Numer 3/2009(5)

Java eXPress

CZASOPISMO DLA DEWELOPERÓW JAVA

OSGi: Modularność bez restartów w aplikacjach J2EE

Flex i Java

SCJP w pigułce

Lista zadań w Grails

BONUS: Refaktoryzacja cz. I

>> Automatyczne generowanie kodu

>> Recenzja GroovyMag 07/2009

>> Express killers cz. IV

nextIssue.wait()

To były ciężkie dni. Od ostatniego wydania JAVA exPress sporo się wydarzyło. Niestety miało to odbicie w dosyć dużym opóźnieniu wydania tego numeru. Niemniej jednak mam nadzieję, że czekać było warto. A co wpłynęło na to opóźnienie?

Po pierwsze, wydaliśmy w „międzyczasie" angielską wersję czwartego numeru JAVA exPress. Nie było łatwo, bo tłumaczenie tekstów oraz korekta to jedno, a skład tekstu to drugie. Tutaj z pomocą przyszło Adobe i numer angielski jako pierwszy powstał przy użyciu Adobe InDesign. Z jednej strony koniec kłopotów ze Scribusem, a z drugiej konieczność stworzenia wszystkich szablonów od podstaw. Daliśmy radę ;) Wielkie dzięki na ręce tłumaczy z Pawłem Cegłą i Magdaleną Rudny na czele, wraz z Łukaszem Baranem i Bogusławem Osuchem.

Po trzecie, nowe strony www zarówno JAVA exPress, jak i dworld.pl. To co najfajniejsze, to Blogsfera na dworld.pl. W jednym miejscu wszystkie polskie blogi o Javie ze śledzeniem oglądalności i atrakcyjności wpisów. Blogsfera została dobrze przyjęta przez Was, z zastrzeżeniami co do wyglądu, ale będziemy nad tym pracowali. Release 1.0 miał dostarczyć funkcjonalność, a nie bajery graficzne. A to nie koniec ciekawych rozwiązań na dworld.pl. Warto odwiedzać nas co jakiś czas. Dzieki Blogsferze niektore blogi zwiekszyly ruch o ponad 50%. Więc efekty są ;) Za strony wielkie dzięki Markowi Podsiadłemu i Jakubowi Sosińskiemu (kontakt@vriltek.com). Bez nich stron by nie było.

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

Do zobaczenia w grudniu,
Grzegorz Duda

Schedule

Redaktor: Grzegorz Duda
Korekta: Agnieszka Wadowska-Duda
Strony www i teksty html: Marek Podsiadły
Grafika: Jakub Sosiński
Tłumaczenie: Paweł Cegła, Magdalena Rudny, Łukasz Baran, Bogusław Osuch

Na horyzoncie...

Grzegorz Duda

JDD

Już 16 października odbędzie się kolejna edycja największej konferencji o Javie w Polsce - JDD. Tym razem z zagranicznych gwiazd warto wymienić Marka Richardsa oraz Scotta Davisa. Obydwaj Panowie są wybitnymi prelegentami światowych konferencji, jak również regularnie występują na konferencjach NFJS.

Więcej informacji na stronie http://09.jdd.org.pl

Autorzy JAVA exPress mogą wygrać wejściówkę na aukcjach dworld.pl.

COOLuary

Dzień po JDD, czyli w sobotę 17 października odbędzie się 3 edycja COOLuarów. Jedynej w Polsce UnConference o Javie. Tym razem naszymi gośćmi będą gwiazdy JDD, czyli Mark i Scott. Koszt uczestnictwa to jedyne 120 zł, a w tym kawa, herbata, ciastka, obiad oraz jak zwykle koszulki, i gadżety Javowe i możliwość wygrania wejściówek na konferencje oraz książek.

Więcej informacji znajdziecie na stronach http://dworld.pl

Blogsfera

Ruszyły nowe strony http://dworld.pl a na nich Blogsfera, czyli agregator blogów javowych w Polsce wraz ze śledzeniem popularności i atrakcyjności wpisu. Już wkrótce kolejne atrakcje.

Pomóż JAVA exPress i dWorld

Jak zwykle apel o Waszą pomoc. Zainteresowanych proszę o mail na kontakt@dworld.pl oraz odwiedzenie działu Współpraca na http://www.javaexpress.pl/

RuPy

W dniach 7-8 listopada odbędzie się w Poznaniu konferencja RuPy (http://rupy.eu) poświęcona dynamicznym językom. Ludzie związani z Javą zapewne zwrócą uwagę na Griffon, JRuby, czy Git. Koszt uczestnictwa niewielki, a więc warto skorzystać.

Devoxx

Devoxx w tym roku nieco wcześniej, więc może będzie nieco cieplej w Antwerpii. Od 16 - 20 listopada Belgia stanie się mekką javowców. Patrząc na niepewną przyszłość JavaOne, może się okazać, że Devoxx przejmie pałeczkę największej konferencji Javowej na świecie. Niestety cena w tym roku nieco wyższa, ale wciąż patrząc na zawartość i atmosferę konferencji można wybrać się w ciemno.

Wygrana w Java Guide

Jeszcze przed uruchomieniem nowych stron dworld.pl nasz serwis wygrał w konkursie Java Guide (44% głosów) organizowanym przez JDD. Serdeczne dzięki za docenienie. Mam nadzieję, że nowa odsłona dworld.pl podoba się Wam jeszcze bardziej.

BLENDED FOR THE JAVA COMMUNITY

DEVOXX™

the java™ community conference

SCJP w pigułce

Michał „Chlebik” Piotrowski

Chyba każdy programista Java słyszał o certyfikacie SCJP. Wielu mniej go zdawało, a i oni pewnie mają do powiedzenia tylko tyle, że dobrze, iż mają to za sobą. W poniższym artykule zamierzam zaprezentować podsumowanie mojej lektury podręcznika przygotowującego do zdania tego egzaminu.

Słowem wstępu

Egzamin SCJP (Sun Certified Java Programmer) jest jednym z najlepiej rozpoznawalnych egzaminów certyfikacyjnych oferowanych przez firmę SUN. Nie tylko jako pierwszy stopień do zdawania kolejnych, ale również jako potwierdzenie znajomości Javy na dość wysokim poziomie. Dla osób, które pragną zawodowo zajmować się tworzeniem aplikacji w języku Java jest on zatem bardzo cennym atrybutem, cenniejszym tym bardziej, iż uznawanym na całym świecie.

Swoją przygodę z nauką pod kątem zdawania tegoż egzaminu zacząłem jakiś czas temu. Jako człowiek, który nie ukończył kierunku w minimalnym stopniu związanego z informatyką – egzamin ten miał być potwierdzeniem własnej wiedzy oraz umiejętności programistycznych. Moje przemyślenia z lektury kolejnych rozdziałów podręcznika autorstwa duetu Bates & Sierra publikowałem w formie cyklicznych wpisów na swoim blogu (http://www.chlebik.wordpress.com). Jednakże pomimo kategoryzacji oraz przejrzystości wpisów – brakowało jednolitej syntezy wszystkich zapisanych wskazówek. Co więcej, szereg testowych egzaminów dostępnych w internecie również miał wpływ na moją wiedzę, a tym samym zaowocowało to rozbudowaniem niektórych porad. Dzięki uprzejmości redaktorów Java exPress mam możliwość przedstawienia całości w formie ujednoliconej oraz bardziej rozbudowanej, niż miało to miejsce w formie periodycznych wpisów na blogu.

Poniższy artykuł to spisane zagadnienia, które sprawiły mi problemy (lub wzbudziły wątpliwości) podczas rozwiązywania testów zamieszczonych na końcu każdego z rozdziałów podręcznika, a także testów próbnych (tzw. *mock exams*). Wskazówki są momentami dość szczegółowe, ale to przecież nie wada – zawsze bowiem lepiej jest wiedzieć więcej, niż mniej. Całość podzieliłem na 10 części (tyle, ile jest rozdziałów w książce), a także na kilka rad ogólnych. Autorzy podręcznika mają bardzo miłą tendencję do wplatania w treść rozdziałów zabawnych wypowiedzi czy aforyzmów. Nie służy to tylko zabawie – jest to doskonała mnemotechnika, którą warto sobie przyswoić przy nauce czegokolwiek. Każdą część okrasiłem zatem tego typu cytatem (w oryginale), co mam nadzieję uprzyjemni lekturę. Do dzieła!

Porady ogólne:

Należy zawsze zwracać uwagę na poprawność każdego kawałka kodu! Wiem, że może to brzmieć śmiesznie, ale prawda jest taka, że paru błędów można by uniknąć przy bardziej skrupulatnej analizie dostarczonego listingu. Jak pokazuje doświadczenie – praca z IDE potrafi drastycznie obniżyć czujność programisty. Podstawowe czynności przy analizie otrzymanego kodu to:

- sprawdzanie poprawności indeksów w tablicy, zwłaszcza dotyczy to parametrów przekazywanych do wywołania metody *main()* (z linii poleceń).
- zwracanie uwagi na import stosow-

> Pętle for potrafią być momentami zwodnicze.

nych klas. Mnie sytuacja ta niemile zaskoczyła w przypadku wyrzucanego wyjątku (bodajże *IOException*). Należy pamiętać (przynajmniej taka konwencja jest stosowana w podręczniku), iż kiedy listingi programów rozpoczynają się od linii z numerem 1, wówczas widzimy cały przedstawiony kod i możemy z dużą dozą prawdopodobieństwa założyć, że jest on poprawny. Jednakże kiedy widzimy tylko kod metody, a numeracja linii rozpoczyna się mniej więcej od 5, wówczas należy zwrócić uwagę czy czasem nie brakuje stosownej instrukcji importu.

- czy wszystkie zmienne instancji zostały zainicjalizowane, a jeśli tak to czy na pewno są to określone wartości, czy też mają one przypisane wartości domyślne?
- czy użyto poprawnych identyfikatorów. Należy pamiętać, że taki potworek ($___$___$) rodem z najgorszych zaułków *nasza-klasa.pl* jest jak najbardziej poprawny – świetnie sprawdzi się jako identyfikator zmiennej.
- rzutowanie i polimorfizm. Czy aby na pewno dany obiekt może zachowywać się w dany sposób (głównie dotyczy kolekcji, ale gdzie indziej też potrafi zaskoczyć)?
- troszeczkę uwagi przy czytaniu treści zadania, a konkretniej zrozumienie o co jesteśmy tak naprawdę pytani. Otóż *„compile and run”* to nie to samo co *„run”*, ani też to nie to samo, co *„compile”*. Takie kruczki często są wykorzystywane do zaznaczenia różnic pomiędzy wersjami Javy (głównie w pytaniach poświęconych kompilatorowi).

Rozdział 1 – Deklaracje i kontrola dostępu

“As a Java programmer, you want to be able to use components from the Java API, but it would be great if you could also buy the Java component you want from “Beans ‘R Us,” that software company down the street.”

- Pętle *for* potrafią być momentami zwodnicze. Potencjalna kombinacja pułapek w przypadku deklaracji, bądź też wykonania jest całkiem spora. Zważmy na taki przypadek:

```
for (int __x = 0; __x < 3; __x++);
```

Jest to konstrukcja jak najbardziej poprawna składniowo! Pętla jest pusta, stąd średnik na końcu linii. Użyte identyfikatory są jak najbardziej legalne. Za odpowiednią możemy również uznać poniższą instrukcję:

```
for(int i = 0, y = 2; i < 10 &&
      y < 11; ++i, y++ ) {
  System.out.println(
    „Zmienna  i to: „ + i +
    „zaś zmienna y to: „ + y );
}
```

> Polimorfizm nie dotyczy metod statycznych

Poprawnie zainicjowano zmienne *i* oraz *y*, drugie wyrażenie w pętli daje w wyniku wartość *BOOL*, zaś ostatecznie zwiększamy wartości obu zmiennych. Warto podkreślić, iż niezależnie od użytego operatora (++i czy i++) wartość zmiennej *i* będzie zwiększała się dopiero **po każdorazowym wykonaniu ciała pętli.**

- W Javie 5 pojawiła się konstrukcja, która umożliwia przekazywanie do metody nieznanej zawczasu ilości parametrów konkretnego typu. Rozwiązanie to nazwano **var-args** i jest cudownym wręcz sposobem na zniszczenie nieznającego tej konstrukcji programisty. Użyty w pierwszym rozdziale przykład został wyjaśniony dogłębniej w rozdziale drugim, co jednakże nie przeszkodziło w zamieszczeniu pytania w rozdziale pierwszym.

```
static void sifter(A[]... a2) {
  s += "1"; }
static void sifter(B[]... b1) {
  s += "2"; }
static void sifter(B[] b1) {
  s += "3"; }
static void sifter(Object o) {
  s += "4"; }
```

Wywołania metod, w których deklaracjach wymieniono **var-argsy** są z definicji brane przy przeciążaniu na samiuteńkim końcu (chyba, że to jedyny parametr). I warto sobie tę prawdę wbić do głowy. Drugą dość istotną kwestią jest traktowanie tablic – jak wiadomo są one kontenerem wartości określonego typu, jednakże tablica sama z siebie jest również **obiektem!** Zatem gdybyśmy korzystając z powyższego przykładu wywołali metodę z tablicą jako parametrem, wówczas do zmiennej *s* dopisano by wartość „4”.

- Tablice. Nie potrzeba o nich zbyt rozwlekle pisać, jedyną dziwną rzeczą (zwracają na to uwagę autorzy podręcznika), jest możliwość ich udziwnionego tworzenia. Wszystkie poniższe instrukcje są jak najbardziej poprawne.

```
Boolean [] tablica [];
  //Utworzenie tablicy
  //dwuwymiarowej
Boolean[] tablica;
  //Tablica jednowymiarowa
Boolean tablica [];
  //Jak wyżej
```

Rozdział 2 – Programowanie obiektowe

“[…] when the JVM invokes the true object’s (Horse) version of the method rather than the reference type’s (Animal) version—the program would die a horrible death. (Not to mention the emotional distress for the one who was betrayed by the rogue subclass.)”

- Polimorfizm nie dotyczy metod statycznych. Weźmy dla przykładu ten fragment kodu:

```
public class Tester {
  public static void main(
      String[] sd ) {
    BetaTester t =
      new BetaTester();
    Tester t2 = new BetaTester();

    t.zrobCos();
    t2.zrobCos();
}

  public void zrobCos() {
    System.out.println(1);
  }
}

class BetaTester extends Tester {
  public void zrobCos() {
    System.out.println(2);
  }
}
```

Na wyjściu programu otrzymamy wynik: „ 2 2”. Dlaczego? Gdyż za każdym

> Im niższy coupling, tym większa kohezja, i na odwrót!

razem zostanie wywołana nadpisana metoda *zrobCos* z klasy *BetaTester*. Jednakże teraz wystarczy zmienić jej deklarację (w obu klasach) na *static* i w tym momencie zwracane wartości będą się różniły. W przypadku metod statycznych – polimorfizm nie jest w tym momencie możliwy.

- Polimorfizmu i nadpisywania metod ciąg dalszy. Należy pamiętać o sytuacji, w której jedna klasa rozszerzając drugą, dziedziczy po niej własności i metody (choć to raczej typowe dla klasy, prawda?). Jedna z metod zostaje przeciążona. Własność zaś domyślnie ma nadawaną inną wartość niż miała w klasie rodzicielskiej!. Co z tego wynika? (przykład z podręcznika)

```
public class Tester {
  public static void main(
       String[] args) {
    new Tester().go();
  }

  void go() {
    Mammal m = new Zebra();
    System.out.println(m.name +
      m.makeNoise());
  }
}

class Mammal {
  String name = "siersc ";
  String makeNoise() {
    return "jakis odglos";
  }
}

class Zebra extends Mammal {
  String name = "pasy ";
  String makeNoise() {
    return „meczy";
  }
}
```

Na wyjściu programu zobaczymy „sierść meczy". Jak zatem widać polimorfizm nie dotyczy nadpisywanych własności, albo tez innymi słowy – w przypadku nadpisywania pierwszeństwo ma typ obiektu, a nie referencji do niego.

