Java eXPress

Numer 2 (2) Grudzień 2008

OSGi Declarative Services

oraz

Maven 2, cz. I

Google Data API

Alternatywny kurs Javy, cz. II

Nie tylko refaktoring, cz. II

Express killers, cz. I

new Double();

To już drugi numer czasopisma JAVA exPress. Mam nadzieję, że będzie Wam się podobał co najmniej tak samo jak pierwszy. Tym razem oprócz kontynuacji alternatywnego kursu Javy i artykułu o refaktoringu będziecie mieli okazję przyjrzeć się Declarative Services w OSGi, rozpocząć naukę Maven 2 i nauczyć się korzystać z zasobów aplikacji spod znaku „G". A dla lubiących łamigłówki Damian przygotował niezła dawkę zagadek Javowych.

Chciałem w tym miejscu podziękować naszym 2 patronom – firmie e-point i iSolution. Dzięki ich pomocy możemy spokojnie myśleć o kolejnych numerach JAVA exPress.

Dziękuję także wszystkim, którzy wspierają JAVA exPress. Zarówno autorom artykułów, jak również tym, co wspomagają nas finansowo. W szczególności wspierają nas: Robert Sajdok, Mateusz Gałek, Dawid Bielecki, Michał Bodzek, Krzysztof Wojtas, Łukasz Dziedziul i cała masa „no name".

Szczególne podziękowania należą się Jakubowi Sosińskiemu, który podjął się trudu zmienienia oprawy graficznej czasopisma (już możecie podziwiać jego dzieło – logo dWorld, COOLuary i pomoc przy logo JAVA exPress). Kolejny numer to dopiero będzie rewolucja graficzna.

Zdjęcie na okładce pochodzi z galerii Roberta Scotta. Możecie je znaleźć pod adresem http://www.flickr.com/photos/rsdreamphotos/2081366879/.

Przy okazji – udanych mikołajków...

Pozdrawiam,
Grzegorz Duda

Plan podróży

DEVOXX™
the java™ community conference

Już w poniedziałek zaczyna się w Belgii największa konferencja Javowa w Europie - Devoxx (było Javoxx, było Javapolis). JAVA exPress jest jednym z sześciu patronów medialnych tego wydarzenia. Jest to nasz pierwszy patronat, a od razu tak znamienity.

Podobnie jak w poprzednim roku, wszystkie bilety zostały wyprzedane dużo przed konferencją. A więc kryzys, nie kryzys, kształcić się trzeba...

Od poniedziałku 3200 developerów będzie od rana do późnego wieczora przyglądać się różnym aspektom Javy i nie tylko. A po konferencji liczne spotkania w knajpach i nieprzespane noce. To będzie ciężki tydzień, ale jakże owocny.

Po konferencji szukajcie relacji na http://dworld.pl.

Więcej informacji znajdziecie na stronie konferencji: http://devoxx.com.

COOLuary

1 otwarte dyskusje o Javie - Kraków, 24 stycznia 2009

Pamiętacie JavaUnderground na JDD w tym roku? Podobało Wam się?

Chciałem Wam zaproponować kolejną nowość na naszym polskim rynku konferencji o Javie. Mowa tutaj o UnConference, czy inaczej Open Space Conference. To konferencje organizowane zgodnie z metodą Open Space Techology. W skrócie, to uczestnicy decydują o agendzie konferencji i mają możliwość aktywnego udziału w konferencji poprzez włączanie się do dyskusji. Taką też formę ma konferencja organizowana przez najbardziej znany podcast Javowy - Java Posse Roundup.

Taką też formę będą miały COOLuary - konferencja, która odbędzie się w sobotę, 24 stycznia w Krakowie.

Dlaczego UnConference? Większość osób wracających z dużych konferencji (także tych w Polsce) twierdzi, że najlepsze na konferencji były… przerwy. Tak, tak. Przerwy. To na nich można wymienić doświadczenia i podyskutować na nurtujące nas tematy.

COOLuary, to w ponad 70% rozmowy kuluarowe znane z przerw na konferencjach organizowanych w tradycyjnym formacie.

Więcej informacji na stronie http://dworld.pl/cooluary/

Konferencja 4Developers to największe przedsięwzięcie tego typu w Polsce - 700 uczestników, 4 równoległe ścieżki (Java, .NET & C#, Zarządzanie projektami IT oraz Języki specjalizowane) i ponad 30 prelegentów.

Podczas sesji Java wystąpi m. in. Adam Bien, Java Champion, członek Netbeans Dream Team, który z Javą i J2EE pracuje od samego początku. W Polsce znany przede wszystkim ze świetnego wystąpienia na JDD08. Gościem honorowym sesji .NET & C# będzie Ted Neward, niezależny konsultant pracujący dla największych korporacji, autorytet w dziedzinie integracji Javy i .NET. Sesję Zarządzanie projektami IT otworzy prezentacja jednego z najgłośniejszych projektów ostatniego roku, gry Wiedźmin. W ramach ścieżki Języki specjalizowane, uczestnicy konferencji będą mogli wysłuchać wykładów na temat programowania baz danych, PHP, Ruby i systemów wbudowanych.

Call for papers ciągle trwa!

Wymyśl i zaprojektuj logo 4Developers i wygraj darmową wejściówkę na konferencję oraz odtwarzacz mp4!

Dowiedz się więcej na stronie konferencji http://4developers.org.pl/

JavaFx

W czwartek, 4 grudnia, SUN wypuścił JavaFX w wersji 1.0. Czy to będzie pogromca FLEXa? Czy znajdzie swoje miejsce w torcie zwanym RIA? A może już jest za późno na ratowanie apletów? Do tej pory byłem nieco sceptycznie nastawiony do tego dzieła spod znaku SUN. Niemniej jednak prezentacje i dema robią wrażenie. Pora przyglądnąć się temu z bliska. A może uda się coś przygotować an COOLuary?

Więcej informacji o JavaFX na stronie http://javafx.com/

Kubek Kawy - czyli alternatywny kurs Javy, cz. II

Bartek Kuczyński

Artykuł jest modyfikacją kursu z portalu http://4programmers.net/java/Podstawy_Javy

Siłą komputerów nie jet ich inteligencja, bo jej nie mają. Generalnie komputer potrafi wykonać kilka podstawowych operacji logicznych takich jak porównanie, koniunkcja, alternatywa, alternatywa wykluczająca czy przypisanie. Nawet dodawanie, które jest realizowane w systemie dwójkowym, jest ciągiem kilku prostych operacji logicznych. Prawdziwą siłą komputerów jest to, że potrafią wykonywać miliardy takich operacji na sekundę.

