnie strumień ten jest polem klasy. Może zatem paść propozycja, aby metoda ta nie zwracała żadnego wyniku, tylko ustawiała to pole. Jest to opcja poprawna, natomiast ja osobiście preferuję zwracanie wyniku, gdyż dzięki temu uzyskujemy jawną informację o efekcie działania metody. Przyjrzyjmy się wersji z niejawnym ustawieniem pola `bufferedReader`.

```
// ...
public List<DictionaryWord> search() {
    List<DictionaryWord> result = new ArrayList<DictionaryWord>();

    String polishWord = null;
    String englishWord = null;
    int counter = 1;

    try {
        init();
        // ...
}

private void init() throws IOException,
        MalformedURLException {
    String[] commandParts = command.split(" ");
    String wordToFind = commandParts[1];

    String urlString = "http://www.dict.pl/dict?word="
            + wordToFind + "&words=&lang=PL";

    bufferedReader = new BufferedReader(new InputStreamReader(
            new URL( urlString).openStream()));
}
```

Wywołanie `init()` nie daje żadnych informacji o tym, że w ciele tej metody następuje zmiana stanu, co wyraźnie widać w poprzednim przykładzie `bufferedReader = prepareBufferedReader()`. Jest do dosłowny zapis celu działania metody, którym jest przygotowanie obiektu typu `BufferedReader`, a ten z kolei zostanie zapamiętany jako pole w klasie.

Zajmiemy się główną częścią algorytmu, która teraz przedstawia się obecnie następująco:

```
// ...

boolean polish = true;
String line = bufferedReader.readLine();
while (hasNextLine(line)) {

    Pattern pat = Pattern
            .compile(".*<a href=\"dict\\?words?=(.*)&lang.*");

    Matcher matcher = pat.matcher(line);
    if (matcher.find()) {
        String foundWord
            = matcher.group(matcher.groupCount());
```

```
            if (polish) {
                System.out.print(counter + ") "
                        + foundWord + " => ");
                polishWord
                    = new String(foundWord.getBytes(), "UTF8");
                polish = false;
            } else {
                System.out.println(foundWord);
                polish = true;

                englishWord
                    = new String(foundWord.getBytes(), "UTF8");

                result.add(new DictionaryWord(polishWord,
                        englishWord, new Date()));
                counter++;
            }
        }

        line = bufferedReader.readLine();
    }
} catch (MalformedURLException ex) {

// ...
```

Taki zapis jest bardzo zawiły, jest sporo zmiennych tymczasowych: `polish` — do przechowywania informacji o bieżącej wersji językowej, `polishWord`, `englishWord` — zmienne przechowujące tymczasowo wartości wersji polskiej i angielskiej słowa.

Zatrzymajmy się na chwilę. Jeśli przyjrzymy się jeszcze raz strukturze strony HTML, to zauważymy, że słowo polskie i angielskie podlega analizie w identyczny sposób. Słowa te występują naprzemiennie — najpierw słowo polskie, później słowo angielskie. Dlaczegóżby zatem nie uprościć nieco algorytmu? Wydzielmy metodę, która będzie potrafiła znaleźć kolejny wyraz (polski lub angielski). Użyjemy jej do naprzemiennego znajdowania wyrazów polskiego i angielskiego.

Wyszukiwanie kolejnego słowa oprzemy na istniejącym już algorytmie z powyższego kodu źródłowego. Wydzieloną metodę nazwijmy `moveToNextWord`, będzie zwracać kolejny znaczący wyraz polski lub angielski znajdujący się na stronie HTML lub `null`, jeśli wszystkie słowa zostały odnalezione. Głównym celem tej metody jest stworzenie mechanizmu do odnajdywania kolejnych znalezionych wyrazów na stronie HTML.

```
private String moveToNextWord() {
    try {
        String line = bufferedReader.readLine();
```

```
        while (hasNextLine(line)) {

            Pattern pattern = Pattern
                    .compile(".*<a href=\"dict\\?words?=(.*)&lang.*");

            Matcher matcher = pattern.matcher(line);

            if (matcher.find()) {
                String foundWord = matcher.group(matcher.groupCount());

                return new String(foundWord.getBytes(), "UTF8");
            } else {
                line = bufferedReader.readLine();
            }
        }
    } catch (IOException e) {
        // TODO obsluzyc blad
    }

    return null;
}
```

Refaktoryzacja: Wprowadzenie klarownej obsługi wyjątków

Na chwilę chciałbym się zatrzymać nad obsługą wyjątków. Do tej pory nie poświęcaliśmy temu tematowi zbyt wiele czasu. Odruchowo stosowaliśmy schemat:

```
        // ...

        } catch (IOException ex) {
            ex.printStackTrace();

        }

        // ...
```

Dławienie wyjątków

W przypadku wystąpienia sytuacji wyjątkowej, program wprawdzie wypisze informacje o stosie wywołań w momencie wystąpienia wyjątku, jednak nie jest w żaden sposób przygotowany na jego obsługę. Powyższe rozwiązanie to nieco lepsza odmiana techniki

zwanej dławieniem wyjątków, która w 99% przypadków jest niedopuszczalna. Wygląda ona mniej więcej tak:

```
// ...

} catch (IOException ex) {
    // nic nie robie
}

// ...
```

Takie posunięcie spowoduje, że w momencie wystąpienia błędu nic się nie stanie z aplikacją, a nie jest to pożądany efekt. Błąd będzie niezauważony. Na sytuację wyjątkową w jakiś sposób należy zareagować, być może przedstawić jakiś komunikat użytkownikowi, być może dokonać próby wznowienia operacji. Coś należy zrobić.