- Podczas egzaminu przyda się także znajomość tych oto pojęć:

- **kohezja (cohesion)** – to inaczej „spoistość", termin głównie odnoszony do klasy. Dla celów zdania egzaminu wystarczy wiedzieć, że za tym pojęciem stoi dążenie do tworzenia klas skupionych ściśle na jednym określonym zadaniu.

- **zależność (coupling)** – tłumaczenie może nie jest odpowiednie, jednakże jakoś nie przychodzi mi do głowy inne. Otóż *coupling* to wyznacznik zależności pomiędzy klasami lub modułami. **Im niższy coupling, tym większa kohezja, i na odwrót!** Im mniejsze zależności tym klasy bardziej autonomiczne i „niemieszające" we własnych implementacjach.

Rozdział 3 – Przypisania

"Again, repeat after me, 'Java is not C++.'"

- Bloki inicjalizacyjne są tym zagadnieniem, które potrafi nieźle namieszać i ostatecznie doprowadzić do utraty punktów na zasadniczo banalnych pytaniach. Jak powszechnie wiadomo bloki inicjacyjne dzielimy na **statyczne** i **instancyjne**. Bloki statyczne, jak większość artefaktów języka oznaczonych statycznie – są wykonywane tylko raz – podczas ładowania klasy przez maszynę wirtualną. Bloki instancyjne wykonywane są przy tworzeniu każdej nowej instancji obiektu. W czym problem? W zwróceniu uwagi na dwie rze-

> przy przekazywaniu prymitywów
> lub referencji do obiektu przekazujemy kopię

czy:

- **kolejność wykonywania** – bloki są wywoływane po wywołaniu wszystkich konstruktorów (także rodziców). W przypadku wystąpienia kilku z nich w jednej klasie pod uwagę brana jest kolejność (z góry do dołu).

- **podchwytliwe pytania** – mieszanie trzech klas, w których jedna dziedziczy po drugiej zaś najstarsza z nich posiada bloki statyczne… bla, bla, bla. Takie pytania to esencja pytań o bloki inicjalizacyjne. Spójrzmy na taki kod:

```
class A {
  {
    //  inicjalizujemy zmienne
  }
}

class B extends A {
  static
  {
    //też coś robimy
  }
}

class C extends B {
// tutaj metoda main, konstruktor,
również statyczne bloki inicjali-
zujące
}
```

Na powyższym przykładzie widać rzecz wyraźnie – klasa B dziedziczy po A, zatem w momencie kiedy kompilator napotka słowo kluczowe *extends*, ładuje klasę A do pamięci **oraz wykonuje jej statyczne bloki inicjalizujące!** Problem polega na tym, aby nie rozpędzić się i nie zauważyć problemu z tym, iż klasa A nie ma **statycznych bloków inicjalizujących!** Na takie „oczywiste oczywistości” bardzo łatwo się nabrać.

- „Przyciemnianie zmiennych” (*shadowing*) to kolejna wariacja problemów z polimorfizmem. Rzecz jest dość prosta, wystarczy pamiętać o kilku istotnych zagadnieniach:

- **przy przekazywaniu prymitywów lub referencji do obiektu przekazujemy kopię** – skutkuje to faktem, iż zmiany dokonywane na wartości prymitywnej dotyczą tylko ciała konkretnej metody – dokładniej kopii przekazanej wartości. Jeśli mowa o referencjach do obiektów to należy pamiętać o tym, iż możemy **zmienić wskazywany obiekt**, nie jesteśmy zaś w stanie **zmienić samej referencji!** Zatem w poniższym kodzie:

```
public function zrobCos(
  Klasa a ) {
  a.setWartosc( 2 );
    // To jest dozwolone
  a = null;
    // No ale to już nie
}
```

- **modyfikator final dotyczy referencji, nie wskazywanego obiektu!**

- *Var-argsy* po raz mam nadzieję ostatni. Metoda z parametrami w formie *var-args* zostanie wybrana tylko i wyłącznie wówczas, kiedy nie będzie **żadnej innej metody dla pojedynczych parametrów!** Oto mały przykład (z podręcznika):

```
public class Bertha {
  static String s = "";
  public static void main(
      String[] args) {
    int x = 4;
    Boolean y = true;
    short[] sa = {1,2,3};

    doStuff(x, y);
    doStuff(x);
```

> Operatory są zwodnicze! Przeczytać trzy razy, splunąć przez lewe ramię, obrócić się przez ramię prawe, przeczytać ponownie

```
    doStuff(sa, sa);
    System.out.println(s);
  }

  static void doStuff(Object o) {
    s += "1"; }
  static void doStuff(
    Object... o) {
    s += "2"; }
  static void doStuff(
    Integer... i) {
    s += "3"; }
  static void doStuff(Long L) {
    s += "4"; }
}
```

Co zobaczymy na wyjściu? "212" – istotną rzeczą do zapamiętania jest fakt, iż w przypadku poszerzania prymitywów (*widening*), mechanizm ten nie działa w połączeniu z *autoboxingiem*. Zatem wywołanie metody *doStuff()* z parametrem *int* nie może skorzystać z wywołania ostatniej deklaracji metody (dla parametru typu *Long*). Stąd wywołanie pierwszej wersji (z parametrem typu *Object*).

- *Garbage Collector* (GC). Podchwytliwie można naciąć się na pytaniach o możliwość usunięcia obiektu z pamięci przez odśmiecacz. Należy bowiem pamiętać o tym, iż składowe klasy będące obiektami również liczą się jako obiekty, które posiadają prawo do własnego bytowania. Dlatego instancja poniższej klasy:

```
public class Chlebik {
  String nick = „Chlebik”;
  int    wiek = 25;
}
```

Nie zostanie usunięta nawet po „zgubieniu" do niej referencji, jeśli składowa *nick* będzie wciąż dostępna. **Dotyczy to zwłaszcza statycznych składowych klas.**

Rozdział 4 – Operatory

"Having said all this about bitwise operators, the key thing to remember is this:

BITWISE OPERATORS ARE NOT ON THE EXAM!"

- **Operatory są zwodnicze!** Przeczytać trzy razy, splunąć przez lewe ramię, obrócić się przez ramię prawe, przeczytać ponownie, zasada 4xZ (zapamiętać, zapamiętać, zapamiętać, zapamiętać)
- Operatory inkrementacji (++) oraz dekrementacji (--) są używane „na bieżąco" nawet jeśli operują na tej samej zmiennej w jednym wierszu:

```
static int zmienna = 7;

public static void main(
     String[] args) {
  System.out.print( zmienna++ +
    „ „ + ++zmienna );
}
```

Powyższy kod w wyniku da „7 9".

- Wartości ENUM można porównywać zarówno z użyciem operatora == jak i metody *equals*. Zawsze zwrócą to samo.
- Operator && przy pomyślnym wyniku pierwszego argumentu przechodzi do drugiego. Jeśli jednakże pierwszy argument zwróci wartość FALSE, wówczas kończy działanie.

Rozdział 5 – Przepływ, wyjątki i asercje

"Using assertions that cause side effects can cause some of the most maddening and hard-to-find bugs known to man! When a hot tempered Q.A. analyst is screaming at you that your code doesn't work, trotting out the old 'well it works on MY machine' excuse won't get you very far. "

> Zmienna wskazująca na kolejne elementy tablicy musi zostać zadeklarowana w ciele pętli.

- Metoda main() może zadeklarować wyrzucanie wyjątków. Dlaczego by nie?
- O ile w przypadku pętli *for* nie ma problemów z użyciem w roli licznika zmiennej zadeklarowanej wcześniej (np. zmiennej lokalnej), o tyle w przypadku *rozszerzonej pętli for* taka operacja spowoduje wygenerowanie błędu. Poniższy kod nie zadziała:

```
Integer i = 1;
for(i:jakasTablicaWartosciInteger
) {}
```

Jak widać zmienna wskazująca na kolejne elementy tablicy musi zostać zadeklarowana w ciele pętli.

- Ciekawe są pytania o potencjalną możliwość przepełnienia stosu (*StackOverflow*). Należy pamiętać, że samo wywołanie nieskończonej pętli, **która nie alokuje dodatkowej pamięci**, nie spowoduje przepełnienia stosu. Zatem kod:

```
for( int i = 0; i < 10; i++ ) {
  if( i == 9 ) i = 1;
}
```

Będzie się wykonywał dopóki Google będzie miało swoje serwery i jeden dzień dłużej. Jeżeli jednakże zrobimy coś takiego:

```
public class Tester {
  public static void main(
      String[] args) {
    new Tester().zrobCos();
  }

  void zrobCos() {
    zrobCos();
  }
}
```

Zaowocuje pięknym *StackOverflowError*, gdyż kolejne wywołania metody wymuszają zarezerwowanie pewnej części pamięci dla swego działania.

- Klasy nadpisujące metody ze swej klasy rodzicielskiej nie mogą deklarować szerszego zakresu generowanych wyjątków niż klasy rodzicielskie! Czyli jeśli metoda *zrobCos()* deklaruje, że wyrzuca *IOException*, wówczas jeśli klasa potomna chce nadpisać tę metodę nie może zadeklarować wyrzucenia wyjątku o typie *Exception*!
- Jeśli zamiast standardowego formatu zapisu asercji zdecydujemy się na drugi (z możliwością wygenerowania na wyjście pewnej informacji), wystarczy, że drugi parametr asercji będzie zwracał **jakikolwiek obiekt** (choć bardziej rzecz w metodzie *toString()*).

Rozdział 6 – Łańcuchy, parsowanie, I/O, formatowanie

"There are over 500 ISO Language codes, including one for Klingon ('tlh'), although unfortunately Java doesn't yet support the Klingon locale. We thought about telling you that you'd have to memorize all these codes for the exam…but we didn't want to cause any heart attacks."

- Co ostatecznie podlega, a co nie podlega serializacji? Pytanie jest dość istotne, gdyż w grę wchodzi szereg czynników, na które dobrze jest zwracać uwagę. Rozpatrzmy zachowanie takiego kodu:

```
class A {}

class B extends A implements
    Serializable {
  A zmienna = new A();
  static int zmienna2 = 9;
  int transient zmienna3 = 1;
}
```

> Jeżeli dwa obiekty są znaczeniowo te same (przy nadpisaniu metody equals), wówczas ich hashcodes muszą być takie same. Nie działa to jednakże w drugą stronę

W przypadku serializacji i deserializacji sprawa wygląda następująco. Na etapie kompilacji kodu **nie zostaną wyłapane oczywiste błędy!** Klasa A nie może być serializowana, co mimo to nie powoduje błędu kompilacji klasy B. W toku działania programu zostanie wygenerowany wyjątek, jednakże kompilacja się powiedzie.

Zmienna o identyfikatorze *zmienna3* po procesie deserializacji **nie otrzyma wartości 1**, ale za to domyślną wartość dla prymitywu o typie *int* (czyli 0). Pozostając już przy deserializacji wypada również wspomnieć o tym, iż o ile klasa B podlega serializacji i jej odtworzenie nie powoduje uruchomienia konstruktora, o tyle zostanie wywołany konstruktor klasy rodzicielskiej (czyli A).

Na sam koniec zostawiłem wartości statyczne – należy pamiętać, że istnieją one niezależnie od instancji, zatem ich wartości nie są w ogóle „ruszane" przez proces serializacji/deserializacji.

- Rozdział ten skupia się w znacznej mierze na API – ktoś kto miał okazję pracować z Javą przez dłuższy czas nie powinien mieć większych problemów. Początkujący mogą jednakże momentami napotkać ciekawe kwiatki. Ja natrafiłem na dwa:
  - niekonsekwencję w nazewnictwie metod (*mkdir()*, ale *createNewFile()*)
  - metoda *setMaximumFractionDigits()* klasy *NumberFormat* dotyczy tylko jej metody *format()*, w przypadku metody *parse()* nie ma ona żadnego wpływu na wynik.

Rozdział 7 – Typy generyczne I kolekcje

"You're an object. Get used to it. You have state, you have behavior, you have a job. (Or at least your chances of getting one will go up after passing the exam.)"

Zanim w ogóle zacznę, pragnę poinformować, iż ten rozdział doczekał się u mnie na blogu dłuższego wpisu. Zatem poniższa lista jest tylko szybkim podsumowaniem – zachęcam do przeczytania całego artykułu.

- Należy zapamiętać proste zależności między równością obiektów oraz ich *hashcode*. Jeżeli dwa obiekty są znaczeniowo te same (przy nadpisaniu metody *equals*), wówczas ich *hashcodes* **muszą być takie same.** Nie działa to jednakże w drugą stronę – jeśli dwa obiekty posiadają takie same *hashcodes* to wcale nie oznacza, iż są one znaczeniowo tożsame.
- Pozostając w tematyce *hashcodes* – jak podają sami autorzy na potrzeby egzaminu można założyć, iż kiedy nie nadpisujemy metody *hashcode* w klasie, wówczas każdy obiekt jest inny – będzie posiadał inny *hashcode*.
- Często w deklaracjach można napotkać takie oto krzaczki:

```
List<List<Integer>> table =
  new List<List<Integer>>();
```

Co raczej nie zadziała, gdyż *List* jest interfejsem i nie za bardzo można tworzyć jego instancje. Nie zadziała również taki kod:

```
List<List<Integer>> table =
new ArrayList<ArrayList<Integer>>();
```

Nie zadziała z tej prostej przyczyny, iż w przypadku generyków deklaracja musi

VRILTEK.COM

Logotypy

Portale

Plakaty

Wizytówki

Strony WWW

DTP

Sklepy internetowe

Ulotki

Banery

> Metody niestatyczne nie mogą być wywoływane ze statycznych metod

pokrywać się z tworzonym obiektem, pomimo faktu, iż *ArrayList* rozszerza *List*. Co więcej – ta sama sytuacja dotyczy metod. Jeśli ma zwracać np.: List<Number> to jeśli ostatecznie zwracamy ArrayList<Integer> - również nie da się takiego kodu skompilować.

- Ciekawą klasą jest *TreeSet*. Kwestia konkretnie dotyczy dwóch rzeczy, choć wynikających z tego samego założenia. Klasa *TreeSet* posiada kilka różnych konstruktorów, których celem jest stworzenie zbioru elementów, które są uporządkowane w formie drzewa. Co z tego wynika? Ano tyle, iż elementy będące składnikami klasy muszą implementować interfejs *Comparable*, albo też „same z siebie" posiadać możliwości porównywania jednych z drugimi (np. klasa *String*). Jeśli do obiektu klasy *TreeSet* (bez wykorzystania typów generycznych) dodamy nawet kilka obiektów różnych klas kod skompiluje się bez problemu. Natomiast z całą pewnością dostaniemy błędy podczas uruchomienia programu – kompilator po prostu nie będzie wiedział z jakimi obiektami ma do czynienia i wyrzuci *ClassCastException*.

Rozdział 8 – Klasy wewnętrzne

"More important, the code used to represent questions on virtually any topic on the exam can involve inner classes. Unless you deeply understand the rules and syntax for inner classes, you're likely to miss questions you'd otherwise be able to answer. ***As if the exam weren't already tough enough****."*

- Zasadniczo zagadnienia dostępności/widoczności poszczególnych klas czy metod dotyczy większość pytań w tym rozdziale. Tutaj napiszę tylko, że w przypadku *method-local inner class* ma ona nielimitowany dostęp do klasy otaczającej. Taki kod:

```
public class Tester {
  final String lancuch = „Chlebik";
  public void pokazKlaseWewnetrz-
na() {
    class takaSobieKlasa {
      void hej() {
        System.out.println(
          lancuch );
      }
    }
  }
}
```

jest jak najbardziej w porządku i zadziała bez problemu.

- pamiętaj o statykach – zarówno o tym, że metody niestatyczne nie mogą być wywoływane ze statycznych metod, a także, że statyczne zmienne trzymają się twardo i mają za nic tworzenie instancji klasy.
- widoczność klas wewnętrznych – kod z pytania testowego:

```
class A { void m() {
  System.out.println("outer");
  }
}

public class TestInners {
  public static void main(
      String[] args) {
    new TestInners().go();
  }

  void go() {
    new A().m();
    class A {
      void m() {
        System.out.println(
          "inner");
      }
    }
  }

  class A {
    void m() {
```

> Metoda sleep() jest metodą statyczną klasy Thread. Dlatego też zawsze usypia działanie bieżącego wątku

```
        System.out.println(
          "middle");
      }
    }
}
```

Pytanie brzmi – co zostanie wyświetlone na wyjściu programu? Odpowiedź: *middle*. Dlaczego? Ano bo klasa wewnątrz metody jest zadeklarowana dopiero po wywołaniu (czyli po fragmencie *new A().m();*, w związku z czym jest niewidoczna. Klasa A, która jest w pierwszej linijce kodu jest "poziom wyżej" niż klasa dająca na wyjściu "middle" i dlatego też nie zostanie wykorzystana.