Poprzednią część kursu zakończyliśmy na modyfikatorach metod, klas i pól. Dziś zajmiemy się kolejną partią teorii. Porozmawiamy o tym jakie są typy w Javie. Jak sterujemy programem i jak przechowywać dane.

Typy Proste i Obiekty

W pierwszej części pojawiła się definicja Javy. Było w niej magiczne pojecie „zorientowany obiektowo". Jak pisałem oznacza to, że w Javie występują zarówno obiekty jak i typy proste. Typów prostych jest 8:

- byte
- short
- int
- long
- float
- double
- char
- boolean

Wszystkie inne dane w Javie traktowane są jako obiekty. Należy pamiętać, że typy proste nie występują w programach jako byty niezależne. Pojawiają się jako pola lub zmienne w obiektach i metodach. Warto zaznaczyć, że od wersji 1.5 Java oferuje mechanizm "pudełkowania" (autoboxing) dla typów prostych. Zasada działania tego mechanizmu jest teoretycznie dość prosta. Wszelkie typy proste są w procesie kompilacji zamieniane na odpowiadające im obiekty. Mechanizm zamieniający kompilatora dba jednak o to, żeby skompilowany kod zachował zgodność wstecz. Obiekty odpowiadające typom prostym mogą zostać "wypakowane" (unboxing) do typu prostego.

Każdy obiekt i typ prosty jest reprezentowany przez zmienną.

Przykłady deklaracji zmiennych typów prostych:

```
int a;
//deklaracja zmiennej a typu całkowitego
char b;
//deklaracja zmiennej b która zawiera znaki Unicode
```

Równoczesna deklaracja i inicjacja zmiennych:

```
int a = 3;
char b = 'c';
```

Oprócz typów prostych w Javie istnieją typy obiektowe. Są to np:
Button, Panel, Label, Okno - nazwa klasy stworzonej przez użytkownika

Deklaracja zmiennych typu obiektowego:

```
Button b = new Button();
```

Typy proste i obiekty mogą zostać najpierw zdefiniowane, a zainicjowane dopiero później. Zmienne, które są polami obiektów zawsze są inicjowane z wartością domyślną. Zmienne lokalne (zwane też automatycznymi) muszą zostać jawnie zainicjowane. Przykład:

```
class Foo {
  int zmienna;
  // zmienna będzie miała domyślna wartość 0
  // w momencie stworzenia obiektu klasy
  Object obj;
  // zmienna będzie miała domyślna wartość null
  // w momencie stworzenia obiektu klasy

  public void bar() {
    /*
     * zmienna musi zostać jawnie zainicjowania,
     * inaczej kompilator zwróci błąd
     *  variable may not been initialized
     *  jeżeli będziemy chcieli jej użyć
     */
```

```
    int zmienna2;
    zmienna2 = 1;
    System.out.println(zmienna); // 0
    System.out.println(zmienna2); // 1
  }
}
```

Od wersji 1.5 wprowadzono nowy typ **enum**. Jest to rozwiązanie problemów pojawiających się, gdy stosujemy konstrukcje podobne do poniższej:

```
class Klasa {
  // definiujemy jakieś stałe - flagi
  public static final String ADD = "dodaj";
  public static final String LIST = "Wypisz";
  public static final String DELETE = "Usuń";
  // ....
}

// później w kodzie:

class A {
  public void metoda(String akcja) {
    if (akcja.equals(Klasa.ADD)) {
      // ...
    } else if (akcja.equals(Klasa.LIST)) {
      // ...
    } else if (akcja.equals(Klasa.DELETE)) {
      // ...
    } else {
      // ...
    }
  }
}
```

Takie rozwiązanie ma wiele wad. Do najważniejszych należą:

- możliwość użycia złej nazwy, słaba typowalność nazw,
- skrajna nierozszerzalność, użycie słowa kluczowego final wyklucza zmianę klucza. Przypadkowa próba nadpisania klucza powoduje błędy kompilacji.

Zamiast takiego podejścia należy używać typu **enum**:

```
enum Akcje {
  ADD,
  LIST,
  DELETE;
}
```

Problematyka typów wyliczeniowych (enum) jest złożona. Na chwilę obecną musimy zapamiętać tylko tyle, że są.

Operatory

W Javie istnieje kilka operatorów. Możemy je podzielić na trzy główne grupy. Pierwszą stanowią operatory matematyczne. Drugą operatory logiczne, w tym operatory porównania, a trzecią operatory bitowe.

Operatory matematyczne

W Javie zefiniowane są następujące operatory matematyczne:

- Dodawanie (+)
- Odejmowanie (-)
- Mnożenie (*)
- Dzielenie (/)
- Reszta z dzielenia (%)
- Wynik dzielenia z resztą (\)

Dodatkowo zdefiniowane są jeszcze dwa operatory inkrementacji i dekrementacji:

- Inkrementacja - zwiększenie o 1 (++)
- Dekrementacja - zmniejszenie o 1 (--)

Powyższe operatory mogą stać zarówno po nazwie zmiennej, jak i przed. W pierwszym przypadku mówimy o postinkrementacji lub postdekrementacji. Oznacza to, że zmienna jest zwiększana lub zmniejszana po użyciu. W drugim przypadku mówimy o preinkrementacji lub redekrementacji i odpowiednia operacja jest wykonywana przed użyciem zmiennej.

Wszystkie operatory z tabeli mają też odpowiedniki mające sens "wykonaj działanie i przypisz". Za przykład niech posłuży nam operator dodawania:

```
int a = 1;
a += 1; // a ma wartość 2
```

Operatory logiczne

Komputer jest maszyną "myślącą" w sposób całkowicie logiczny. Na podstawowym poziomie komputer wykonuje najprostsze operacje logiczne. Zasada te ma odwzorowanie w operatorach logicznych oraz w operatorach porównania.