Ważne

Jeśli wystąpi wyjątek, nie należy udawać, że nic się nie stało. Należy zareagować na sytuację wyjątkową. Inaczej w systemie zginie ważna informacja, która jest trudna do odtworzenia w przypadku analizy lub naprawy systemu.

Jest kilka możliwości reakcji.

1. Zareagować natychmiast w bloku `catch`.
   Jeśli tylko jesteś w stanie sensownie zareagować w miejscu wystąpienia wyjątku, zrób to.

2. Przerzucić wyjątek do metody wywołującej (`throws` w sygnaturze).
   Jeśli nie jesteś w stanie obsłużyć wyjątku lub z pewnego powodu nie chcesz tego zrobić, możesz przerzucić wyjątek do metody wywołującej daną metodę; opcja ta może mieć sens przy stosowaniu refaktoryzacji *Wydzielenie metody*.

3. Opakować wyjątek lub wygenerować własny wyjątek kontrolowany (ang. checked) dziedziczący po `java.lang.Exception`.
   Jeśli wyjątek powinien mieć wpływ na inną część systemu (np. na interfejs użytkownika) i ta część systemu powinna być przygotowana na jego obsługę, rzuć własny wyjątek lub opakuj nim wyjątek źródłowy.

4. Opakować wyjątek lub wygenerować własny wyjątek niekontrolowany (ang. unchecked) dziedziczący po `java.lang.RuntimeException`.

Jeśli wyjątek jest błędem, na który system w sposób bezpośredni nie będzie reagować lub nie jest w stanie reagować, rzuć wyjątek niekontrolowany lub opakuj nim wyjątek źródłowy.

Wróćmy do przykładu. W metodzie `moveToNextWord` wyjątek wystąpi wtedy, kiedy pojawią się problemy podczas pracy ze strumieniem. Jest to sytuacja, co do której nie przewidujemy bezpośredniej obsługi w naszym przypadku, dlatego zastosujemy czwartą opcję obsługi wyjątku — stworzymy własną klasę wyjątku o nazwie `WebDictionaryException` dziedziczącą z `java.lang.RuntimeException`. W przypadku gdy błąd wystąpi, program zostanie przerwany. Jeśli chcemy reagować na tę sytuację po stronie interfejsu użytkownia, w metodzie `WebDictionary.main`, `WebDictionary.processMenu` lub `WebDictionary.searchWord` możemy dodać obsługę tego wyjątku. Przypomnę tylko, iż obsługa wyjątków dziedziczących po `RuntimeException` nie jest wymuszana przez kompilator — nie ma konieczności ich deklaracji w `throws`.

Klasa wyjątku będzie wyglądać następująco:

```
package pl.bnsit.webdictionary;

public class WebDictionaryException extends RuntimeException {

    public WebDictionaryException() {
    }

    public WebDictionaryException(String arg0) {
        super(arg0);
    }

    public WebDictionaryException(Throwable arg0) {
        super(arg0);
    }

    public WebDictionaryException(String arg0, Throwable arg1) {
        super(arg0, arg1);
    }

}
```

Zaś obsługa wyjątku wygląda następująco:

```
        } catch (IOException e) {
            throw new WebDictionaryException(e);
        }
```

Niby nic wielkiego się nie stało, jednak wyjątek został odpowiednio przygotowany — jeśli zdarzy się w systemie coś niepożądanego, na pewno pojawi się o tym odpowiednia informacja.

Refaktoryzacja: Zmiana nazwy metody

Wróćmy do naszej metody `moveToNextWord`. Teraz możemy ją w pełni wykorzystać, aby uprościć algorytm.

```
public List<DictionaryWord> search() {
    List<DictionaryWord> result = new ArrayList<DictionaryWord>();

    try {
        bufferedReader = prepareBufferedReader();
        String currentWord = moveToNextWord();
        int counter = 1;

        while (hasNextWord(currentWord)) {
    // ...

private boolean hasNextWord(String word) {
    return (word != null);
}
```

W ten sposób ustawiamy się na pierwszym słowie znalezionym wśród wierszy strony HTML. Zauważmy, że dzięki takiemu zabiegowi, przenosimy nasz algorytm na inny poziom abstrakcji. Dzięki wydzieleniu metody `moveToNextWord` nie musimy już funkcjonować na niskim poziomie pojedynczych wierszy HTML, a poruszamy się po kolejnych słowach, które gdzieś na tej stronie się znajdują. Jest to ogromna zmiana jakościowa, która znacząco upraszcza konstrukcję algorytmu. Spójrzmy jeszcze na argument pętli `while`.

```
        while (hasNextWord(currentWord)) {
```

Ponieważ głównym elementem napędowym algorytmu są teraz słowa a nie wiersze, toteż dokonaliśmy bardzo prostej, ale jakże potężnej w skutkach refaktoryzacji *Zmiana nazwy metody*. Poprzednia nazwa (`hasNextLine`) po wprowadzonych zmianach nie odzwierciedla już odpowiedzialności metody, dlatego wprowadziliśmy nową nazwę `hasNextWord`.