Rozdział 9 – Wątki

"In fact, the most important concept to understand from this entire chapter is this:

When it comes to threads, very little is guaranteed."

- Usypianie wątku to bardzo interesujące zagadnienie. O metodzie *sleep()* trzeba wiedzieć dwie rzeczy. Raz – jest metodą statyczną klasy *Thread*. Dlatego też zawsze usypia działanie bieżącego wątku. Do tego wyrzuca ona *InterruptedException*, zatem musimy wywołanie metody zawrzeć w klauzuli *try..catch*, albo też przekazać obsługę wyjątku wyżej.
- Metoda *join()* – podobnie jak powyższa wyrzuca wyjątek *InterruptedException*. Nie jest jednakże metodą statyczną. Spójrzmy na taki kod:

```
public static void main( String[]
args ) {

  Thread t = new Thread(
    new Runnable() {
      public void run() {
        System.out.println(
          "Początek pętli" );
        for( double i = 0;
          i < 1000000000; i++ )
        { }
        System.out.println(
          „Koniec pętli" );
      }
    }
  );

  System.out.println(
    „Chlebik 1:" );
  t.start();
  System.out.println(
    "Chlebik 2:" );

  try {
    t.join();
  } catch( Exception e ) { }

  System.out.println(
    „Chlebik 2:" );
}
```

Obecnie wykonywany wątek (ten z metody *main*) nie wyświetli napisu „Chlebik 2:" dopóki nie zostanie zakończone działanie wątku reprezentowanego przez obiekt *t*!

- Metoda *wait()* – odstaje troszeczkę od powyższego towarzystwa, gdyż jest ona właściwa dla wszystkich obiektów w Javie (pochodzi z klasy *Object*). Nie jest statyczna, zaś jej wywołanie nie powoduje wyrzucenia wyjątku. W parze z nią idą dwie inne metody z klasy *Object* – *notify()* i *notifyAll()*. Wszystkie są oznaczone jako *final*, zatem nie potrzeba w ich przypadku karkołomnych zabaw jak z np. metodą *equals()*.

Idąc dalej – metody te mogą być wywołane tylko i wyłącznie w kontekście *synchronized*! Próby użycia poza tymże kontekstem skutkują wyrzuceniem *IllegalMonitorStateException* (i to nie jest sprawdzalny wyjątek więc nie trzeba definiować jego łapania). Meto-

> I tak raz za razem – kolejność cyfr oczywiście bywa różna

dy te służą do zarządzania blokadami obiektu (dlatego są elementami klasy *Object*). Metoda *wait()* pozwala na wstrzymanie działania wątku, który posiada blokadę obiektu, aż do wywołania przez ten obiekt metody *notify()*, albo *notifyAll()*. Nie za bardzo podejmuję się więcej tłumaczyć to pisząc – myślę, że kod powie więcej (wzięty z podręcznika):

```
class ThreadA {
  public static void main(
      String [] args) {
    ThreadB b = new ThreadB();
    b.start();

    synchronized(b) {
      try {
        System.out.println(
          "Waiting for b to" +
          complete...");
        b.wait();
      } catch
        (InterruptedException e)
      {}
      System.out.println(
        "Total is: " + b.total);
    }
  }
}

class ThreadB extends Thread {
  int total;

  public void run() {
    synchronized(this) {
      for(int i=0;i<100;i++) {
        total += i;
      }
      notify();
    }
  }
}
```

- Kiedy blokada istnieje, a nie kiedy nie – to temat bardzo istotny. Oto cytat z Thinking in Java w wersji 4.

„Ważne jest, aby zrozumieć, że wywołanie metody sleep() nie zwalnia blokady obiektu, tak samo jak nie czyni tego wywołanie yield(). Z drugiej strony, wywołanie wait() zainicjowane w obrębie synchronizowanej metody wymusza zawieszenie wątku i zwolnienie blokady danego obiektu."

- Metoda *getId()* – na to się trochę wkurzyłem. Ni stąd, ni zowąd wyskoczyły mi pytania o tę metodę. Może to i sposób, aby w pytaniach testowych poruszać zagadnienia, których nie było w konkretnym rozdziale. Ale to nie lepiej by było po prostu dać listę metod, o które potencjalnie jeszcze mogą paść pytania? Uczenie się całego API na pamięć to chyba nie jest cel egzaminacyjny? No nic, koniec narzekania – rzecz w metodzie *getId()*.

  Dokumentacja mówi, że zwraca ona (metoda rzecz jasna) unikatowy identyfikator wątku (prymityw typu long). Spójrzmy na ten kod:

```
// Obiekty a-a3 są tego samego
// typu jak ten poniżej
read a4 = new Thread(
    new Runnable() {
  public void run() {
    for( int i = 0; i < 100000;
        i++ ) {
      if( i == 99999 )
        System.out.println('.' +
        Thread.currentThread().
          getId());
    }
  }
});

System.out.println(
  Thread.currentThread().
  getId());
a.start();
System.out.println(a.getId());
a2.start();
System.out.println(a2.getId());
```

> Błędy wykonania (*run*),
> to co innego niż błędy kompilacji (*compile*).

```
a3.start();
System.out.println(a3.getId());
a4.start();
System.out.println(a4.getId());
```

Oto efekt działania:

```
1
8
9
10
11
54
55
56
57
```

I tak raz za razem – kolejność cyfr oczywiście bywa różna (poza pierwszymi trzema-czterema) – wiadomo, nieokreśloność wywoływania wątków. Tak to wygląda. Dwie kwestie – jak widać wątek metody *main()* **zawsze** (przynajmniej u mnie) ma numer 1. Pozostałe w miarę równo i oczywiście idąc w górę. Pytania na testowym egzaminie dotyczyły ewentualnego wyniku na wyjściu programu podobnego do powyższego. Ciekawe, ale jednak to w wielu momentach jest po prostu loteria.

Rozdział 10 – Development

"When you start to put classes into packages, and then start to use classpaths to find these classes, things can get tricky. The exam creators knew this, and they tried to create an especially devilish set of package/classpath questions with which to confound you."

- Statyczne importowanie – oczywiście koniecznym jest ich użycie za pomocą słów kluczowych *static import* (w takiej kolejności). Jednakże mniej oczywistym zapisem jest to, iż możemy importować w ten sposób nawet pojedyncze stałe i metody.
- asercje – było o nich w rozdziale piątym, ale nie zaznaczyłem tam rzeczy najistotniejszej. Otóż należy pamiętać, że asercje zostały wprowadzone już w wersji 1.4! I dlatego też wywoływanie kompilatora i wirtualnej maszyny w ten sposób:

```
javac -source 1.4 plik.java
java -ea plik
```

Spowoduje, że kod, w którym występują niespełnione asercje (zwracające wartość *false*) spowoduje wygenerowanie błędu podczas wykonania programu (czyli po prostu asercje będą działały). **Wykonania, nie kompilacji!** Powtórzmy – błędy wykonania (*run*), to co innego niż błędy kompilacji (*compile*).

- Do czego tak naprawdę służy nam dyrektywa *classpath*? Otóż jej zadaniem jest głównie znalezienie wszystkich klas, których kompilowana/uruchamiana klasa będzie potrzebowała. To jest główne zadanie dla *classpath*. Pamiętać należy również o tym, iż w przypadku kompilacji (polecenie *javac*) podanej nazwy pliku do kompilacji poszukuje się domyślnie w bieżącym katalogu. W przypadku uruchamiania pliku tak nie jest! No i rzecz ostatnia – podanie wartości dla *classpath* powoduje nadpisanie zmiennej systemowej (o ile rzecz jasna istnieje).
- pliki JAR – archiwa są dość proste do zrozumienia, co więcej, na egzaminie nie ma pytań dotyczących ich tworzenia i zarządzania. Natomiast na pewno trzeba wiedzieć, że po utworzeniu pliku JAR z konkretnego katalogu, nawet po dodaniu go do *classpath* do klas zawartych w archiwum należy odwoływać się w kodzie poprzez podanie pełnej nazwy klasy (łącznie z nazwą pliku

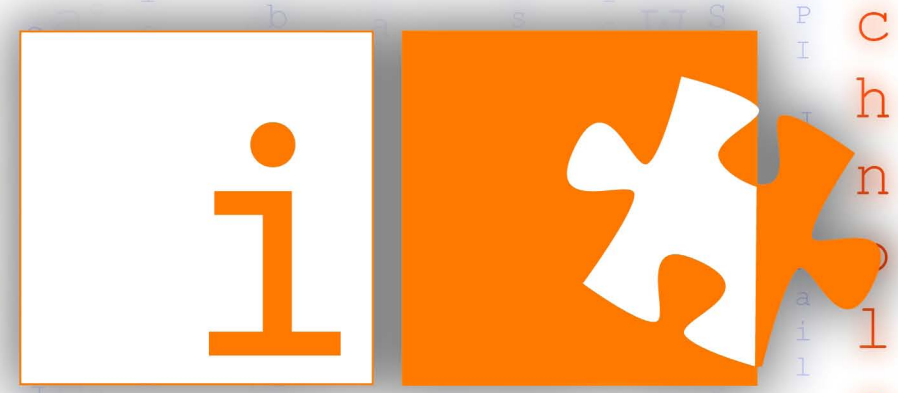
> Uczenie się całego API na pamięć
> to chyba nie jest cel egzaminacyjny?

JAR). Oto przykład:

```
test|
     plik.uzywajacy.klasy.z.jara
     tutaj.utworzymy.plik.jar
     katalog.do.zjarowania|
           podkatalog1
           podkatalog2  |
               plik.java
```

Odwołując się do pliku w archiwum JAR, które utworzyliśmy w katalogu test należy podawać pełną ścieżkę. A zatem nasz plik w katalogu test, w którym chcielibyśmy wykorzystać klasę z archiwum musi odwoływać się do niej poprzez zapis PLIK_JAR/katalog.do.zjarowania/podkatalog2/plik – pomimo dodania pliku JAR do *classpath*.

Michał Piotrowski jest programistą „przypadkowym" – bo co mogą mieć wspólnego magisterium z politologii oraz współtworzenie (w PHP, o zgrozo!) portalu społecznościowego? Nic. Pewnie dlatego taki fenomen mógł w ogóle zaistnieć. W wolnym czasie, którego wciąż mu brakuje, poznaje tajniki Javy (ze szczególnym upodobaniem do JEE), a efekty swych postępów publikuje na blogu dostępnym pod adresem *http://chlebik.wordpress.com*



Java Team

dołącz do nas www.isolution.pl

OSGi: Modularność bez restartów w aplikacjach JEE

Joris Kuipers

Ci, którzy śledzili rozwój technologii związanych z Javą 7 z pewnością spostrzegli wszechobecność pojęcia modularności. To zagadnienie przewija się przez kilka Java Specification Requests (JSR), takich jak (aktualnie zawieszony) JSR-277, oparty na technologii OSGi JSR-291, system modułów dla Javy, który jest częścią samego języka (JSR-294) czy też projekt Jigsaw, będący Sun-owską inicjatywą w kierunku modularyzacji maszyny wirtualnej. Chociaż większość ze wspomnianych JSR-ów odkrywa nowe obszary, sam standard OSGi ugruntował już na dobre swoją pozycję w wielu licznych zastosowaniach. Stał on się standardem około roku 1999 w środowiskach systemów wbudowanych. Biorąc pod uwagę ograniczoną pojemność takich systemów niemożliwym było uruchomienie nowej maszyny wirtualnej dla każdej aplikacji, skutkiem czego aplikacje musiały w łatwy i bezpieczny sposób współdzielić jedno środowisko uruchomieniowe. Java sama w sobie nie dostarcza wystarczającego wsparcia w tym zakresie. Z wielu powodów: liniowy *classpath* nie pozwala na pracę z wieloma wersjami tej samej biblioteki, nie istnieje skuteczna metoda ukrycia szczegółów implementacyjnych przed innymi aplikacjami oraz zadeklarowania zależności explicite. Ponadto, brakuje jednego standardowego mechanizmu, który zapewniałby aplikacjom jeden jasny cykl życia; nie ma także mechanizmu publikacji i odkrywania usług oferowanych przez aplikacje działające na tej samej maszynie wirtualnej. Aby umożliwić aplikacjom od różnych dostawców efektywną i wydajną pracęw obrębie jednej maszyny wirtualnej bez konieczności restartu po każdej instalacji czy aktualizacji, mówimy o wymaganiach importu.

Aby sprostać tym wymaganiom, utworzony został standard OSGi. Jest on zarządzany przez wiele organizacji tworzących konsorcjum Gosi. Istnieją liczne implementacje standardu, z których obecna wersja jest nosi numer 4.1: wersja 4.2 jest w trakcie ciągłego rozwoju i ma być opublikowana jeszcze w tym roku. Do dobrze znanych open source'owych implementacji można zaliczyć Apache Felix i Equinox. Ten ostatni jest znany zwłaszcza jako silnik, na którym zbudowany został pluginowy model popularnego Javowego IDE – Eclipse'a.

Zastosowania OSGi

Przez lata zastosowania OSGi były ograniczone do systemów wbudowanych (czyli np. tych, które firma BMW montuje w swoich samochodach) i pewnych aplikacji specjalnego przeznaczenia. Jakkolwiek w ciągu ostatnich kilku lat standard ten wzbudzał coraz większe zainteresowanie w społeczności J2EE. Wiele serwerów aplikacyjnych już używa OSGi na swoje wewnętrzne potrzeby. Poza nimi standard wykorzystywany jest przez liczne frameworki i środowiska uruchomieniowe. Pozwalają one rzeszy developerów na wygodne korzystanie z tej technologii. Istnieje również pewna liczba rozwiązań pomagających zajmować się skomplikowanymi relacjami, które obecne są w pełni dynamicznym środowisku takim jak OSGi. Przykładowo, każda usługa może być uruchomiona w dowolnym momencie, ale może także być w dowolnej chwili zatrzymana. Aplikacja wspierająca standard OSGi musi potrafić obsłużyć takie sytuacje. Na pierwszy rzut oka, wydaje się to jedynie zwiększać złożoność systemu szczególnie, jeśli odniesiemy się do tradycyjnego modelu aplikacji Enterprise Java. Po głębszej analizie okazuje się, że obsłużenie takich sytuacji jest warunkiem koniecznym budowy każdej niezawodnej aplikacji, która kontynuuje swoją pracę, także wtedy, gdy inne aplikacje

> Przez lata zastosowania OSGi były ograniczone do systemów wbudowanych i pewnych aplikacji specjalnego przeznaczenia.

bądź usługi są tymczasowo niedostępne (gdyż na przykład instalowana jest nowa wersja).

Projektem, który odgrywa tutaj ważną rolę jest Spring Dynamic Modules (Spring -DM), open source'owy szkielet aplikacji integrujący model framework'u Spring ze standardem OSGi. Efektem jest rozwiązanie, w którym OSGi otrzymuje w pełni dojrzały model komponentowy. Pionierska praca nad Spring-DM, wykonana przez ludzi z zespołu SpringSource, a wcześniej z firm BEA i Oracle, jest obecnie przedmiotem standardyzacji jako dokument RFC-124 i docelowo będzie częścią OSGi 4.2. Spring-DM 2 będzie z kolei wzorcową implementacją tego standardu.