- Operacja LUB (||)
- Operaacja I (&&)
- Negacja (!)
- Większy niż (>)
- Mniejszy niż (<)
- Większy lub równy (>=)
- Mniejszy lub równy (<=)
- Różny od (!=)

Jak widać w powyższej tabeli zabrakło ważnego operatora. Operator równy w języku Java sprawia osobą początkującym wiele problemów. Wynika to ze specyfiki języka. Zapis:

```
a == b
```

ma różne znaczenie w zależności od tego czy odnosi się do typów prostych, czy też obiektów. Jeżeli porównujemy w ten sposób dwie zmienne typu prostego, to operator działa "intuicyjnie". Przykład:

```
int a = 1;
int b = 1;
System.out.println(a == b); // zwróci true
```

W odniesieniu do zmiennych obiektowych zasada ta jest inna. Operator == oznacza **Identyczność**, a nie równość:

```
Integer a = new Integer(1);
Integer b = new Integer(1);
Integer c = a;
// c jest referencją do tego samego obiektu
// na stercie co a
System.out.println(a == b);
// zwróci flase, a i b to różne obiekty
// tej samej klasy
System.out.println(a == c);
// zwróci true, a i c to ten sam obiekt
```

Jeżeli chcemy porównać dwa obiekty, należy użyć metody `equals()`:

```
Integer a = new Integer(1);
Integer b = new Integer(1);
System.out.println(a.equals(b)); // zwróci true
```

Ostatnim operatorem porównania dla obiektów jest słowo kluczowe **instanceof**. Przykład:

```
Integer a = new Integer(1);
// dziedziczy po Number, a ta po Object
System.out.println(a instanceof Integer);
// zwróci true a jest klasy Integer
System.out.println(a instanceof Object);
// zwróci true, klasa Integer dziedziczy po Object,
// a jest klasy Object
System.out.println(a instanceof Serializable);
// zwróci true, klasa Number implementuje
// Serializable, a jest Serializable
```

Prawie 90% błędów popełnianych przez początkujących programistów związanych z warunkami logicznymi związane jest z niezrozumieniem i pomyleniem operatora == i metody equals().

Operatory bitowe

Komputery pracują na bitach. Operacje na nich w wielu przypadkach pozwalają na znaczne przyspieszenie obliczeń.

- Operacja LUB (|)
- Operacja I ($)
- Negacja (~)
- Operacja XOR (^)
- Operacja przesunięcia w prawo (>>)
- Operacja przesunięcia w lewo (<<)
- Operacja przesunięcia w prawo z wypełnieniem zerami (>>>)

Instrukcje sterujące

W momencie, gdy chcemy, aby program dokonał wyboru jednej z dróg na podstawie prawdziwości jakiegoś warunku logicznego możemy użyć jednej z dwóch instrukcji sterujących. Jeżeli chcemy, aby o drodze decydował jakiś warunek logiczny, to używamy instrukcji **if/else**. Jeżeli chcemy, aby wybór został dokonany na podstawie stanu obiektu możemy użyć przełącznika - **switch**.

Instrukcja if / if else

Najprostszą instrukcją warunkową jest instrukcja if:

```
if (warunek_logiczny) {
  // instrukcje wykonane
  // jeżeli warunek jest PRAWDZIWY
}
```

odmianą tej instrukcji jest instrukcja if else:

```
if (warunek_logiczny) {
  // instrukcje wykonane
  // jeżeli warunek jest PRAWDZIWY
} else {
  // instrukcje wykonane
  // jeżeli warunek jest FAŁSZYWY
}
```

instrukcje można zagłębiać:

```
if (warunek_logiczny) {
  if (warunek_logiczny2) {
    // instrukcje wykonane
    // jeżeli warunek jest PRAWDZIWY
  }
    }
```

oraz dokonywać wielokrotnego wyboru

```
if (warunek_logiczny) {
  // instrukcje wykonane
  // jeżeli warunek1 jest PRAWDZIWY
} else if (warunek_logiczny2) {
  // instrukcje wykonane
  // jeżeli warunek2 jest PRAWDZIWY
} else {
  // instrukcje wykonane
  // jeżeli warunek1 i warunek 2 są FAŁSZYWE
}
```

Operator trójargumentowy ? :

Jeżeli chcemy, aby zmienna przyjęła jakąś wartość w zależności od warunku logicznego możemy, zamiast bloku if else, użyć specjalnego operatora trójargumentowego:

```
zmienna = warunek ? wartosc_jak_prawda :
          wartosc_jak_falsz;
```

Jest to szybsza i czytelniejsza forma od:

```
if (warunek) {
  zmienna = wartosc_jak_prawda;
} else {
  zmienna = wartosc_jak_falsz;
}
```

Blok switch

Jeżeli chcemy, aby jakiś kod został wykonany w momencie, gdy zmienna znajduje się w określonym stanie, to możemy użyć bloku switch:

```
switch (key) {
case value1:
  // instrukcje dla key równego value1
  break;
case value2:
  // instrukcje dla key równego value2
  break;
default:
  break;
}
```

W języku Java klucz (key) może być tylko typu **int** lub **char** (ten jest odczytywany jako liczba z tablicy unicode), a od wersji 1.5 też **enum**. Warto zauważyć, iż słowo kluczowe **break** jest w pewnym sensie obowiązkowe. Jeżeli nie użyjemy go, to instrukcje będą dalej przetwarzane. Zatem rezultatem takiego kodu:

```
int i = 0;
switch (i) {
case 0:
  System.out.println(0);
case 1:
  System.out.println(1);
  break;
default:
  System.out.println("default");
  break;
}
```

będzie:

0
1

Pętle

Jeżeli chcemy wykonać jakiś fragment kodu wielokrotnie, to możemy wypisać go explicite:

```
System.out.println(1);
System.out.println(2);
System.out.println(3);
```

Takie rozwiązanie jest jednak złe. Co jeżeli chcemy wypisać np. wszystkie posty z forum? Jest tego trochę. W dodatku liczba ta wciąż rośnie więc w momencie uruchomienia kodu na pewno nie będzie tam ostatnich postów. Rozwiązaniem tego problemu jest specjalna instrukcja języka - Pętla. Ogólna zasada działania pętli jest bardzo prosta i można ją ująć na trzy sposoby:

WYKONAJ *POLECENIE* N-KROTNIE
lub
WYKONAJ *POLECENIE* DOPÓKI SPEŁNIONY JEST *WARUNEK*
lub
WYKONAJ *POLECENIE* DLA KAŻDEGO ELEMENTU ZBIORU *B*

Tak oto zdefiniowaliśmy trzy podstawowe rodzaje pętli w Javie, a ogólniej w programowaniu.

Pętla for

Jest to pętla policzalna, czyli taka, o której możemy powiedzieć, iż wykona się określoną liczbę razy. Ogólna składnia pętli for wygląda w następujący sposób:

```
for (int i = 0; warunek; krok) {
  // instrukcja
}
```

Uwagi:

- Zmienną i nazywamy *Indeksem Pętli*
- Indeks może być dowolnym typem prostym poza **boolean**. Typ **char** ograniczony jest do 65535
- Warunek może być dowolnym zdaniem logicznym, należy jednak zwrócić uwagę by nie była to tautologia. Otrzymamy wtedy pętle nieskończoną
- Krok pętli może być dowolny jednak tak samo jak w przypadku warunku, trzeba uważać na zapętlenie się programu.