Dalsza część algorytmu jest bardzo prosta — w pętli wyszukujemy polskie i angielskie słowo umieszczając je w obiekcie klasy `DictionaryWord`, który to obiekt jest ostatecznie dodawany do wynikowej listy i wypisywany na konsoli.

```
// ...
while (hasNextWord(currentWord)) {
    DictionaryWord dictionaryWord = new DictionaryWord();
    dictionaryWord.setPolishWord( currentWord );
    currentWord = moveToNextWord();
    dictionaryWord.setEnglishWord( currentWord );
    currentWord = moveToNextWord();

    result.add( dictionaryWord );

    System.out.println( counter + ") "
            + dictionaryWord.getPolishWord()
            + " => " + dictionaryWord.getEnglishWord());
    counter = counter + 1;
}
// ...
```

Zapis ten upraszcza algorytm, który możemy niemalże natychmiast odczytać z powyższego kodu.

Dopóki jest dostępne słowo:

- stwórz nowy obiekt klasy `DictionaryWord`,
- zapisz bieżące słowo jako polską wersję słowa,
- przejdź do kolejnego słowa,
- ustaw bieżące słowo jako angielską wersję słowa,
- przejdź do kolejnego słowa,
- do listy dodaj nowe tłumaczenie (obiekt klasy `DictionaryWord`),
- wyświetl tłumaczenie na ekranie,
- zwiększ licznik tłumaczeń.

Kod czyta się niczym książkę. Kod całej zrefaktoryzowanej klasy `SearchWordService` znajduje się poniżej. Porównaj ten kod z kodem początkowym. Nie jest on kodem krótszym, ale jest kodem zdecydowanie prostszym. Bez większego wysiłku można go przeanalizować i wywnioskować, o co chodzi.

```
package pl.bnsit.webdictionary;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.ArrayList;
import java.util.List;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class SearchWordService {

    private String command = null;
    private BufferedReader bufferedReader = null;

    public SearchWordService(String command) {
        this.command = command;
    }

    public List<DictionaryWord> search() {
        List<DictionaryWord> result = new ArrayList<DictionaryWord>();

        try {
            bufferedReader = prepareBufferedReader();
            String currentWord = moveToNextWord();
            int counter = 1;

            while (hasNextWord(currentWord)) {
                DictionaryWord dictionaryWord = new DictionaryWord();
                dictionaryWord.setPolishWord( currentWord );
                currentWord = moveToNextWord();
                dictionaryWord.setEnglishWord( currentWord );
                currentWord = moveToNextWord();

                result.add( dictionaryWord );

                System.out.println( counter + ") "
                        + dictionaryWord.getPolishWord()
                        + " => " + dictionaryWord.getEnglishWord());
                counter = counter + 1;
            }
        } catch (MalformedURLException ex) {
            ex.printStackTrace();

        } catch (IOException ex) {
            ex.printStackTrace();
```

```
        } finally {
            try {
                if (bufferedReader != null ) {
                    bufferedReader.close();
                }
            } catch (IOException ex) {
                ex.printStackTrace();
            }
        }

        return result;
    }

    private String moveToNextWord() {
        try {
            String line = bufferedReader.readLine();
            Pattern pattern = Pattern
                .compile(".*<a href=\"dict\\?words?=(.*)&lang.*");

            while (hasNextWord(line)) {

                Matcher matcher = pattern.matcher(line);

                if (matcher.find()) {
                    String foundWord = matcher.group(matcher.groupCount());

                    return new String(foundWord.getBytes(), "UTF8");
                } else {
                    line = bufferedReader.readLine();
                }
            }
        } catch (IOException e) {
            throw new WebDictionaryException(e);
        }

        return null;
    }

    private BufferedReader prepareBufferedReader() throws IOException,
            MalformedURLException {
        String[] commandParts = command.split(" ");
        String wordToFind = commandParts[1];

        String urlString = "http://www.dict.pl/dict?word="
                + wordToFind + "&words=&lang=PL";

        return new BufferedReader(new InputStreamReader(
                new URL( urlString).openStream()));
    }
```

```
    private boolean hasNextWord(String word) {
        return (word != null);
    }
}
```

Kod, który stworzyliśmy jest dużo czytelniejszy. Jest już całkiem niezły i do zaakceptowania. W tym momencie warto zrobić sobie chwilę przerwy, aby celebrować osiągnięcie.

Ważne

Każde zwycięstwo należy celebrować. Tylko wtedy jesteśmy w stanie w pełni doświadczyć zwycięstwa.