OSGi po stronie serwera

Wszystko powyższe oznacza, iż przed standardem OSGi otwarł się zupełnie nowy obszar zastosowań - strona serwera. Możliwości oferowane przez OSGi takie jak łatwa modularyzacja aplikacji, dynamiczność, a także zorientowanie na usługi stanowią znakomitą odpowiedź na życzenia i wymagania stawiane przed dzisiejszym serwerami aplikacyjnymi. Na przykład, zbyt często obecnie nie można pozwolić sobie na wyłączenie całej aplikacji tylko dlatego, iż jedna jej część wymaga upgrade'u, nie mówiąc już o restartowaniu serwera, na którym aplikacja jest zainstalowana. Ponadto, coraz więcej problemów pojawia się w sytuacjach, w których aplikacja wymaga wielu wersji zewnętrznych bibliotek. Wtedy jedynym rozwiązaniem jest spakować całą aplikację wraz z wszystkimi zależnościami (których rozmiar często przewyższa rozmiar samej aplikacji) do jednego dużego archiwum i mieć nadzieję, iż zależności te będą wzajemnie kompatybilne. Jednakże inna wersja aplikacji znajdująca się wyżej w hierarchii classloaderów nie może mieć dostępu do tych zależności bez specjalnych zabiegów w trybach delegacji classloadera. W przypadku współdzielenia standardowej funkcjonalności pomiędzy aplikacjami praktycznie jedyną realną opcją jest wykorzystanie rozwiązań zdalnych, albowiem brakuje konstrukcji pozwalającej współdzielić między aplikacjami lokalnie oferowane dynamiczne usługi. Zespół Enterprise OSGi Expert Group pracuje obecnie nad zbieraniem wymagań, które dotyczą zastosowania OSGi w środowisku biznesowym oraz nad ustandaryzowaniem niektórych rozwiązań w tej dziedzinie.

Aktualne zastosowania standardu OSGi w serwerach aplikacyjnych ograniczają się wyłącznie do ich środka. Produkty takie jak WebSphere Application Server czy też ostatnia wersja Sun-owskiego Glassfisha w dużym stopniu wykorzystują OSGi wewnętrznie, jednakże w żadnym przypadku nie obejmuje to twórców aplikacji. Muszą oni budować swoje systemy w tradycyjny sposób określony technologią J2E, w której ogromne monolityczne aplikacje osadzane na serwerze bez większej interaktywności czy modularności. Rynek jednak się zmienia. Rozwijanych jest wiele produktów, których głównym celem jest udostępnienie OSGi po stronie serwera, np. Infiniflow tworzony przez firmę Paremus. Produkty te pozwalają ich użytkownikom rozwijać aplikacje biznesowe, które w pełni wykorzystują funkcjonalności oferowane przez standard OSGi. Wymagają one jednak sposobu pracy innego niż ten, do którego przywykła większość twórców aplikacji. Jeden z powodów, dla których większość serwerów aplikacyjnych nie udostępnia programistom standardu OSGi jest obok oczywistych benefitów płynących z jego zastosowania, pewna liczba właściwych

> Aby móc skorzystać z tych wszystkich zalet, wymagane jest, aby wszystkie potrzebne biblioteki były dostępne jako pakunki OSGi z odpowiednimi metadanymi.

tylko dla OSGi problemów. Większość z tych problemów jest spowodowana faktem, iż wiele istniejących bibliotek i frameworków zawiera kod niekompatybilny z modelem OSGi wymuszającym bardzo ścisłe reguły widoczności. Na przykład, domyślnie biblioteka „nie widzi" kodu aplikacji, która ją wykorzystuje, podczas gdy działanie wielu nowoczesnych bibliotek jest oparte właśnie na takim założeniu. Znaczy to, że przykładowo biblioteki nie mogą dynamicznie generować kodu opartego na kodzie aplikacji: w modelu OSGi jest to niemożliwe bez dodatkowej konfiguracji, ponieważ biblioteki te nie deklarują wyraźnie zależności od kodu aplikacji.

The SpringSource dm Server

Innym rozwiązaniem w tym obszarze jest The SpringSource dm Server. Jest to serwer aplikacyjny, który jest oparty na Apache Tomcat, Equinox oraz Spring-DM. Celem jego jest umożliwienie programistom wykorzystania OSGi w ich aplikacjach webowych i korporacyjnych, jednocześnie pozwalając na użycie istniejących bibliotek i frameworków takich jak Spring czy Hibernate. Wielką różnicą w porównaniu z innymi serwerami aplikacji jest to, iż OSGi nie jest skryte gdzieś „pod maską", ale dostępne dla twórców aplikacji, którzy mają możliwość samodzielnego używania i uruchamiania modułów OSGi (nazywanych także bundle'ami bądź pakunkami), co z kolei pozwala na zmodularyzowanie wielkich aplikacji, które komunikują się ze sobą poprzez architekturę zorientowaną na usługi.

Nie jest to SOA tracyjnie pojęta, pojedyncze moduły wciąż działają na tym samym serwerze, ale ich zależności zostały zadeklarowane explicite oraz usługi są w znacznie mniejszym stopniu od siebie zależne. Ma to wiele zalet. Jedna z nich to większy znacznie większy potencjał reużywalności - wiele aplikacji może używać ten sam kod, a nawet korzystać z tych samych usług. Przy wielu aplikacjach działających na tym samym serwerze aplikacyjnym oznacza to mniejsze zużycie pamięci, ponieważ biblioteki muszą być załadowane tylko raz razem ze startem serwera, a nie za każdym razem, gdy startuje aplikacja. Serwer sam w sobie jest również zbudowany z modułów, tak więc nieużywane funkcjonalności mogą być łatwo wyłączone. Oznacza to szybszy czas startu i mniejsze wykorzystanie zasobów. Na przykład serwer Tomcat może być całkowicie wyłączony, jeśli nie ma aplikacji, które potrzebują interfejsu webowego.

Aby móc skorzystać z tych wszystkich zalet, wymagane jest, aby wszystkie potrzebne biblioteki były dostępne jako pakunki OSGi z odpowiednimi metadanymi. Dla wielu popularnych open source'owych bibliotek nie stanowi to problemu. Aby sprostać wymaganiom wobec zmieniających się wersji pakunków zewnętrznych bibliotek, SpringSource zainicjował osobny projekt: Enterprise Bundle Repository, dostępny pod adresem http://www.springsource.com/repository. Na stronie tej można znaleźć setki bibliotek Javowych w formie pakunków OSGi. Można je ściągnąć zupełnie za darmo i używać z dowolną implementacją standardu, nie tylko z dm Serverem.

Istotną funkcjonalnością dodawaną przez dm Server do czystego standardu OSGi jest koncepcja aplikacji składającej się z wielu pakunków, tzw. Platform Archive czy plik PAR. Poprzez spakowanie pakunków jako aplikację, otrzymuje się jedną zarządzalną jednostkę, dla której podstawowe czynności administracyjne takie jak deployment czy monitoring stają się

> Zalety pracy w prawdziwie zmodularyzowanym środowisku przyciąga wielu programistów.

łatwiejsze niż w przypadku pojedynczych pakunków. Pomysł ten pozwala także na pewien rodzaj ‚scopingu', w którym typy i usługi wewnątrz jednej aplikacji nie są widoczne dla innych aplikacji. Jedną z zalet tak sformułowanej enkapsulacji jest to, iż różne wersje jednej aplikacji mogą działać na tym samym serwerze bez powodowania konfliktów. Kod, który ma być współdzielony jest oczywiście nadal dostępny, ale nie jest spakowany jako część aplikacji. Co więcej, dm Server dostarcza rozwiązań do pracy z bibliotekami, które używają technik nie działających wewnątrz standardowego środowiska OSGi, takich jak dynamiczna generacja kodu lub tzw. „wplatanie" kodu, a które to techniki wykorzystywane są przez wiele implementacji JPA. Sewer ten dostarcza także całe zaplecze dla aplikacji webowych, dla których OSGi posiadał jedynie ograniczone wsparcie.

Konkluzje

Jest jasnym, iż OSGi jest wschodzącą gwiazdą w świecie Enterprise Java. Zalety pracy w prawdziwie zmodularyzowanym środowisku przyciąga wielu programistów. Wiele serwerów aplikacji już używa OSGi a wraz z innowacyjnymi produktami takimi jak Paremus Infiniflow czy też SpringSource dm Server użycie standardu OSGi staje się łatwe także dla twórców aplikacji. Potrwa jeszcze trochę zanim OSGi stanie się mainstreamową technologią po stronie serwera, ale bezsprzecznie OSGi w zastosowaniach korporacyjnych to przedsięwzięcie, o którym będzie głośno w nadchodzących latach.

O Autorze

Joris specjalizuje się w technologii J2EE odkąd pojawiły się pierwsze standardy. Wcześniej pracował z poprzednikami J2EE takimi jak IBM SanFrancisco framework. Został mianowany Java Technical Consultant w centralnym banku Holandii w 2003 roku. W kwietniu 2007r. Joris dołączył do zespołu SpringSource w Holandii jako Senior consultant. Specjalizuje się w rozwoju warstwy pośredniej i jest odpowiedzialny za prowadzenie szkoleń z SpringSource dm Servera.

Flex i Java

Cornel Creanga

Czym jest Flex?

Adobe Flex jest jednym z najlepszych rozwiązań dla deweloperów chcących tworzyć aplikację Rich Internet Application. Jest to darmowy, open source'owy framework do budowania wysoko interaktywnych i ekspresywnych aplikacji, działającą we wszystkich głównych wyszukiwarkach i systemach operacyjnych. Aplikacja ta zapewnia nowoczesny, oparty o standardy, język oraz model programowania, który wspiera powszechnie stosowane wzorce projektowe. MXML, język oparty na XML'u, jest używany do opisania wyglądu interfejsu oraz akcji użytkownika. Natomiast ActionScript™ 3, zorientowany obiektowo język programowania o szerokim wachlarzu możliwości, używany jest do tworzenia logiki klienta. Framework ten jest częścią architektury Flash Platform i udostępnia przyjazny dla programistów język.

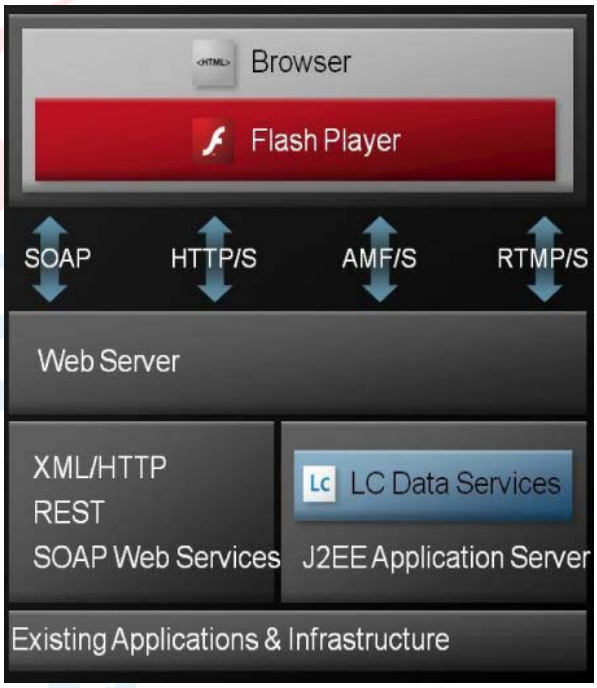
Rodzina produktów Flexa zawiera kilka komponentów:

- Adobe Flex SDK – darmowy framework dostępny na zasadach open source
- Adobe Flex Builder 3 – IDE oparte na środowisku Eclipse
- BlazeDS – open source'owy projekt dla remotingu i messangingu
- Adobre LiveCycle Data Services ES – pozwala aplikacjom RIA dostęp do danych z back-endu i logiki biznesowej
- IBM ILOG Elixir – narzędzie do graficznej prezentacji danych

Niniejszy artykuł wyjaśni w jaki sposób aplikacja kliencka stworzona we Flexie może uzyskać dostęp do części aplikacji Javy po stronie serwera. Dodatkowo tekst ten pokaże jak bezpośrednia i wszechstronna może być integracja Flexa z Javą.

Architektura aplikacji Flex/Java

Istnieje kilka sposobów integracji Flexa z Javą. W celu ich wyjaśnienia, poniżej załączam diagram przedstawiający architekturę aplikacji Flex/Java.



Kanały wymiany danych

Jest kilka możliwości (tzw. kanałów) wymiany danych pomiędzy klientem a serwerem.

Po stronie serwera mamy do czynienia z różnymi rodzajami usług, do których ma dostęp kliencka aplikacja: REST, SOAP i tzw. LC Data Services. Podczas gdy REST i SOAP są obojętne wobec serwera (można generować webserwisy dla dowolnego języka programowania), dla Data services potrzebna będzie serwer aplikacyjny J2EE. Nie obejdzie się też bez użycia open source'owej aplikacji BlazeDS (lub jej komercyjnej wersji LiveCycle DataServices) oraz

> Istnieje kilka sposobów integracji Flexa z Javą.

integracji istniejącego już kodu. W artykule zajmę się jedynie Blaze DS oraz podstawowymi usługami – remote services.

Pierwsze kroki

Zacznijmy od ściągnięcia aplikacji Blase DS z open source'owego repozytorium Adobe [http://opensource.adobe.com/wiki/display/blazeds/Downloads]. Należy pobrać archiwum binarne, gdzie znajduje się plik blazeds.war. Plik ten jest prostą aplikacją webową, w której odnajdziemy pewne jary i pliki konfiguracyjne.

Od tego momentu możliwe są dwie opcje: chcąc zacząć budowanie aplikacji Javy od podstaw, możesz oprzeć ją o strukturę dostarczoną w pliku blazeds.war. Wybierając drugą opcję, powinieneś zintegrować biblioteki i pliki konfiguracyjne z blazeds.war z twoją aplikacją webową. Chcąc dowiedzieć się jak przeprowadzić integrację, możesz skorzystać z artykułu: http://cornelcreanga.com/2009/01/blazeds-lcds-and-integration-with-existing-application/

Sprawę może ułatwić skorzystanie z Flex Buildera, jednak tak naprawdę możesz użyć dowolnego IDE.

Hello World

Zaczniemy od budowania części javowej i stworzymy prostą klasę Hello.

```
package test;

public class Hello {
  public Message sayHello(Message
message) {
    return new Message("Hello " +
message.getBody());
  }
}
```

Oraz klasę Message, która reprezentuje nasz obiekt domenowy

```
package test;

public class Message {
  private String body;

  public Message() {
  }

  public Message(String body) {
    this.body = body;
  }
  // geters and setters..
}
```

Powyżej widać metodę sayHello, która otrzymuje parametr Message a także zwraca obiekt Message. Zobaczmy teraz jak możemy wywołać tą metodę z Flexa.

Pierwszy krokiem jest otworzenie pliku konfiguracyjnego remoting.config.xml i dodanie następującego wpisu:

```
<destination id="hello">
  <properties>
    <source>test.Hello</source>
  </properties>
</destination>
```

Deklaracja ta powiąże klasę Javy z identyfikatorem „hello" - użyjemy tego identyfikatora w aplikacji Flexa w celu wywołania metody z klasy javowej.

Zaczniemy od pierwszego pliku ActionScript – Message.as. Ma on taką samą strukturę jak odpowiadający mu Message.java.

```
package test{
  [RemoteClass(alias=
    "test.Message")]
  public class Message{
    public function Message(){
    }
    public var body:String;
  }
}
```

> Jak to działa? Pierwszym rozwiązaniem jest wykorzystanie protokołu binarnego AMF

Na powyższym przykładzie widać deklarację [RemoteClass(alias="test.Message")]. Jest to informacja dla kompilatora by powiązać klasę ActionScript z klasą javowej.

Główna plikiem tej aplikacji Flexa jest helloworld.mxml. Spójrzmy na kod poniżej:

```
<?xml version="1.0"
  encoding="utf-8"?>
<mx:Application xmlns:mx="http://
www.adobe.com/2006/mxml"
  layout="horizontal">
  <mx:Script>
  <![CDATA[
    import mx.rpc.events.ResultE-
vent;
    import mx.controls.Alert;
    import test.Message;
    private function messageRece-
ived(event:ResultEvent){
      var message:Message =
        event.result as Message;
      Alert.show(message.body);
    }
    private function sendMessage
      (event:Event){
      var message:Message =
        new Message();
      message.body =
        inputName.text;
      service.sayHello(message);
    }
  ]]>
  </mx:Script>

  <mx:Panel title="Hello World"
    id="mainPanel">
    <mx:VBox id="vbox"
      verticalGap="5">
      <mx:HBox width="100%"
        horizontalGap="5"
        paddingTop="10"
        paddingBottom="10"
        paddingLeft="10"
        paddingRight="10">
        <mx:Label
        text="Write your name:"/>
        <mx:TextInput
          id="inputName"
          maxChars="32"
    enter="sendMessage(event)"/>
        <mx:Button
          label="Say hello"
    click="sendMessage(event)" />
      </mx:HBox>
    </mx:VBox>
  </mx:Panel>

  <mx:RemoteObject
    id="service"
    destination="hello"
result="messageReceived(event)"/>
</mx:Application>
```

Jak działa aplikacja?