Gdzie należy używać pętli for? Odpowiedź na to pytanie jest jedną z kwestii spornych i wywołuje gorące dyskusje. Pętla ta najlepiej sprawdza się gdy chcemy wykonać jakąś operację na wszystkich elementach tablicy. Jest naturalną i najbardziej czytelną dla tego typu problemów:

```
int[] a = new int[] { 1, 2, 3 };
for (int i = 0; i < a.length; i++) {
  System.out.println(a[i]);
}
```

Odmianą pętli for wprowadzoną w wersji 1.5 jest wersja przyjmująca dwa argumenty. Iterator i warunek. Operuje on na kolekcjach. Przykład:

```
Collection<Object> col = new HashSet<Object>();
col.add("a");
col.add("b");
for (Iterator<Object> it = col.iterator(); it
    .hasNext();) {
  System.out.println(it.next());
}
```

Pętle while i do while

Pętle while i do while są pętlami niepoliczalnymi, czyli takimi, o których nie możemy powiedzieć ile razy się wykonają. Ich składnia jest następująca:

```
while (warunekLogiczny) {
  // instrukcja
}
/*---------*/
do {
  // instrukcja
} while (warunekLogiczny);
```

Uwagi:

- warunek musi być zmienną **boolean** lub obiektem klasy **java.lang.Boolean**.

Obie te konstrukcje są bardzo podobne do siebie. Główna różnica polega na tym, iż w pętli while warunek jest sprawdzany przed wykonaniem instrukcji, a w pętli do while po wykonaniu instrukcji. Oznacza to, że pętla do while wykona się co najmniej jeden raz. Poniższy przykład ilustruje różnicę:

```
int i = 0;
while (i < 1) {
  System.out.println("while " + i);
  i++;
}
do {
  System.out.println("do while " + i);
  i++;
} while (i < 1);
```

Druga pętla wykona się pomimo iż warunek, 1 < 1, nie jest prawdziwy.

Kiedy używać? Najczęściej pętla while jest wykorzystywana do nasłuchiwania. Polega to na stworzeniu nieskończonej pętli, najczęściej podając jako argument słowo **true**, której zadaniem jest wykonywanie danego kodu nasłuchującego. Prosty szablon animacji:

```
while (animuj) {
  // kod animacji
}
```

Dopóki flaga animuj jest prawdziwa, wykonywana jest animacja. Podobnie ma się sprawa z prostymi serwerami które nasłuchują w ten sposób na portach.

Pętla for element : Iterable

W raz nadejściem Javy w wersji 1.5, pojawiła się możliwość użycia konstrukcji **for E : I**. Ta zdawać by się mogło dziwaczna konstrukcja jest w rzeczywistości odpowiednikiem pętli foreach. Jako argument przyjmuje tablicę lub obiekt klasy implementującej interfejs Iterable. Przykład:

```
List<Object> l = new LinkedList<Object>();
l.add("a");
l.add("b");
for (Object e : l) {
  System.out.println(e.toString());
}
```

Gdzie stosować? Konstrukcja ta jest najodpowiedniejsza dla wszelkiego rodzaju list, kolekcji i wektorów.

Przerywanie i przechodzenie do następnego kroku w pętlach

Czasami zdarza się, że chcemy przerwać lub pominąć krok pętli, jeżeli spełniony jest jakiś warunek. Aby uzyskać taki efekt, musimy użyć jednego z dwóch słów kluczowych.

break

Jeżeli chcemy przerwać wykonanie pętli gdy spełniony jest jakiś warunek, to musimy użyć słowa **break**:

```
int i = 0;
while (true) {
  if (i == 5)
    break;
  System.out.println(i);
  i++;
}
```

continue

Jeżeli chcemy pominąć jakiś krok w pętli, to musimy użyć słowa **continue**:

```
for (int i = 0; i < 10; i++) {
  if (i == 5)
    continue;
  System.out.println(i);
}
```

Podsumowanie pętli

- Mamy trzy rodzaje pętli,
- Różnią się one zasadą działania,
- Należy uważać, by nie popełnić błędu i nie stworzyć pętli nieskończonej.

Tablice i Kontenery

Omówiliśmy już wszystkie najistotniejsze elementy języka Java. Jedną z ostatnich kwestii, jakie zostaną poruszone w tym artykule, jest temat tablic i kolekcji.

Wyobraźmy sobie, że mamy do stworzenia kilka Obiektów tej samej Klasy. Możemy zrobić to na kilka sposobów. Najprostszym jest zdefiniowanie ich jako kolejnych zmiennych:

```
Object o1 = new Object();
Object o2 = new Object();
Object o3 = new Object();
```

Metoda ta posiada wiele wad:
- tworzymy dużo dodatkowego kodu,
- tworzymy wiele zmiennych,
- chcąc wykonać operację na każdym z obiektów musimy duplikować kod.

Chcąc uniknąć tych nieprzyjemności, możemy wykorzystać jeden z mechanizmów do obsługi zbiorów obiektów.

Tablice

Kontener na dane, w którym do każdej z komórek można odwołać się za pomocą klucza. Klucz jest co do zasady wartością numeryczną.

Najprostszym opisem tablicy jest porównanie jej do tabeli, w której każda komórka ma swój unikatowy numer. Za pomocą numeru możemy odwołać się do komórki i pobrać lub umieścić w niej pewną daną. Java pozwala na definiowanie tablic na kilka sposobów:

```
// n - wielkość tablicy
Object[] o1;
o1 = new Object[n];
Object o2[] = { new Object(), new Object() };
int[] i = new int[10];
```

Komórki w tablicach są numerowane od **0**. Oznacza to, że ostatni element znajduje się w komórce o numerze o jeden mniejszym od długości tablicy. Tablica może mieć maksymalnie 2^31 -1 komórek. Uwaga! Inicjacja tablicy o maksymalnej długości może spowodować błąd przepełnienia stosu.