Omawiana aplikacja jest bardzo prosta. Graficzny interfejs użytkownika zawiera pole tekstowe i przycisk „wyślij". W momencie gdy użytkownik naciska przycisk, aplikacja wywołuje metodę sendMessage. Ma ona również zadeklarowany obiekt zdalny o identyfikatorze „hello" (taki sam jak w przypadku pliku konfiguracyjnego remoting-config.xml), oraz metodę uchwytu (handler method) która zostanie wywołana po tym jak rezultat tej metody będzie zwrócony z Javy do Flexa. Dlaczego potrzebny nam jest ów uchwyt (handler)? Otóż, operacje wejścia i wyjścia we Flexie są asynchroniczne, tak więc musimy zarejestrować metody uchwytu, które zostaną wywołane zgodnie z potrzebami przetwarzania.

Metoda sendMessage jest jasna: wywołuje ona metodę „sayHello" z klasy Java z parametrem Message. I to właściwie wszystko. Nie ma potrzeby ręcznie rzutować danych i jest to także uzasadnione dla złożonych obiektów dziedzin (domain objects).

Jak to działa? Pierwszym rozwiązaniem jest wykorzystanie protokołu binarnego AMF, podstawowego formatu stosowanego w usługach komunikacyjnych BlazeDS. Protokół AMF dokonuje automatycznej serializacji i deserializacji pomiędzy Fle-

> Drugim rozwiązaniem jest zastosowanie refleksji Javy by zinstancjonować obiekty Javy i wywołać metody

xem a Javą stosując tablicę konwersji typów [http://livedocs.adobe.com/flex/2/docs/wwhelp/wwhimpl/common/html/wwhelp.htm?context=LiveDocs_Parts&file=00001103.html] oraz jak zapewnia konwersję danych. Obiekty dziedziny stworzone w Javie powinny mieć swoje odpowiedniki w Action Script (można użyć narzędzia by wygenerować je na podstawie kodu Javy), a komunikacja odbywa się za pomocą bramy AMF. W naszym przypadku zserializowany obiekt Message z Action Script jest konwertowany to pliku Message.java, a następnie zserializowany obiekt Javy jest z powrotem konwertowany do pliku Message.as.

Drugim rozwiązaniem jest zastosowanie refleksji Javy by zinstancjonować obiekty Javy i wywołać metody, czego szczegółowo nie chce omawiać w tym artykule.

Protokół AMF obsługuje również sytuację, kiedy metoda javowa rzuci wyjątkami. Wówczas wyjątek jest serializowany jako zwykły obiekt i zostaje przesłany aplikacji fleksowej.

Dowiedz się więcej

W celu uzyskania szczegółowych informacji jak zintegrować Flexa z projektami Java wejdź na: [http://www.adobe.com/devnet/java/]

O autorze

Cornel Creanga, ewangelista Adobe z Rumunii

Został zatrudniony jako Java Technical Leader w Adobe w sierpniu 2007. W 2001 roku ukończył studia informatyczne w Bukareszcie. Wcześniej pracował m.in. dla Oracle jako software developer i analityk biznesowy, a wcześniej w kilku małych i średnich firmach. Posiada szerokie kwalifikacje w technologii Adobe RIA, bazach danych, frameworkach O/RM, Javie/Javie EE oraz implementowaniu obiektów domenowych. Cornel znany jest z dążenia do ulepszania rzeczy oraz czerpania przyjemności z prowadzenia dyskusji technicznych.

Jeśli chcesz dowiedzieć się więcej o autorze artykułu i/lub nawiązać z nim kontakt, zajrzyj na jego bloga: [http://cornelcreanga.com/].

Lista zadań w Grails

Mateusz Mrozewski

W poprzednim numerze JavaExpress przedstawiłem sposób na rozpoczęcie pracy z Grails, jak zainstalować środowisko, jak utworzyć pierwszy projekt, pierwsze klasy domenowe z ograniczeniami oraz jak wykorzystać dynamiczny *scaffolding*. Teraz zajmiemy się utworzeniem prostej listy zadań w Grails od podstaw. Zobaczymy jak zbudować kontroler, wykorzystać podstawowe funkcjonalności GORM i zbudować własne widoki – w skrócie CRUD ręcznie, bez generowania.

Utworzenie projektu i klasy domenowej

Na początek tworzymy nowy projekt o nazwie ToDo. Jak pamiętamy z poprzedniego artykułu jest to wszystko co tak naprawdę trzeba zrobić, aby rozpocząć pracę z projektem Grails – HSQLDB oraz Jetty w paczce z Grails są już gotowe do pracy. Tworzymy naszą klasę domenową o nazwie `Task`. Nasze zadanie będzie miało temat, priorytet, maksymalny termin wykonania oraz flagę oznaczającą, czy zadanie jest wykonane czy też nie.

```
class Task {
  String subject
  Date dueDate
  Boolean completed

  static constraints = {
  }
}
```

Utworzenie kontrolera

W kolejnym kroku tworzymy kontroler Task (kontroler, jak i klasę domenową, możemy utworzyć korzystając z menu kontekstowego projektu lub wywołując odpowiednią komendę z linii poleceń). Definiujemy w nim następujące metody: `list`, `edit`, `save`, `delete`. Bedą one odpowiednio odpowiedzialne za: wyświetlenie listy zadań, wyświetlenie formularza edycji/dodawania rekordu, zapisanie rekordu (nowego lub zmienionego) oraz usunięcie rekordu.

Pobranie rekordów

Na początku zajmijmy się akcją `list`. W Grails przekazanie danych z kontrolera do widoku odbywa się przez zwrócenie z akcji mapy. Klucze mapy będą nam odpowiednio reprezentowały zmienne dostępne w widoku. Nasza akcja może zatem wyglądać następująco:

```
def list = {
  [tasks:Task.findAll()]
}
```

Nawiasy kwadratowe oznaczają tutaj utworzenie mapy. Pod kluczem *tasks* znajdzie się lista zadań pobranych z bazy. Jest tutaj widoczna jeszcze jedna cecha Groovy – brak słowa kluczowego `return`. Domyślnie zwracany jest wynik ostatnio wykonanej instrukcji, w naszym wypadku utworzona mapa.

Do pełni szczęścia potrzeba nam jeszcze pliku GSP, na którym wyświetlimy nasze zadania. W widoku projektu zaznaczamy `Views and Layouts → task` i z menu kontekstowego wybieramy `New → GSP File`. Jako nazwę podajmy *list*. Grails wyznaje zasadę konwencji ponad konfigurację, tak więc plik widoku nazywa się tak samo jak nazwa akcji. Oczywiście w miare potrzeb można wyrenderować dowolny inny widok, jednak nam na razie wystarczy to, co dostajemy z paczki.

> Do pełni szczęścia potrzeba nam jeszcze pliku GSP

Do naszego widoku dodajemy następujący fragment kodu:

```
<g:each in="${tasks}" var="task">
  ${task}
</g:each>
```

Wykorzystujemy tutaj *tag* dostarczany razem z Grails pozwalający na przeiterowanie po kolekcji elementów w widoku. Atrybut *in* określa zmienną po jakiej będziemy iterować. Atrubut *var* określa, jak w kolejnych iteracjach będzie nazywał się nasz element. Bez podania tego atrybutu będzie się on domyślnie nazywał *it*, tak jak nazwa domyślnego parametru w domknięciach w Groovy. Można również podać atrybut *status*, który będzie określał nazwę zmiennej przechowującej licznik w kolejnych iteracjach (przydatne przy tworzeniu numerowanych list, itp.).

Na tym etapie można by się pokusić o uruchomienie projektu i sprawdzenie działania, jednak nie mamy jeszcze żadnych rekordów, ani widoku do ich dodawania.

BootStrap.groovy

Z pomocą przyjdzie nam tutaj plik BootStrap.groovy. Zawiera on sekcję init, która jest wykonywana podczas startu aplikacji. Możemy w niej dodać następujący fragment kodu:

```
new Task(subject:"Pranie", priority:1,
dueDate:new Date(), completed:false).
save()
```

Utworzy on nam i zapisze w bazie jedno zadanie. Przykład ten pokazuje również w jaki sposób można tworzyć instancje obiektów przekazując wartości dla poszczególnych właściwości – to kolejna przyjazna cecha Groovy. Grails daje również możliwość utworzenia instancji klasy domenowej z automatycznym przypisaniem wartości parametrów żądania do właściwości obiektu. Wykorzystamy to później przy zapisywaniu nowego zadania.

Po takiej modyfikacji pliku *BootStrap* nasza aplikacja powinna już wyświetlać pierwsze zadanie na liście.

Dodawanie i edycja rekordu

Dodawanie i edycję rekordu wykonamy korzystając ze wspólnego widoku. Gdy nie będzie podanego identyfikatora, wyświetlimy pusty formularz w celu dodania nowego zadania. W przypadku, gdy identyfikator zostanie podany, przed wyświetleniem formularza wczytamy odpowiedni rekord w celu umożliwienia jego edycji. Zapis rekordów wykonamy we wspólnej metodzie save.

Na początek przygotujemy sobie widok *edit* (analogicznie do list). Nie musimy na razie nic wpisywać w kontrolerze w akcji *edit*, gdyż póki co chcemy tylko dodać nowy rekord, więc wystarczy nam wyświetlenie pustej formatki.Może ona wyglądać następująco:

```
<g:form name="addForm" action=
  "save" id="${task?.id}">
  Subject: <g:textField
    name="subject"
    value="${task?.subject}"/>
  <br/>
  Due Date: <g:datePicker
    name="dueDate"
    value="${task?.dueDate?:
      (new Date())}"/><br/>
  Completed: <g:checkBox
    name="completed"
    value="${task?.completed}"/>
  <br/>
  <g:submitButton name="save"
    value="Save" />
</g:form>
```

> W Groovy nie tylko typ boolean może zostać użyty w warunkach

Pojawia się tutaj nowy *tag* `g:form`. Służy on do budowania formularzy HTML, a wraz z nim widzimy również `g:textField`, `g:datePicker`, `g:checkBox` oraz `g:submitButton`. Zachęcam do zapoznania się z dokumentacją w zakresie możliwych atrybutów do skonfigurowania dla tych znaczników. W tym fragmencie kodu pojawiają się również dwie ciekawostki z Groovy: operator „?.” - safe navigation operator, który zapobiega wyjątkowi `NullPointerException` (jeśli operand z lewej jest null, to zamiast `NullPointerException` zostanie po prostu zwrócony null), oraz drugi operator „?:”, który jest małą odmianą swojego brata z Javy, operatora ternarnego (jeśli zmienna po lewej jest równa null lub false, to przypisywana jest do niej wartość z prawej, w przeciwnym wypadku wartość pozostaje bez zmian). Dzięki obsłużeniu wartości null oraz wartości domyślnych nasz formularz może zarówno służyć do edycji, jak i dodawania nowych rekordów.

Nasz formularz wysyła dane do akcji save, która zapisze nam nowy rekord:

```
def task = new Task(params)
task.save()
redirect(action:"list")
```

Akcja ta tworzy nowy obiekt klasy Task na podstawie przekazanej mapy parametrów żądania (*params*). Następnie zapisuje ten rekord i wykonuje przekierowanie na akcję wylistowującą nasze rekordy.

Dodajmy zatem odnośnik pozwalający na edycję rekordu oraz zmodyfikujmy akcję *edit* tak, aby wczytywała rekord przed wyświetleniem formatki. Na początek widok *list*:

```
<g:each in="${tasks}" var="task">
  <g:link action="edit"
    id="${task.id}">
      ${task}
  </g:link><br/>
</g:each>
```

Wprowadziliśmy tutaj nowy *tag g:link* pozwalający na tworzenie odnośników. Jak widać Grails w formularzach oraz odnośnikach domyślnie wspiera podawanie nazw kontrolerów, akcji oraz identyfikatorów. Trzymanie się tutaj konwencji narzuconej przez Grails bardzo ułatwia sprawę, jednak należy pamiętać, że w miarę chęci i potrzeb nad wszystkim mamy kontrolę.

Zmodyfikowana akcja *edit* wygląda w następujący sposób:

```
if (params.id) {
  [task:Task.get(params.id)]
}
```

W momencie, gdy został podany parametr żądania o nazwie *id*, to staramy się wczytać rekord task o właśnie takim identyfikatorze. Ciekawie wygląda tutaj instrukcja warunkowa. Zauważmy, że *params* to mapa zawierająca parametry żądania, które są typu *String*. Jak w takim razie zadziałało to w warunku? To kolejna ciekawa cecha Groovy, tzw. *Groovy Truth*. W Groovy nie tylko typ *boolean* może zostać użyty w warunkach. Wartość 0, pusty łańcuch, null, pusta tablica lub mapa również traktowane są jak wartość negatywna w warunkach. Znacznie to upraszcza część operacji.

Pozostała nam do zmodyfikowania akcja save. Dotychczas tworzyła ona nowy rekord na podstawie przekazanych parametrów, a teraz będzie musiała dodatkowo obsłużyć zmiany w istniejącym rekordzie:

```
def task
if (params.id) {
  task = Task.get(params.id)
  task.properties = params
} else {
```

> A co najważniejsze, jej wykonanie nie zajęło nam specjalnie dużo czasu

```
    task = new Task(params)
}
task.save()
redirect(action:"list")
```

Usuwanie rekordu

Pozostało nam usunięcie rekordu. W tym celu na widoku list dodajemy nowy odnośnik przy każdym rekordzie przekierowujący na akcję delete. Tym razem obejdziemy się bez pokazania kodu – każdy na pewno sobie poradzi.

A jak będzie wyglądała nasza akcja w kontrolerze? Zgodnie z oczekiwaniami, niezbyt skomplikowanie:

```
def task = Task.get(params.id)
task.delete()
redirect(action:"list")
```

Podobnie, jak w akcji save wczytujemy tutaj rekord na podstawie podanego parametru żądania. Następnie usuwamy go i wykonujemy przekierowanie na listę zadań.

Podsumowanie

W tym artykule przedstawiłem jak wykonać prostą aplikację Grails z wszystkimi operacjami CRUD. Tym razem nie korzystaliśmy ze scaffodingu i wszystko stworzyliśmy ręcznie. Nasza aplikacja nie zalicza się do najbardziej rozbudowanych, ale jest już funkcjonalna, a co najważniejsze, jej wykonanie nie zajęło nam specjalnie dużo czasu. Poznaliśmy też podstawowe operacje GORM (Grails Object Relationship Mapping), które pozwoliły nam wczytać, zapisać i usuwać rekordy. Wiemy już również jak wygląda klasa domenowa, kontroler oraz widoki, czyli podstawowe składowe Grails. Jednocześnie poznaliśmy też parę ciekawostek Groovy, które może zachęcą niektórych do jego poznania (nie koniecznie w parze z Grails).

Co dalej

Powszechnie wiadomo, że samodzielne rozwiązywanie problemów poprawia zapamiętywanie. W związku z tym, w ramach samodzielnych ćwiczeń proponuję wykonanie następujących zadań:

- modyfikację listy zadań, tak aby była czytelniejsza i bardziej przyjazna dla oka
- wprowadzenie walidacji danych (np. niepusty temat zadania). W ramach podpowiedzi: trzeba zadbać o sekcję contraints klasy domenowej oraz zapyrzjaźnić się z metodą validate(). Dodatkowo przyjdzie nam z pomocą tag g:renderErrors
- Wprowadzić menu na stronie. W ramach podpowiedzi: można zmodyfikować układ strony, najlepiej przez dołączenie do niego oddzielnego elementu, którym będzie menu.

Kod źródłowy można znaleźć pod adresem: http://code.google.com/p/java-express-grails-todo/ . Być może ktoś z czytelników będzie chciał rozwinąć projekt. Powodzenia i miłej zabawy.

Automatyczne generowanie kodu

Marek Berkan

Wstęp – dla kogo to jest?

Automatyczne generowanie kodu ma zastosowanie w dwóch zasadniczych przypadkach: gdy developer chce szybko napisać dużą część powtarzalnego kodu lub gdy implementujemy duży system z udziałem wielu osób, przewidziany do wieloletniego rozwoju i utrzymania.

Development bez użycia automatycznego generowania kodu - przykład

Żeby zainspirować wyobraźnię czytelnika przedstawię sytuację z jaką możemy się spotkać nie stosując automatycznej generacji kodu w najbardziej powszechnym miejscu jakim jest dostęp do bazy danych (zwany najczęściej O/R mapping'iem). W tej sytuacji oprogramowujemy dostęp do bazy danych przy pomocy klasycznego JDBC. Mamy kawałek skryptu SQL który pobiera nam dane z tabeli „klient", w tym pole o nazwie „adres":

```
q = „SELECT … , adres, ... FROM
klient WHERE id = 1";
```

Następnie z wyników zapytania pobieramy zawartość i przepisujemy do zmiennej lokalnej:

```
String adres =
  rs.getString("adres");
```

Dodajmy do tego jeszcze fakt, że pole o nazwie "adres" jest używane jeszcze w kilku innych tabelach, np. „zamówienie", „dostawca", „punkt_odbiorczy", itp.

Przypuśćmy teraz że, po roku działania aplikacji musimy wdrożyć ją w innym kraju, gdzie zwyczajowo używa się dwóch linii na pole adresu. Osoba odpowiedzialna za bazę danych wykonuje zmianę w modelu bazy wszystkich wystąpień pola „adres" na „adres_linia_1" i „adres_linia_2", przygotowuje odpowiednie skrypty migracyjne a następnie przekazuje zmianę do oprogramowania programiście. W większych zespołach najpewniej będzie to inna osoba niż autor oryginalnego kodu, więc najlepsze co może taka osoba zrobić to metodycznie wyszukać wszystkie wystąpienia w kodzie napisów „adres" i cierpliwe je zamieniać na obsługę dwóch nowych pól.

Niestety takie podejście niesie za sobą bardzo poważne zagrożenia:

1. niekompletność: może się zdarzyć że developer, jako że też jest człowiekiem i ma swoje gorsze i lepsze dni może nie znaleźć wszystkich wystąpień, szczególnie jeżeli poprzednik zostawi mu pułapkę w postaci:

```
q = „SELECT … , adr" + „es, ... FROM
klient WHERE id = 1";
```

2. nadmiarowość: developer rozpędzi się i zmieni kod w miejscu, gdzie model bazy nie uległ zmianie.

O ewentualnej pomyłce dowiemy się nie w momencie kompilacji kodu, ale w czasie testów funkcjonalnych lub (bardzo często) po wdrożeniu produkcyjnym - od rozgniewanego klienta.

Development z użyciem automatycznego generowania kodu - przykład

Zastosowanie automatycznej generacji kodu pozwala nam wykorzystać zaletę płynącą z kompilacji kodu przed uruchomieniem. Dzięki temu możliwe jest wcześniejsze dostrzeżenie błędu przez developera i uniknięcie kompromitacji przed klientem.

Automatyczne wygenerowanie kodu dostępu do bazy danych powinno tworzyć

> Developer, nawet mało doświadczony i
> po nieprzespanej nocy, dostanie wszystko na tacy

nam zarówno obiekty reprezentujące rekordy w bazie danych jak i klasy umożliwiające łatwy dostęp do nich. Analogiczne fragmenty kodu przedstawione w powyższym przykładzie przed zmianą modelu będą wyglądać następująco:

```
Klient k = klientDAO.getByPK(1);
String adres = k.getAdres();
```

Po zmianie w modelu pola „`adres`" na „`adres_linia_1`" oraz dodaniu nowego „`adres_linia_2`" i ponownym wygenerowaniu kodu dostępu do bazy danych, wszystkie miejsca gdzie było odwołanie do pola „`adres`", czyli druga linia w powyższym kodzie, przestaną się kompilować. Developer, nawet mało doświadczony i po nieprzespanej nocy, dostanie wszystko na tacy - w szczególności dlatego, że współczesne środowiska developerskie (np. Eclipse) oznaczą nam błąd tak wyraźnie, jakbyśmy napisali w edytorze tekstu „gura" zamiast „góra"...

Ergonomia pracy

Pisząc o środowiskach developerskich nie sposób nie wspomnieć o jeszcze jednym ułatwieniu bardzo podnoszącym efektywność pracy programisty: funkcja automatycznego dopełniania nazw metod, która jest dostępna np. w Eclipse po wciśnięciu kombinacji klawiszy Crtl+Space: wystarczy że w powyższym przykładzie programista napisze „`k.getA`" i użyje dopełniania, a edytor sam dopisze brakujący fragment „`dres()`". W tym wypadku zysk nie jest duży, ale jeżeli mamy w bazie danych kolumnę o nazwie „`care_of_edit_for_one_time_addr`" (to nie żart...), to jej ręczne bezbłędne przepisanie z wydrukowanego schematu bazy danych za pierwszym razem jest w zasadzie niemożliwe.

Implementacja

Najprostszym sposobem generowania gotowego kodu jest znalezienie gotowej biblioteki, która zrobi to za nas. Szukając gotowego rozwiązania trzeba zwrócić uwagę czy:

- jest ono popularne, ma wiele użytkowników i ewentualne wsparcie: forum, listy dyskusyjne;
- będzie ono dostosowane do naszego systemu budowania aplikacji - jeżeli korzystamy np. z ant czy maven'a;
- jest na bieżąco poprawiane i rozwijane.

Wybierając gotową bibliotekę trzeba mieć świadomość, że będzie to dłuższa znajomość, niemal jak małżeństwo... W obszarach tak krytycznych jak O/R-mapping trzeba będzie ją dobrze poznać, zrozumieć, a ewentualny rozwód i wymiana na inny (młodszy...) model będzie bardzo trudna lub wręcz niemożliwa.

Wdrażając bibliotekę automatycznie generującą kod trzeba pamiętać o podstawowych zasadach:

- wygenerowany kod **nie może być umieszczany w repozytorium** (np. CVS, SVN) tylko przy każdej iteracji budowania musi być tworzony ze źródeł (model bazy, pliki konfiguracyjne) - w przeciwnym wypadku mielibyśmy do czynienia z „dwoma źródłami prawdy", które prędzej czy później okażą się sprzeczne;
- wygenerowany kod **nie może być edytowany ręcznie** - to powinno być zrozumiałe, ponieważ przy powtórzeniu iteracji generowania nasze zmiany zostaną zamazane.

> Generowanie kodu powinno stosować się wszędzie gdzie jest to możliwe i/lub są do tego stosowne narzędzia

Obszary zastosowań

Generowanie kodu powinno stosować się wszędzie gdzie jest to możliwe i/lub są do tego stosowne narzędzia. Sugestią do generowania kodu jest istnienie w aplikacji kontraktu zapisanego w postaci jakiegoś pliku. Takim kontraktem może być:

- model bazy danych (kontrakt z architektem systemu),
- model wymiany danych przez WebService zapisany jako WSDL (kontrakt z systemem zewnętrznym),
- pliki konfiguracyjne (kontrakt z projektantem aplikacji)
- klucze plików lokalizacyjnych (kontrakt z osobami tłumaczącymi wersje językowe).

Baza danych (O/R-mapping)

To zastosowanie zostało częściowo opisane w przykładzie wprowadzającym. Jest to najczęściej spotykany obszar gdzie używa się generacje kodu, ponieważ:

- większość aplikacji pisanych w Javie korzysta z bazy danych,
- baza danych ma dużo tabel i kolumn, więc ręczne tworzenie kodu dostępowego (np. przez JDBC) jest bardzo czasochłonne;
- baza danych rozwijanej aplikacji podlega częstym zmianom;
- w większych projektach inna osoba jest odpowiedzialna za zmiany w modelu bazy danych (architekt) a inna za zmiany w kodzie aplikacji (programista), przez co może dojść do niedopowiedzeń i pomyłek przy prowadzaniu zmian;
- baza danych jest krytyczna dla działania aplikacji i nawet drobna literówka w kodzie powoduje wystąpienie sytuacji wyjątkowej (dobrze znany `SQLException`) oraz błąd biznesowy aplikacji.

Powstało wiele bibliotek do komunikacji z bazą danych przez mapowanie rekordów na obiekty (ang. O/R-*mapping*), jednak należy zwrócić uwagę, że niewiele z nich generuje kod Java. Ponadto rozważając wybór takiej biblioteki trzeba uważnie sprawdzić listę wspieranych baz danych, szczególnie jeżeli korzystamy lub mamy zamiar korzystać z mniej popularnego produktu.

Hibernate

Aby pokazać, że nie każda biblioteka O/R-mappingowa spełnia wszystkie warunki stawiane automatycznemu generowaniu kodu zacznę od opisu możliwości Hibernate (http://www.hibernate.org) - chyba najpowszechniej używanej bibliotece do O/R-mapping'u w Javie. Hibernate jest bez wątpienia produktem dopracowanym, bardzo popularnym, intensywnie rozwijanym, posiadającym wręcz doskonałe wsparcie. Niestety, jest jeden problem: strategicznym zamysłem przy tworzeniu Hibernate było zapewnienie mapowania już napisanych klas w Javie na rekordy w bazie a nie generowanie ich automatycznie. Faktem jest, że generowanie obiektów POJO (*ang. Plain Old Java Object* - proste obiekty odpowiadające strukturze rekordu z bazy) jest możliwe przy pomocy dodatkowej biblioteki „z rodziny" Hibernate o nazwie „hibernate-tools" jednakże nie można uniknąć wrażenia, że ten temat ma niski priorytet: rozwój tej biblioteki nie zawsze nadąża za główną biblioteką hibernate-core (co było dobrze widoczne przy przejściu z Java 1.4 na 1.5). Ponadto Hiber-

> Niestety, jest jeden problem

nate nie umożliwia tworzenia zapytania w oparciu o wygenerowane stałe odpowiadające nazwom tabel i kolumn.

Przykład zapytania pobierającego rekord po kluczu głównym oraz rekordów spełniających dwa warunki (nazwa ulicy i status aktywny):

```
Session session = HibernateUtil.
  getSessionFactory().
  getCurrentSession();

Klient k = (Klient) session.
  get(Klient.class, 1);
String adres = k.getAdres();

String q = " FROM " +
  Klient.class.getName() +
  " AS klient " +
  " WHERE klient.adres = :adres" +
  " AND klient.aktywny =" +
  " : aktywny ";
List<Klient> result =
  (List<Klient>)session.
    createQuery(q).
  setString("adres",
    "Marszalkowska").
  setBoolean("aktywny", true).
    list();
```

Jak widać na tym przykładzie, problem ze zmianą pola „`adres`” na „`adres_linia_1`” i „`adres_linia_2`” zostałby przez kompilator znaleziony tylko częściowo, tzn. przestałaby się kompilować linia 3 zawierająca „`k.getAdres()`” (bo nie ma już takiej metody), jednakże zapytanie zapisane w zmiennej `q` ciągle byłoby prawidłowe – je właśnie programista ręcznie musiałby znaleźć i poprawić wszystkie takie wystąpienia.

Torque

Torque (http://db.apache.org/torque/) to produkt bliższy naszym oczekiwaniom: głównym zamysłem jego twórców było właśnie generowanie kodu klas POJO oraz DAO (*ang. Data Access Object* - obiekt dostępu do danych) - wygenerowane przez Torque klasy tego typu mają niezbyt szczęśliwy sufiks „Peer” (*ang. wypatrywać*).

Torque składa się z dwóch części: pierwsza służy do generowania obiektów, druga używana jest w aplikacji do korzystania z nich. Biblioteka ta posiada możliwość tworzenia zapytań w oparciu o wygenerowane stałe odpowiadające nazwom tabel i kolumn, dzięki czemu zapewniamy sobie gwarancję, że zmiana nazw kolumn czy tabel spowoduje błędy kompilacji dzięki którym łatwo obsłużymy ją w kodzie.

Minusem Torque jest fakt, że to biblioteka bardzo stara, utworzona wcześniej jako część większego frameworku Turbine. Jej kod źródłowy jest nie najwyższej jakości a niektóre wzorce bardzo niewygodne, począwszy od statycznych metod, w oparciu o które działają wygenerowane klasy Peer, co praktycznie uniemożliwia mockowanie i testy jednostkowe.

Przykład zapytania pobierającego rekord po kluczu głównym oraz rekordów spełniających dwa warunki (nazwa ulicy i status aktywny):

```
Klient k =
  KlientPeer.retrieveByPK(1);
String adres = k.getAdres();

Criteria crit = new Criteria();
crit.add(KlientPeer.ADRES,
  „Marszalkowska”).
  and(KlientPeer.AKTYWNY, true);
List<Klient> result =
  KlientPeer.doSelect(crit);
```

Jak widać na tym przykładzie, problem ze zmianą pola „`adres`” na „`adres_linia_1`” i „`adres_linia_2`” zostałby przez kompilator znaleziony we wszystkich miejscach, tzn. przestałaby się kompilować linia 2

> Zabezpieczeniem przed takim błędem jest wykorzystanie biblioteki generującej kod Java na podstawie kontraktu

zawierająca „`k.getAdres()`” (bo już nie ma takiej metody) oraz linia 5 zawierająca „`KlientPeer.ADRES`”, ponieważ nie ma już takiej stałej – programista poprawiając błędy kompilacji będzie miał pewność, że nowe zapytania SQL będą poprawne.

XML

W przypadku XML'a mamy podobną sytuację. Załóżmy, że integrujemy się z jakimś zewnętrznym systemem, umawiamy się że dane będą wymieniane jako pliki w formacie XML. Sposobów na wygenerowanie plików XML są dziesiątki, począwszy od System.out.println (...) aż do tworzenia reprezentacji w pamięci obiektów DOM (*ang. Document Object Model*). Podobnie jest z czytaniem plików XML: mamy do dyspozycji reprezentacje DOM, interfejs SAX (*ang. Simple API for XML*) – szczególnie przydatny w przypadku dużych plików które mogą w całości nie zmieścić się w pamięci, itp. Niestety w przypadku, gdy zmieni się kontrakt (zapisany w dokumencie DTD lub Schema), aplikacja będzie nadal kompilować się bez przeszkód a o ewentualnym błędzie dowiemy się dopiero po uruchomieniu aplikacji.

Zabezpieczeniem przed takim błędem jest wykorzystanie biblioteki generującej kod Java na podstawie kontraktu. Przykładem takiej biblioteki jest JAXB (ang. *Java Architecture for XML Binding*, https://jaxb.dev.java.net/). Na podstawie kontraktu zapisanego w pliku schema generuje pliki Java które następnie są wypełniane danymi i przekształcane w postać strumienia XML.