Jeżeli chcemy otrzymać wartość elementu na pozycji m, to wystarczy odwołać się do niego w ten sposób:

```
int elementM = i[m];
```

Tablica, nawet jeżeli jest tablicą typów prostych, jest też Obiektem. Warto o tym pamiętać, ponieważ częstym błędem jest porównywanie tablic w taki oto sposób:

```
Object[] o1;
o1 = new Object[2];
o1[0] = new Object();
o1[1] = new Object();
Object[] o2 = { new Object(), new Object() };
System.out.println("== " + (o1 == o2));
System.out.println("equals " + o1.equals(o2));
/***
 * == false equals false
 **/
```

Jedyną prawidłową metodą jest porównanie każdego elementu tablicy z elementem o takim samym indeksie w drugiej tablicy:

```
public boolean porownaj(Object[] o1, Object[] o2) {
  if (o1 == null || o2 == null)
    return false;
  if (o1.length != o2.length)
    return false;
  for (int i = o1.length - 1; i >= 0; i--) {
    if (!o1[i].equals(o2[i]))
      return false;
  }
  return true;
}
```

Java umożliwia też tworzenie tablic wielowymiarowych:

```
Object[][] o3 = {
    { new Object(), new Object() },
    { new Object(), new Object() } };
Object[] o4[];
o4 = new Object[10][10];
Object[][] o5 = new Object[2][];
o5[0] = new Object[10];
o5[1] = new Object[1];
```

Jak widać każda tablica w tablicy jest niezależna. Zmienna o5 to tablica dwuwymiarowa w której pierwszy wymiar ma krotność 2, a poszczególne komórki tego wymiaru odpowiednio 10 i 1.

Ostatnią istotną kwestią dotyczącą tablic jest zagadnienie inicjacji wartości komórek. Tablica jest inicjowana na tych samych zasadach co obiekt, to znaczy:

- jeżeli tablica jest polem Klasy, to jest inicjowana na **null**. Nie posiada rozmiaru.

- jeżeli tablica jest zmienną lokalną, to nie jest inicjowana i trzeba ją inicjować ręcznie.

Co jednak z poszczególnymi komórkami? Jeżeli tablica zostanie zainicjowana, to wszystkie komórki zostaną zainicjowane tak, jak by były polami obiektu i przyjmą wartości domyślne. Przykład:

```
Object o6[] = new Object[2];
System.out.println(o6[0]);
int[] j = new int[2];
System.out.println(j[0]);
```

Kontenery

Jednym z ograniczeń tablic jest ich ograniczona wielkość. Ograniczona zarówno w sensie ilości elementów, ale też zmiany rozmiarów tablicy. Chcąc zaradzić temu problemowi Java posiada dość pokaźny zbiór klas i interfejsów rożnych typów kontenerów.

Kontener (ang. collection) jest to struktura pozwalająca na przechowywanie danych w sposób uporządkowany. Posiada mechanizmy dodawania, usuwania i zamiany elementów.

Pakiet java.util zawiera w sobie wszystkie najważniejsze rodzaje kontenerów, a są to:

Listy

Lista to uporządkowany zbiór danych. Do elementu listy można dostać się za pomocą podania wartości indeksu. Interfejs java.util.List pozwala na przechowywanie danych w postaci list wiązanych, list opartych o tablice, stosu, wektora.

Mapy

Mapy pozwalają na dostęp do obiektów na podstawie wartości klucza. Każdy element mapy składa się z pary . Interfejs java.util.Map pozwala na stworzenie map opartych o hasz jak też o drzewa.

Set

Jest to specyficzny rodzaj kontenera, w którym obiekty przechowywane nie mogą się powtarzać. Interfejs java.util.Set pozwala na przechowywanie danych w postaci listy wiązanej bez powtórzeń, drzewa, kolekcji opartej o funkcję skrótu (hasz).

Kolejka

Kolejka jest to rodzaj kontenera pozwalający na dostęp do danych w oparciu o algorytmy FIFO i LIFO. Interfejs java.util.Queue pozwala na stworzenie kolejek blokujących, synchronizowanych i innych.

Podsumowanie

Jak można zauważyć, tablice i kontenery mają swoje wady i zalety. Wybierając kontener należy kierować się kilkoma prostymi pytaniami:

- czy wielkość zbioru jest stała?
- czy ilość dostępów do elementów w środku zbioru jest duża?
- czy ilość wstawień w środku zbioru jest duża?
- czy elementy mogą się powtarzać?

Odpowiedzi na te pytania pozwolą na określenie, jakiego typu kontenera należy użyć.

Tak oto szczęśliwie dobrnęliśmy do końca teorii związanej z Javą. W kolejnej części zajmiemy się już praktyką. Napiszemy pierwszy program i poznamy podstawowe narzędzia, które pozwolą nam kontynuować przygodę z programowaniem bez niepotrzebnego stresu.

ABG

MEMBER OF asseco GROUP

Producent oprogramowania dostawca rozwiązań informatycznych

DRQ softwarehouse → ABG

W związku ze zmianami organizacyjnymi w **Grupie Kapitałowej Asseco Poland**, pomiędzy **ABG S.A.** i **DRQ S.A.** zawarta została umowa objęcia zorganizowanej części przedsiębiorstwa **ABG S.A.** przez **DRQ S.A.**

Według harmonogramu połączenia, kolejnym krokiem jest planowana zmiana nazwy **DRQ S.A.** na **ABG S.A.** W strukturze Spółki znajdzie się **Pion Telekomunikacja** skupiający kompetencje **ABG S.A., SPIN S.A., DRQ S.A.** oraz **Asseco Poland.**

Telekomunikacja

ABG S.A., Biuro Kraków
ul. Podwale 3, 31-118 Kraków, tel. 012 42 30 969, fax 012 42 31 524

www.abg.com.pl

Google Data API

Bartosz Chrabski

Google postanowił zawojować rynek za pomocą aplikacji codziennego użytku. Kolejne propozycje programów użytkowych Google'a przerastają najśmielsze oczekiwania użytkowników.

Sam Google swojego czasu postawił na integracje swoich usług, już dziś istnieje możliwość otwierania za pomocą Google Mail dokumentów (Google Documents), oraz zarządzania listą kontaktów z Google Contacts. Jednak to, co najbardziej zdziwiło użytkowników usług firmy z Santa Clara to to, że został upubliczniony protokół GData który pozwala na integracje naszych aplikacji "Googlowych" z innymi aplikacjami przy zastosowaniu protokołu HTTP. Idea samej komunikacji opiera się na dwóch rozwiązaniach: RSS oraz ATOM, które w spójny sposób dostarczają nam dostępu do danych. Logiczne jest więc, że z samego GData może korzystać każdy język programowania wspierający HTTP, w tym między innymi Java dla której zostały udostępnione biblioteki Java Client Library for Google Data API. Zbiór klas dedykowany Javie ma także swoje odpowiedniki dla języków, takich jak .Net, Php, Python, Objective-C czy JavaScript, w których jest jednakowo dobrze wspierany. Sama biblioteka do Javy jest dostarczana w postaci modularnej która bazuje na 2 podstawowych elementach : GData Core oraz GData Client, a także paczkach przypisanych do każdej usługi (pełne źródła możemy pobrać z http://code.google.com/p/gdata-java-client/downloads/list).

Na chwile obecną integracja została zapewniona z aplikacjami takimi jak :

- Google Apps
- Google Base
- Blogger
- Google Calendar
- Code Search
- Google Contacts
- Google Documents
- Google Notebook
- Google SpreadSheets
- Picasa Web Albums
- YouTube
- Webmaster Tools
- Google Health
- Google Finance Portfolio
- Google Book Search

Piękno zapytań

Jak wcześniej wspomniałem działanie samego protokołu GData Api jest oparte o komunikacje HTTP, co wymusza odwoływanie się do zasobów przez odpowiednie adresy URL.

Dobrym przykładem na początek jest zapytanie skierowane do serwisu Picasa:

http://picasaweb.google.com/data/feed/
projection/path?kind=kind&access=visibility

w którym możemy wyróżnić:

- cześć stałą: http://picasaweb.google.com/data/feed/
- parametry które są ustalane zgodnie z zasadą /nazwa_paramteru/wartosc,

W przykładowym zapytaniu możemy wyróżnić takie atrybuty jak:

- projection (base lub api) - format otrzymywanej odpowiedzi.
  - Base - zgodny ze standardem Atom bez informacji specyficznych dla serwisu.
  - API - odpowiedz w formacie GData zawierająca dodatkowe informacje
- path - zdefiniowane źródło pobierania danych
- kind - rodzaj zwracanej informacji (photo/album/comment)
- access - rodzaj dostępności zasobu (public,private,all)

Uzupełniając poprawnie parametry otrzymujemy przykładowo:

http://picasaweb.google.com/data/feed/
base/user/lodzjugexample?kind=album&
access=public

Jako ciekawostkę możemy stworzyć zapytanie do przeszukiwania wszystkich zasobów Picasy:

http://picasaweb.google.com/data/feed/base/all?q=Java%20User&max-results=100

Niestety w obecnej chwili nie ma jednakowego standardu dla wszystkich aplikacji, jednak same biblioteki pozwalają na dynamiczne tworzenie zapytań.

Zapytania do przykładowych aplikacji Google:

Google Calendar:
www.google.com/calendar/feeds/userId/visibility/full

Blogger:
www.blogger.com/feeds/userId/blogs

Code Search:
www.google.com/codesearch/feeds/search?q=query

Contacts:
www.google.com/m8/feeds/contacts/userId/projection

Google Documents:
www.docs.google.com/feeds/documents/visibility/projection

Google Notebook:
www.google.com/notebook/feeds/userId/

YouTube:
www.gdata.youtube.com/feeds/projection/video/field

Picasa Web Albums:
www.picasaweb.google.com/data/feed/projection/path

Java Client Library for GData

Wracając z szarej krainy HTTP do naszej ulubionej Javy, powoli okazuje się, że sam protokół nie jest taki straszny bo Google, zapewnia nam bogate API do zarządzania usługami. Opakowywanie rozwiązania dla klas Javy sprawiło, że tworzenie zapytań stało się banalnie proste, przez zastosowanie klasy Query, a zmian dokonujemy już na obiektach.

Oto przykład potwierdzający powyższą tezę:

```
URL feedUrl = new URL(
    "http://picasaweb.google.com/data/feed/api/"
        + "user/username?kind=album");
PicasawebService myService = new PicasawebService(
    "Example1");
UserFeed myUserFeed = myService.getFeed(feedUrl,
    UserFeed.class);
for (AlbumEntry myAlbum : myUserFeed
    .getAlbumEntries()) {
  System.out.println(myAlbum.getTitle()
      .getPlainText());
}
```

Aplikacja sprowadza się do pętli, która pobiera nazwy albumów i je wyświetla. Cała obsługa usługi odbywa się przez klasę PicasawebService, która jest rozszerzoną wersją GoogleService.

Pytania jakie odnośnie powyższego przykładu mogą paść to:

1. Dlaczego podajemy Example1 w konstruktorze klasy PicasawebService?

2. Dlaczego odwołujemy się do użytkownika, jeżeli chcemy pobrać albumy?

Odpowiedz na pierwsze pytanie może wydać się z początku dziwna, ale Google chce aby podawać nazwę aplikacji która używa protokołu w celach identyfikacyjnych, czyli zapewne są to dane indeksowane. Druga kwestia wynika z budowy samej biblioteki, gdyż obiekt do którego się odwołujemy zawiera informacje o sobie samym i elementach bezpośrednio mu podrzędnych czyli np. UserFeed zawiera informacje o użytkowniku i albumach natomiast AlbumFeed o albumie, oraz należących do niego zdjęciach.

Wchodząc coraz głębiej w przykłady, dochodzimy do modyfikacji danych w naszych aplikacjach i wtedy dopiero możemy zobaczyć prawdziwą funkcjonalność samego protokołu oraz biblioteki, co obrazuje poniższy listing.

```
URL feedUrl = new URL(
    "http://picasaweb.google.com/data/feed/api/"
        + "user/username?kind=album");
PicasawebService myService = new PicasawebService(
    "Example2");
AlbumEntry myAlbum = new AlbumEntry();
myAlbum.setTitle(new PlainTextConstruct(
    "Java Express"));
myAlbum.setDescription(new PlainTextConstruct(
    "Photos to new edition"));
AlbumEntry insertedEntry = myService.insert(
    postUrl, myAlbum);
```

```
// uaktualnianie albumów
myAlbum.setDescription(new PlainTextConstruct(
    "Updated album description"));
myAlbum.update();

// kasowanie albumów
myAlbum.delete();
```

Analizując powyższy listing możemy zauważyć że nigdzie nie ma linijki odpowiadającej za samą autoryzacje do aplikacji, a wynika to z tego że jeżeli jesteśmy zalogowani w przeglądarce do aplikacji a GData Api bazuje na HTTP, to odpowiedź na to pytanie staje się prosta. Nie mniej jednak dostarczone klasy same w sobie dostarczają mechanizm autoryzacji przy użyciu jawnej deklaracji loginu i hasła lub krążącego tokenu (zastosowanie w aplikacjach JEE).

```
CalendarService myService = new CalendarService(
    "exampleCo-exampleApp-1");
myService.setUserCredentials(
    "lodzjugexample@gmail.com", "mypassword");
URL feedUrl = new URL(
    "http://www.google.com/calendar/feeds/default/"
        + "allcalendars/full");

// owncalendars albo allcalendars
CalendarFeed resultFeed = myService.getFeed(
    feedUrl, CalendarFeed.class);
System.out.println("Your calendars:");
System.out.println();
for (int i = 0; i < resultFeed.getEntries().size();
    i++) {
  CalendarEntry entry = resultFeed.getEntries()
      .get(i);
  System.out.println("\t"
      + entry.getTitle().getPlainText());
}
```

W powyższym listingu możemy zauważyć, że autoryzacja odbywa się już poprzez funkcje setUserCredentials, a nie przez sesję zachowaną w samej przeglądarce. Kwestie samych praw dostępu i metody jaką należy wybrać są szeroko poruszane w dokumentacji (http://code.google.com/apis/gdata/articles/java_client_lib.html). Jeżeli chodzi o większą ilość przykładów to można je będzie znaleźć na stronie dWorld, a dodatkowo dla każdej aplikacji (http://code.google.com/apis/gdata/articles/java_client_lib.html#samples)

Dalsze kroki ?