Część definicji w pliku schema mogłaby wyglądać tak:

```
<xsd:element name="klienci"
  type="t:klienci_type" />

<xsd:complexType
  name="klienci_type">
  <xsd:sequence>
    <xsd:element name="klient"
      type="t:klient_type"
      minOccurs="0"
      maxOccurs="unbounded" />
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType
  name="klient_type">
  <xsd:attribute name="id"
    type="t:positive_integer"
    use="required" />
  <xsd:attribute name="imie"
    type="t:text255"
    use="required" />
  <xsd:attribute name="nazwisko"
    type="t:text255"
    use="required" />
  <xsd:attribute name="status"
    type="t:statusKlienta"
    use="required" />
  <xsd:attribute name="adres"
    type="t:text255"
    use="required" />
  <xsd:attribute name="birthdate"
    type="t:dateYYYYMMDD"
    use="required" />
</xsd:complexType>
```

Na podstawie takiej definicji biblioteka JAXB wygeneruje nam obiekty POJO z których możemy skorzystać w przykładowy sposób:

```
KlientType klient1 =
  factory.createKlientType();
[...]
klient1.setStatus(
  StatusKlientaEnum.T);
klient1.setAdres(
  „Marszałkowska”);

KlientType klient2 =
  factory.createKlientType();
[...]
klient2.setStatus(
  StatusKlientaEnum.N);
klient2.setAdres(
  „Al. Niepodległości”);
```

> Większość frameworków i aplikacji korzysta z plików konfiguracyjnych

```
Klienci klienci =
  factory.createKlienci();
klienci.getKlient().add(klient1);
klienci.getKlient().add(klient2);
FileOutputStream out =
  new FileOutputStream(
    new File(
      "/tmp/klienci.xml"));
marshaller.marshal(klienci, out);
```

W wyniku działania programu w pliku klienci.xml znajdziemy treść:

```
<?xml version="1.0"
  encoding="UTF-8"
  standalone="yes"?>
<klienci>
  <klient adres="Marszałkowska"
    birthdate="20090730" id="1"
    imie="Marek"
    nazwisko="Abacki"
    status="T" />
  <klient
    adres="Al. Niepodległości"
    birthdate="20090730" id="2"
    imie="Krzysztof"
    nazwisko="Babacki"
    status="N" />
</klienci>
```

Jak można zauważyć na tym przykładzie, wygenerowane obiekty POJO przypilnują za nas zgodność struktury pliku, zgodność typów argumentów (np. format daty), a nawet tego, czy typy wyliczeniowe są zgodne (klasa `StatusKlientaEnum` zawiera stałe z dozwolonej przestrzeni). Jak łatwo się domyślić, zmiana w pliku schema w definicji klienta pola „`adres`” na „`adres_linia_1`” i „`adres_linia_2`” szybko doprowadzi do błędu kompilacji.

Pliki konfiguracyjne

Większość frameworków i aplikacji korzysta z plików konfiguracyjnych – przykładem może tu być Struts z plikiem `struts-config.xml` zawierający definicje formularzy. Fragment takiego pliku może wyglądać następująco:

```
<form-bean name="klientForm"
  type="org.apache.struts.action.
DynaActionForm">
  <form-property name="imie"
    type="java.lang.String" />
  <form-property name="nazwisko"
    type="java.lang.String" />
  <form-property name="adres"
    type="java.lang.String" />
  <form-property name="aktywny"
    type="java.lang.Boolean" />
</form-bean>
```

Standardowy kod pobierający dane z formularza wygląda podobnie do tego:

```
Klient klient = new Klient();
klient.setImie(form.
  getString(„imie"));
klient.setNazwisko(form.
  getString(„nazwisko"));
klient.setAdres(form.
  getString("adres"));
klient.setAktywny((Boolean)form.
  get(„aktywny"));
```

Jak widać na przykładzie, odwołujemy się do nazw pól formularzy zdefiniowanych w pliku XML poprzez ich nazwy wpisując je „z palca”. Zmiana definicji formularza (np. zmiana pola „`adres`” na „`adres_line_1`” nie spowoduje błędu kompilacji, ale aplikacja przestanie działać.

Wykorzystując proste narzędzie do generowania kodu możemy na podstawie pliku XML wygenerować sobie interfejsy ze stałymi zawierającymi nazwy pól formularza dzięki czemu kod dostępu do tych pól będzie wyglądał następująco:

```
Klient klient = new Klient();
klient.setImie(form.
  getString(klientFormC.imie));
klient.setNazwisko(form.
  getString(klientFormC.nazwi-
sko));
klient.setAdres(form.
  getString(klientFormC.adres));
```

> Nie spotkałem się z gotową biblioteką generującą takie stałe dla frameworku Struts

```
klient.setAktywny((Boolean)form.
  get(klientFormC.aktywny));
```

Po zmianie definicji w pliku XML i przebudowaniu aplikacji powyższy kod przestanie się kompilować i programista będzie musiał nanieść poprawki. Ponadto dużo łatwiej jest znaleźć odniesienia do tego samego elementu: załóżmy że szukamy miejsc w kodzie gdzie są odniesienia do pola „adres” w formularzu klienta. Jeżeli będziemy tekstowo szukać wszystkich wystąpień napisu „„adres”” to znajdziemy również odniesienia do pola „adres” w formularzu kontrahenta, formularzu dostawcy, a także inne przypadkowe wystąpienia niezwiązane z formularzami. Jeżeli jednak poszukamy odwołań do stałej `klientFormC.adres` otrzymamy dokładnie żądany wynik

Nie spotkałem się z gotową biblioteką generującą takie stałe dla frameworku Struts, ponadto każdy framework i aplikacja mają swoje własne formaty plików konfiguracyjnych, to też takie generatory najlepiej napisać samemu. Nakład samodzielnej pracy na napisanie pierwszego takiego generatora w postaci taska Ant’a lub plugina Maven’a jest nie większy niż dzień, kolejne to już kwestia pojedynczych godzin.

Pliki lokalizacyjne

Kolejnym miejscem w którym warto rozważyć generacje kodu są klucze z plików lokalizacyjnych. Załóżmy że mamy plik validations.properties zawierający wiersze:

```
klient_form.adres.required=Pole
„Adres” jest wymagane
klient_form.adres.maxlength=Pole
„Adres” nie może zawierać więcej
niż {0} znaków.
```

Następnie w kodzie aplikacji walidującym dane wprowadzone do formularza mamy następujące odniesienia do kluczy lokalizacyjnych:

```
String adres = form.
  getString(“adres”);
if(“”.equals(adres.trim())) {
  errors.add(“validations”,
  “klient_form.adres.required”);
}
if(adres.trim().length() > 255) {
  errors.add(“validations”,
  “klient_form.adres.maxlength”,
  255);
}
```

Jeżeli ktoś usunie wspomniane wpisy w pliku albo zmieni ich klucze, aplikacja przy próbie prezentacji komunikatu wygeneruje klientowi błąd.

Zastosowanie prostego generatora interfejsów ze stałymi na podstawie pliku propertiesów pozwoli nam zastąpić powyższy kod tak:

```
String adres = form.
  getString(“adres”);
if(“”.equals(adres.trim())) {
  errors.add(validationsC.
    klient_form.adres.required);
}
if(adres.trim().length() > 255) {
  errors.add(validationsC.
    klient_form.adres.maxlength,
    255);
}
```

Usunięcie wpisu w pliku lokalizacyjnym spowoduje że aplikacja przestanie się kompilować a IDE pokaże nam klucze do których się odwołujemy w kodzie a nie ma ich w pliku. Takie rozwiązanie ułatwi nam znacząco również refaktoringi, np. przywoływaną wyżej zmianę pola „adres” na „adres_line_1” i „adres_line_2”.

Nie spotkałem się z gotową biblioteką generującą takie stałe, jednak - podobnie jak w przypadku plików konfiguracyjnych - nakład samodzielnej pracy na napisanie jej

> Generowanie kodu znacząco ułatwia pisanie i dalszy rozwój aplikacji

w postaci taska Ant'a lub plugina Maven'a jest nie większy niż dzień.

Podsumowanie

Generowanie kodu znacząco ułatwia pisanie i dalszy rozwój aplikacji, taki kod jest również nieoceniony w przypadku większych refaktoringów czy szukania wielu odniesień w kodzie do jednego elementu.

Zachęcam każdego do spojrzenia na własną aplikację w poszukiwaniu umieszczonych w kodzie stałych napisowych i zastanowienia się co one reprezentują i czy nie powinny zostać zastąpione stałymi generowanymi. Idealnym stanem byłaby sytuacja gdyby aplikacja zawierała wyłączne odniesienia do generowanych stałych - bez samodzielnego definiowania w kodzie stałych napisowych.

Plusy

- brak konieczności pisania oczywistego i powtarzalnego kodu (np. obiekty POJO reprezentującego wiersze z bazy danych);
- spójność kodu i powtarzalność wzorców;
- błędy znajdowane na etapie kompilacji;
- wykorzystanie auto-uzupełniania w IDE (Crtl-Space w Eclipse);
- łatwiejsze refaktoringi;
- łatwiejsze znajdowanie wielu odniesień do tego samego elementu (Crtl-Shift-G w Eclipse).

Minusy

- uzależnienie od dodatkowych bibliotek zewnętrznych: konieczność aktualizacji zależności, możliwość występowania błędów;
- trudniejsze wprowadzenie nowego developera do projektu;
- bardziej skomplikowany proces budowania aplikacji;
- konieczność tworzenia własnych generatorów do specyficznych elementów (np. pliki konfiguracyjne).

O Autorze

Marek Berkan

programista w firmie e-point SA.

Odpowiada za proces implementacji systemów informatycznych, a następnie za ich techniczne utrzymanie.

Express Killers, cz. IV

Damian Szczepanik

Przykład pierwszy

Na początek krótkie przypomnienie: jaki będzie wynik uruchomienia poniższego kawałka kodu?

```
public class Loader {
  {
    System.out.println(„a”);
  }

  X x;
  static {
    System.out.println(„b”);
  }

  Loader() {
    System.out.println(„c”);
  }

  {
    System.out.println(„d”);
  }

  static class X {

    static {
      System.out.println(„e”);
    }
  }

  public static void main(String[]
args) {
    System.out.println(„f”);
    new Loader();
  }
}
```

Przykład drugi

Napotkawszy poniższy kawałek kodu stwierdzamy (odkrywczo!), że się nie kompiluje, ale rzecz jest prosta i jego poprawienie zajmuje chwilę. Niestety recenzent naszej poprawki stwierdza, że występują następujące ograniczenia w modyfikacji kodu:

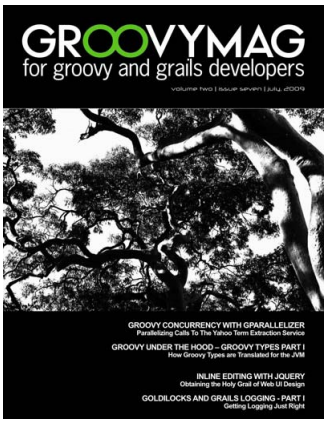
- jest to przykład akademicki i operator warunkowy nie może zostać zmodyfikowany (czyt. metoda main() nie może ulec zmianie)
- klasy A i B są wykorzystywane także w innych projektach i mimo, że nic nie robią, to nie można zmienić ich API (sygnatury metod oraz hierarchii dziedziczenia)

Jak poprawić następujący kod, aby się skompilował, a wynik jego działania nie uległ zmianie?

```
class Parent {
}

class A extends Parent {
  public boolean test() {
    return false;
  }
}

class B extends Parent {
  public boolean test() {
    return true;
  }
}

public class X {
  public static void main(String[]
args) {
    A a = new A();
    B b = new B();
    boolean t = (true ? a :
b).test();
  }
}
```

Recenzja: GroovyMag 07/2009

Krzysztof Konwisarz



GroovyMag to czasopismo tworzone przez użytkowników i entuzjastów Groovy i Grails. Miesięcznik wydawany jest od listopada 2008 jako publikacja pdf.

Groovy Concurrency with GParallelizer

Autor: Jorge Lugo

Artykuł prezentuje trzy podejścia do zrównoleglania aplikacji w Grails. Pierwsze używa asynchronizera w ramach frameworku GParallelizer, kolejne jawnie korzysta z jawowych Executorów, a ostatnie, ponownie korzystając z GParrallelizera, demonstruje użycie aktorów znanych z języka Scala. Wszystko to stworzone zostało na przykładzie Yahoo Term Extraction Service, usługi wynajdywania fraz kluczowych w tekście.

Groovy under the Hood – Groovy Types Part I

Autor: Kirsten Schwark

Groovy ze względu na swoją dynamiczną naturę i kompatybilność ze statyczną Javą używa opcjonalnego typowania. Kiedy najlepiej jawnie zaznaczać typy zmiennych, jakie to ma znaczenie dla prędkości działania i generowanego bajtkodu dla klas, metod i skryptów, kiedy są one sprawdzane, jak mają się typy Groovy do Javy – tego wszystkiego można dowiedzieć się z tej części cyklu.

Inline Editing with jQuery

Autor: Dean Del Ponte

Grails nie doczekało się jeszcze pluginu AJAX, który oferowałby tabele danych z możliwością edycji w wierszu. Choć taki komponent ma pojawić się w kolejnych odsłonach GrailsUI, już dziś możemy skorzystać z rozwiązania prezentowanego przez Deana Del Ponte. Przy użyciu wtyczki biblioteki JQuery oraz jej pluginów jquery-in-place-editor i jquery-ui-datepicker tworzy on asynchronicznie edytowalną tabelę z obsługą dat.

Goldilocks and Grails Logging – Part I

Autor: Robert Fischer

Zaczynając od uzasadnienia dlaczego logowanie jest ważne i kilku rad na temat tego co logować autor przedstawia standardowy system odpowiedzialny za to zadanie w Grails: Log4J. Pokrótce omówiona zostaje budowa Log4J (loggery, kategorie, poziomy i możliwe appendery) oraz jego wykorzystanie i sposób konfiguracji w Grails.

Plugin Corner – Jabber Plugin

Autor: Dave Klein

Wtyczka umożliwia wysyłanie i odbieranie wiadomości do i z serwera Jabbera. Jej działanie poznajemy tworząc prosty komunikator internetowy.

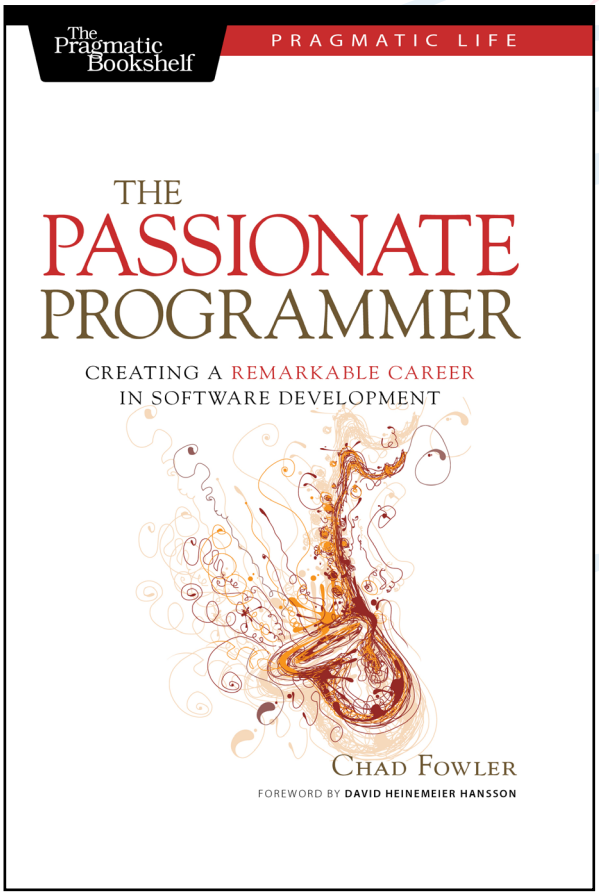
O autorze

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

Recenzja: The Passionate Programmer

Marcin Dąbrowski

Książka, którą chciałbym dzisiaj zrecenzować to kolejna z wielu pozycji, bardzo dobrego moim zdaniem wydawnictwa, The Pragmatic Bookshelf. Jest to firma, która jednocześnie wydaje bardzo ciekawe pozycje oraz zajmuje się prowadzeniem szkoleń. Sami o sobie mówią "We're here because we want to improve the lives of developers." (w wolnym tłumaczeniu "Jesteśmy tu gdyż chcemy poprawić życie developerów"). Książki z cyklu Pragmatic Bookshelf ukazują inne spojrzenie na problemy spotykane w dziedzinie IT. Nie są to jednak książki, które bezpośrednio traktują na temat języka czy technologii.



„The passionate programmer – creating a remarkable career in software development" to pozycja, którą mógłbym śmiało określić, jako "świeża bułeczka" gdyż jej premiera miała miejsce w maju 2009 roku. W obecnej chwili nie doczekała się jeszcze polskiej wersji. Jest to druga edycja książki pt. „My Job Went to India: 52 Ways To Save Your Job". Nawiasem mówiąc tytuł został zmieniony gdyż zdaniem autora ten pierwszy kojarzył się z książką, która powie czytelnikowi jak w efektywny sposób nie stracić swojej pracy. Innymi słowy, co robić, aby nie być zupełnie do "kitu". Sam autor ujął to następująco "And you don't win at life by trying not to suck. Fortunately, the content of the book has never been about trying not to suck. I can't think that way, and neither should you."