Myślę że z powodu ograniczonej długości artykułu oraz przykładów które mogę zamieścić, poniżej nie pozostaje mi nic innego niż zamieszczenie linków do strony które mogą pomóc w zgłębianiu GData API i jego implementacji w Javie.

Wtyczka do eclipse wspierająca Google Data API

http://googledataapis.blogspot.com/2008/07/google-data-apis-java-client-eclipse.html

Ustawienia Eclipse IDE

http://code.google.com/apis/gdata/articles/eclipse.html

Główna strona samej biblioteki :

http://code.google.com/apis/gdata/articles/java_client_lib.html

Official Google Data APIs Blog

http://googledataapis.blogspot.com/

Atom and Google Data API

http://www.snellspace.com/wp/?p=306

Google Data Protocol

http://groups.google.com/group/google-help-dataapi

Introduction to the Google Base Data API

http://pl.youtube.com/watch?v=3WSUIe5id3E&feature=related

O autorze

Autor jest studentem V roku informatyki na wydziale FTIMS Politechniki Łódzkiej.

Poziom trudności

1 – początkujący (wstęp, ogólne informacje)

Nie tylko refaktoring, cz. II

Mariusz Sieraczkiewicz

W poprzednim numerze JAVA exPress zrobiliśmy pierwszy krok w kierunku uporządkowania pierwotnego kodu.

Zróbmy zatem kolejny. Przyjrzyjmy się bliżej następującemu fragmentowi:

```
public class ExtendedBitSet extends BitSet {
  int length;
```

Pole klasy length ma widoczność pakietową. Prawdopodobnie nie to było zamierzeniem autora. Być może zapomniał w sposób jawny napisać odpowiedni modyfikator dostępu. W każdym razie warto znać jedną z podstawowych konsekwencji tego rozwiązania: pole to będzie dostępne dla wszystkich klas w pakiecie, co ogranicza jego enkapsulacje informacji. Przypadek ten można uogólnić do stwierdzenia:

Nadawaj najbardziej restrykcyjny z możliwych modyfikatorów dostępu

W tym przypadku byłby to oczywiście modyfikator prywatny:

```
private int length;
```

W przypadku gdy tworzona klasa będzie klasą bazową innych klas oraz będzie potrzeba udostępnienia tego pola, należy użyć modyfikatora dostępu protected:

```
protected int length;
```

Dobrą praktyką jest jednak wybieranie modyfikatora private, aż do momentu, gdy nie zaistnieje potrzeba użycia tego pola w klasie podrzędnej (czyli do momentu wyprowadzania nowych klas). Jest to analogia do typowej dla administratorów sieciowych strategii bezpieczeństwa: domyślnie blokuj.

Poza powyższymi uwagami, chciałbym zaproponować jeszcze jedno udoskonalenie. Jestem zwolennikiem jawnego inicjalizowania zmiennych, gdyż jest to bezpośrednie ujawnienie intencji autora. Jeśli tworzę pole obiektowe, to jawnie przypisuję mu wartość początkową na null, kiedy tworzę liczbę całkowitą inicjuję ją zerem. W ten sposób oszczędzam potencjalnemu czytelnikowi mojego kodu domyślania się, czy rzeczywiście chodziło mi o wartość domyślną, czy być może nie dokońca znając szczegóły języka, przyjąłem błędne założenie. W przypadku wyszukiwania przyczyn błędów może to mieć duże znaczenie.

Zatem podsumowując:

Jawnie inicjuj pola i zmienne

W efekcie uzyskamy takie rozwiązanie:

```
private int length = 0;
```

W powyżej przytoczonym wierszu ujawnił się jeszcze jeden nawyk związany ze sposobem programowania

Jak najwięcej przestrzeni dla Twoich oczu

Wzrok lubi przestrzeń. Nie lubi zbitych zdań, ogromnych ilości wyrażeń na niewielkiej przestrzeni. Zróbcie prosty eksperyment. Weźcie niewielki fragment swojego kodu i dodajcie kilka spacji (pomiędzy operatorami, pomiędzy wierszami) i porównajcie, który zapis jest czytelniejszy.

Oto przykład:

```
public byte[] toByteArray() {
  int bytesNumber;
  if (length % 8 == 0)
    bytesNumber = length / 8;
  else
    bytesNumber = length / 8 + 1;
  byte[] arr = new byte[bytesNumber];
  for (int j = bytesNumber - 1, k = 0;
  j >= 0; j--, k++) {
    for (int i = j * 8; i < (j + 1) * 8; i++) {
      if (i == length)
        break;
      if (get(i))
        arr[k] += (byte) Math.pow(2, i % 8);
    }
  }
  return arr;
}
```

oraz wersja przestrzenna:

```
public byte[] toByteArray() {
  int bytesNumber = 0;
  if (length % 8 == 0) {
    bytesNumber = length / 8;
  } else {
    bytesNumber = length / 8 + 1;
  }
  byte[] arr = new byte[bytesNumber];
  for (int j = bytesNumber - 1, k = 0;
  j >= 0; j--, k++) {
    for (int i = j * 8; i < (j + 1) * 8; i++) {
      if (i == length) {
        break;
      }
      if (get(i)) {
        arr[k] += (byte) Math.pow(2, i % 8);
      }
    }
  }
  return arr;
}
```

Modyfikacja ta zajęła mi około dwóch minut. Jednak umiejętność tworzenia kodu zgodnego ze stylem kodowania, która jest moim nawykiem, umożliwia mi pisanie takiego kodu bez żadnego dodatkowego nakładu czasu. Za to jaka przyjemność z czytania! A to dopiero początek. Kiedy mówimy o stylu kodowania przychodzi mi do głowy jeszcze jedna reguła:

Używaj konsekwentnie przyjętego standardu kodowania

Obecnie nie wyobrażam sobie tworzenia kodu, który nie podlega z góry ustalonym zasadom. Jeśli chodzi o pracę w zespole, jest to wręcz warunek konieczny pracy grupowej. A mamy ogromne wsparcie, gdyż istnieje wiele gotowych do wykorzystania standardów, np. Code Conventions for the Java Programming Language (http://java.sun.com/docs/codeconv/), oraz narzędzi, które pomogą go, szczególnie w początkowym okresie, sumiennie przestrzegać (Checkstyle http://checkstyle.sourceforge.net/).