TPP w swoim zamierzeniu ma uczyć jak być wyjątkowym albo raczej jak zdać sobie z tego sprawę gdyż wg. autora wszyscy już tacy jesteśmy. Twórcą tej pozycji jest, Chad Fowler, dawniej profesjonalny saksofonista a w obecnej chwili programista, inżynier oprogramowania oraz jak widzimy po przemyśleniach zawartych w książce całkiem niezły myśliciel. Chad posiada również niezrównane poczucie humoru, co jeszcze bardziej w zwiększa przyjemność wynikającą z czytania. Pozycja podzielona jest na pięć niezależnych rozdziałów traktujących o pięciu różnych tematach. Wszystkie rozdziały zawierają bardzo dużą ilość spostrzeżeń oraz rad, których podstawą jest doświadczenie autora.

W części pierwszej zatytułowanej "Choosing Your Market" autor tłumaczy, iż bardzo często życie programisty w firmie jest określone przez czynniki, które nie wynikają bezpośrednio z chęci czy świadomości tej osoby. Developer jest uprzedmiotawiany, który w wyniku zbiegu okoliczności trafia w jedno miejsce gdzie wykonuje określoną pracę, po czym przerzucony zostaje do innego projektu gdzie "bezmyślnie" wykonuje coś kompletnie innego. Autor stwierdza z całym przekonaniem, iż

taki stan rzeczy jest negatywny i nie może występować w życiu osoby, której zamiarem jest odniesieniu sukcesu w swojej karierze. Chad stawia przed czytelnikiem określone pytania oraz stara się wskazać odpowiednią ścieżkę. Z pierwszej części dowiadujemy się także, iż błędem jest brak ryzyka, powinniśmy stać się specjalistami oraz najgorsze, co możemy zrobić to "zamknięcie" się w jednej tematyce.

W drugiej części "Investing in Your Product" czytelnik znajdzie pożyteczne wskazówki mówiące o samorozwoju oraz o preferowanej drodze, którą należy wybrać, aby odnieść sukces. Jedna z rad to "Give a man a fish, feed him for a day. Teach a man to fish, feed him for a lifetime." (Daj człowiekowi rybę, nakarmisz go na jeden dzień. Naucz człowieka łowić, a nakarmisz go na całe życie). Czytelnik dowiaduje się, iż pozostawanie w jednej firmie przez długi czas to nic innego jak powodowanie stagnacji. Ludzie powinni nieustannie zdobywać nowe doświadczenie oraz umiejętności. Chad tłumaczy, iż kluczowymi wartościami jest zdobycie mentora oraz staniem się nim, wyjaśnia jak ważna jest praktyka oraz jak złe jest koncentrowanie się na najpopularniejszych w danej chwili technologiach.

W części trzeciej "Executing" dowiadujemy się jak powinno wyglądać motywowanie się do pracy, jak "czytać w myślach" swoich pracodawców czy też, iż uaktualnianie istniejącego już kodu wcale nie jest takie straszne jak się wydaje. W tym rozdziale autor przedstawił także swoje zdanie na temat molocha jakim jest korporacja.

Czwarta część "Marketing… Not Just for Suits" poświęcona została temu, co należy zrobić, aby zostać zauważonym. Co więcej jak postępować, aby być postrzeganym, jako bardzo dobry, odpowiedzialny, godny zaufania pracownik . Autor poprzez przedstawienie błahego powiedzenia "If a tree falls in the forest but nobody is there to hear it fall, did it make a sound?" (Jeżeli drzewo się przewróciło w środku lasu, ale nikogo nie było kto mógłby to usłyszeć to czy wydało jakikolwiek dźwięk?), nawiązuje do tego, iż osoba, która nie jest zauważana przez otaczających ludzi nagle znacznie zwiększy swoje umiejętności to czy ktokolwiek to zauważy? Chad stale stara się przekonać czytelnika, iż "prawda" leży u jego stóp, lecz pozostaje niezauważona. Autor podaje także parę "drobnych rad", np. marketing nie jest tylko dla ludzi ubranych w garnitury, czy chociażby jak prowadzić swoją stronę, (blog) aby budować własny wizerunek.

Ostatnia już, piąta część książki poświęcona została "This part will show you how to avoid becoming a one-hit wonder." (ta część pokaże Ci jak uniknąć zostania jednorazowym cudem). Na początku autor stwierdza, iż "You are not your job" (nie jesteś swoją pracą) i doradza tym samym, aby nie czuć się zbyt pewnie w swoim miejscu pracy. W dalszym ciągu czytelnik dowiaduje się, iż cel jego pracy w wielkiej firmie tak naprawdę nigdy nie ma końca. Projekt w korporacji zawsze pozostaje żywy i nigdy tak naprawdę nie ma celu, zawsze pozostaje modyfikacja istniejącego rozwiązania. Chad tłumaczy także dlaczego bardzo trudno jest zaobserwować ciągle postępujące zmiany. Posługuje się tu dość ciekawym porównanie do tycia. Osoba, która stopniowo zwiększa swoją wagę tak naprawdę nie zdaje sobie z tego sprawy gdyż widzi się codziennie, należy zatem zastosować odpowiednią technikę aby to uwidocznić.

Książka "The Passionate Programmer" jest godną polecenia pozycją wprowadzającą olbrzymi zasób rad, dowodów oraz pożytecznych przykładów. Z książki Chada Fowlera czytelnik może się dowiedzieć jak wygląda świat IT obserwowany z innej perspektywy. Podczas analizy oraz lektury starałem się doszukiwać zarówno pozytywnych jak i negatywnych stron. Prawda jest jednak taka, iż książka jest naprawdę dobra i z pewnością warto ją przeczytać. Polecam.

Express Killers, cz. IV - odpowiedzi

Damian Szczepanik

Przykład pierwszy

Wynik będzie następujący:

b – po załadowaniu klasy wykonane zostaną bloki statyczne,

f – uruchomienie metody main() spowoduje wykonanie umieszczonych w niej instrukcji,

a,d – stworzenie obiektu klasy Loader spowoduje wykonanie kolejnych (wg ich pozycji w klasie) bloków inicjalizacyjnych klasy,

c – wywołanie właściwego konstruktora klasy.

Litera e nie zostanie wydrukowana, gdyż klasa wewnętrzna statyczna jest ładowana w momencie jej pierwszego wykorzystania. W powyższym przykładzie nie jest taka klasa tworzona, zatem nie zostanie także załadowana. Fakt, że jest ona atrybutem klasy Loader niczego nie zmienia, gdyż jest to deklaracja typu, sam obiekt nie jest jednak tworzony.

Przykład drugi

Operator warunkowy nie można użyć z zaproponowaną składnią, gdyż kompilator:

- będzie szukał pierwszej wspólnej klasy nadrzędnej dla klas A i B,
- sprawdzi, czy odnaleziona klasa posiada metodę, która jest wywoływana, a jeśli jej nie znajdzie, zwróci błąd

Kompilator na etapie kompilacji nie potrafi określić, jaki będzie wynik działania operatora warunkowego (mimo, że w tym przypadku optymalizacja jest w stanie określić zwracany typ), stąd musi być pewnym, że w obu przypadkach wywołanie metody test() będzie możliwe.

Mimo, że w tym przypadku, oba typy implementują metodę test(), to kompilator, jak już wspomniano, poszukiwania rozpocznie od pierwszej wspólnej klasy nadrzędnej, a zatem Parent, który takiej metody nie implementuje.

Co zatem można zrobić w powyższym przypadku? Skoro operator warunkowy nie może ulec zmianie, to znaczy, że należy zapewnić kompilator, że klasa Parent posiada metodę test(), można to zrobić w następujący sposób:

- klasę Parent zamienić na klasą abstrakcyjną i dopisać do niej metodę abstrakcyjną test(), co jednak wiąże się z ograniczeniem takim, że ewentualne wywołanie new Parent() nie będzie poprawne
- dodać metodę test() do klasy Parent, która to metodą będzie rzucać jeden z wyjątków mówiących, że nie jest zaimplementowana, co oznacza, że klasa pochodna powinna ją przesłonić

Najlepszym rozwiązaniem byłoby oczywiście usunięcie operatora warunkowego i zastąpienie go blokiem if-else, w takim przypadku modyfikacja klasy Parent byłaby zbędna.

Mistrz Programowania: Refaktoryzacja, cz. I

Mariusz Sieraczkiewicz

Dzisiaj niespodzianka. 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

Rozdział 3

Jak używać refaktoryzacji do tworzenia w pełni obiektowych aplikacji

Nawyki skutecznej refaktoryzacji

Skuteczne programowanie to niezwykle cenna umiejętność. Przedstawione w poprzednim rozdziale proste techniki mogą całkowicie odmienić sposób pisania kodu. Zawsze zadawałem sobie pytanie: „Skoro jest to takie proste, dlaczego mało kto je stosuje?"

Jest kilka powodów.

Aby zrozumieć, musisz doświadczyć

Z pewnością wiele razy słyszałeś przysłowie „Co nagle, to po diable". Słyszymy je po raz pierwszy jako dzieci. Czasem nawet nie do końca je rozumiemy. Później słyszymy je wiele razy i traci ono dla nas znaczenie. Zużywa się. Aż przychodzi taki czas, kiedy w naszym życiu dzieje się coś takiego, co daje dotkliwie odczuć na własnej skórze znaczenie tego zdania. Wtedy przychodzi czas na prawdziwe zrozumienie.

Tak samo jest z technikami związanymi z inżynierią oprogramowania. Musisz ich doświadczyć, aby odnaleźć ich głęboki sens. Dlatego ta książka nie ma na celu cię nauczyć, jak refaktoryzować. Ma pobudzić cię do działania.

Ważne

Aby zrozumieć, musisz doświadczyć.

To nie techniki działają — to ludzie działają

Nawet najlepsza technika, jeśli nie jest stosowana, jest nic nie warta. Kiedyś spędziłem miesiąc nad tym, aby nauczyć się bezwzrokowo pisać na klawiaturze. Mówiąc szczerze, nie był to łatwy miesiąc. Spędziłem długie godziny, walcząc z dotychczasowymi przyzwyczajeniami. Niejednokrotnie chciałem porzucić ten pomysł, uważając, że stary sposób pisania był wystarczająco dobry. Dzisiaj już wiem, że stary nawyk był bardzo niewydajny — spowalniał wielokrotnie moją pracę. Jeśli chodzi o refaktoryzację (czy dowolną inną umiejętność), jest dokładnie tak samo. Jeśli zaangażujesz się wystarczająco, na pewno przyniesie to ogromne korzyści.

Ważne

Postanów już dziś, że przez najbliższy miesiąc będziesz stosował regularnie poznane tutaj techniki, a po tym czasie sam ocenisz efekt. Ja już wiem, jaki on będzie . . .

Racjonalizacja

Jest to prosty mechanizm psychologiczny, który powoduje, iż znajdujemy powody, aby czegoś nie robić. Im trudniejszy do osiągnięcia cel, tym bardziej stajemy się wyrafinowani w znajdowaniu racjonalizacji. Kilka przykładów?

Potencjalnie chciałbyś zmienić pracę, ale nie robisz tego, bo znajdujesz powody:

- Gdzieś indziej jest tak samo.
- Tutaj nie jest tak źle.
- Mam tu zbyt wielu znajomych.
- Być może mnie niedługo awansują.

Potencjalnie chciałbyś zarabiać więcej (a któżby nie chciał!), ale nie idziesz do szefa na rozmowę, bo znajdujesz powody:

- Ostatnio miałem kilka wpadek i nie jest to najlepszy moment.

- Teraz firma ma kłopoty finansowe, więc może później.
- W zasadzie wystarcza mi to co mam, sporo znajomych ma gorszą sytuację.

Potencjalnie chciałbyś wprowadzić w życie refaktoryzację, ale nie robisz tego, bo znajdujesz powody:

- To zajmie zbyt wiele czasu.
- W moim projekcie to nie zadziała.
- Szef i tak się nie zgodzi.
- Tu jest taki bałagan, że to nie ma sensu.
- Od następnego projektu.

Przypomnij sobie kilka ostatnich sytuacji, w których znajdowałeś powody, z których miałbyś czegoś nie robić. Zapisz je poniżej.

Moje racjonalizacje

Przekonania

Jest to jeden z najsilniejszych mechanizmów, który determinuje nasz sposób myślenia. Przekonanie jest to nasza subiektywna opinia na dany temat, co do której jesteśmy pewni, że jest obiektywna. Na przykład:

- Żeby zdobyć dobrze płatną pracę, trzeba mieć znajomości.
- Mój sposób programowania jest wystarczająco dobry i nie wymaga zmian.
- Żeby być dobrze opłacanym programistą, muszę znać jak najwięcej technologii i posiadać certyfikaty.

Największy problem z przekonaniami jest taki, że zazwyczaj ich sobie nie uświadamiamy. Poświęć trochę czasu i zastanów się, jakie masz przekonania na temat programowania i swojej pracy. Zapisz je poniżej.

Moje przekonania

Mam nadzieje, że powyższe przykłady sprowokowały cię do przemyślenia pewnych spraw. Jak możesz je wykorzystać, aby wprowadzić w życie techniki refaktoryzacji?

Co nagle, to po diable

Ważne

Daj sobie czas. Nie będziesz w stanie wdrożyć refaktoryzacji w życie, jeśli to, co jest opisane w tej książce, przejrzysz powierzchownie. To na pewno nie zadziała. Jeśli zdecydowałeś się czytać dalej — rób to powoli, analizuj poniższe przykłady i eksperymentuj z nimi w swoim ulubionym środowisku programistycznym. Mógłbym przytoczyć wiele przykładów, kiedy próbowałem zrobić coś na skróty, a zajęło to dużo więcej czasu lub nic z tego nie wyszło. W tej książce znajduje się esencja — 20% najważniejszych rzeczy, tych najbardziej użytecznych.

Długie metody nie są wcale dobre

Wróćmy do realizowanego przykładu — aplikacji do znajdowania tłumaczeń słów za pomocą słownika internetowego. Mamy już stworzony szkielet i kilka ważniejszych refaktoryzacji za sobą. Wróćmy do metody `WebDictionary.searchWord`, która pełni jedno z najważniejszych zadań — realizuje wyszukiwanie tłumaczonych słów. Jak już wspomniałem, obecne rozwiązanie ma poważną wadę — metoda jest bardzo długa.

Pierwszym pomysłem może być próba wydzielenia mniejszych metod na bazie metody `searchWord`. Jeśli jednak przyjrzymy się jej bliżej, to zauważymy, że nie jest to

takie proste zadanie.

```
private void searchWord(String command) {
    lastSearchWords.clear();

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

    try {
        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()));

        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");

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

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

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

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

Refaktoryzacja: Zastąpienie metody przez obiekt reprezentujący metodę

Metoda `searchWord` składa się z kilku podczynności, takich jak inicjacja zmiennych, odczyt strony, analiza pojedynczego wiersza, zapamiętanie znalezionych tłumaczeń. Podczynności te współdzielą wspólny stan — jest to obiekt klasy `BufferedReader` dający dostęp do przetwarzanej strony HTML. Ponadto metoda `searchWord` zawiera wiele niezależnych zmiennych lokalnych, co utrudnia, a w zasadzie uniemożliwia prostą refaktoryzację typu *Wydzielenie metody*.

Z drugiej strony funkcjonalność wyszukiwania wyrazów daleko wykracza poza odpowiedzialność klasy `WebDictionary`, która zajmuje się obsługą użytkownika aplikacji. Warto by wydzielić ją do osobnej klasy. Nazwijmy ją `SearchWordService`. Posunięcie polegające na wydzieleniu zawartości metody realizującej złożone przetwarzanie do osobnego obiektu nazywamy *Zastąpieniem metody przez obiekt reprezentujący metodę*. Po tym kroku kod metody `searchWord` będzie wyglądał następująco:

```
    private void searchWord(String command) {
        SearchWordService searchWordService
            = new SearchWordService( command );
        lastSearchWords = searchWordService.search();
    }
```

Klasa SearchWordService będzie wyglądać 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;

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

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

    // ... ta część kodu bez zmian

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

    // ... ta część kodu bez zmian

        return result;
    }

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