Poniżej znajduje się przykład omawianej klasy sformatowany wg standardu opartego na standardzie zaproponowanym przez Suna.

```
public class ExtendedBitSet extends BitSet {
  private int length = 0;

  public ExtendedBitSet(int size, String str) {
    super(size);
    length = size;
    int strLength = str.length();

    for (int i = 0; i < strLength; ++i) {
      if (str.charAt(strLength - 1 - i) == '1') {
        set(i);
      }
    }
  }

  public ExtendedBitSet(String str) {
    super(str.length());
    int strLength = str.length();
    length = strLength;
    for (int i = 0; i < strLength; ++i) {
      if (str.charAt(strLength - 1 - i) == '1') {
        set(i);
      }
    }
  }

  public void merge(ExtendedBitSet extendedBitSet)
{
    for (int i = extendedBitSet.nextSetBit(0);
    i >= 0; i = extendedBitSet.nextSetBit(i + 1)) {
      this.set(this.length + i);
    }
    this.length = this.length
        + extendedBitSet.length;
  }

  public int boolMultiply(
      ExtendedBitSet extendedBitSet) {
    int sum = 0;
    int len = 0;
    if (this.length < extendedBitSet.length) {
      len = this.length;
    } else {
      len = extendedBitSet.length;
    }
    for (int i = 0; i < len; i++) {
      if (this.get(i) && extendedBitSet.get(i)) {
        sum++;
      }
    }
    return sum % 2;
  }

  public byte[] toByteArray() {
    int bytesNumber = 0;
    if (length % 8 == 0) {
      bytesNumber = length / 8;
    } else {
      bytesNumber = length / 8 + 1;
    }

    byte[] arr = new byte[bytesNumber];
    for (int j = bytesNumber - 1, k = 0;
    j >= 0; j--, k++) {
      for (int i = j * 8; i < (j + 1) * 8; i++) {
        if (i == length) {
          break;
```

```
        }
        if (get(i)) {
          arr[k] += (byte) Math.pow(2, i % 8);
        }
      }
    }
    return arr;
  }

  public String convertToBitString(int size) {
    char[] resultArray = new char[size];
    for (int i = 0; i < size; ++i) {
      resultArray[i] = '0';
    }
    for (int i = this.nextSetBit(0); i >= 0;
    i = this.nextSetBit(i + 1)) {
      resultArray[size - 1 - i] = '1';
    }
    return new String(resultArray);
  }

  public String convertToBitString() {
    return convertToBitString(this.length);
  }
}
```

Wróćmy do naszego pola length. Tak naprawdę posiada ono jeszcze jeden mankament – jego nazwa jest dokładnie taka sama jak nazwa metody z klasy bazowej. Jest to sytuacja niekorzystna z dwóch powodów:

- dwa różne byty nie powinny mieć tej samej nazwy (metoda i pole), gdyż może to prowadzić do pomyłek,

- nazwa zmiennej nie odzwierciedla sensu właściwości. Pole to przechowuje wartość, która określa ustaloną długość wektora bitowego. Dużo większy sens miałaby na przykład nazwa fixedLength.

Zmieńmy zatem nazwe pola length na fixedLength. Podsumowując powyższe rozważania:

Nie używaj jednej nazwy do różnych celów

oraz

Nadawaj polom, metodom i klasom nazwy, które jednoznacznie odzwierciedlają ich znaczenie

Analizując dalej przykład, spójrzmy na oba konstruktory, wyraźnie zauważymy pewną właściwość - jest tam mnóstwo powtarzającego się kodu. W ten sposób docieramy do zasady będącej esencją refaktoringu:

Eliminuj wszelkie powtórzenia

Powtórzenie to zło, które towarzyszy programistom na każdym kroku. Kuszące kopiuj-wklej, zazwyczaj ostatecznie prowadzi do kilkunastominutowych lub co gorsza wielogodzinnych poszukiwań błędów, wynikających z rozsynchronizowania się podobnych fragmentów kodu. Powtórzenia na dłuższą metę są nie do utrzymania, stąd ich eliminowanie jest podstawowym celem wszelkich refaktoringów. Przykładowy kod możemy zmienić do następującej postaci:

```
public ExtendedBitSet(int size, String str) {
  super(size);
  fixedLength = size;
  initializeBitSet(str);
}

public ExtendedBitSet(String str) {
  this(str.length(), str);
  initializeBitSet(str);
}

private void initializeBitSet(String str) {
  int strLength = str.length();
  for (int i = 0; i < strLength; ++i) {
    if (str.charAt(strLength - 1 - i) == '1') {
      set(i);
    }
  }
}
```

Kod nam się powoli porządkuje i wygląda coraz lepiej. Wprowadziliśmy zmiany związane z wyglądem (standard kodowania i przestrzeń), wyeliminowaliśmy kilka powtórzeń i niejednoznaczności. Oczywiście zawsze należy wyważyć stopień refaktorowania lub upiększania kodu, tak aby nie stać się ofiarą perfekcjonizmu. Warto wesprzeć się pomocą innych programistów, najlepiej takich, którzy sami posługują się pewnymi zasadami oraz posiadają duże doświadczenie, i poprosić o opinię. Z pewnością wiele można się będzie dowiedzieć na temat swojego programowania.

Przyjrzyjmy się teraz metodzie boolMultipy oraz zmiennej lokalnej o nazwie sum. Zmienna ta jest deklarowana na samym początku metody, używana jest dużo później. Jest to nawyk, który pozostał jeszcze po językach proceduralnych (takich jak wczesne C przy PL/SQL), gdzie wymaga się zadeklarowania wszystkich zmiennych używanych w metodzie na samym początku. Na szczęście większość współczesnych języków programowania (w szczególności języków obiektów) nie ma tego ograniczenia. Zamiast tego podejścia sugeruję realizację zasady leniwej deklaracji zmiennych, którą można wyrazić za pomocą zdania

Deklaruj zmienne najpóźniej jak to tylko możliwe

Warto to robić z bardzo prostego powodu – łatwiej będzie czytać kod, jeśli w zasięgu wzroku będziemy mieli operacje wykonywane na danej zmiennej. Przy bardziej złożonych metodach umieszczenie deklaracji na samym początku, może spowodować, że analizując dalszą część metody nie będziemy w stanie zorientować się czy zmienna była do tej pory przetwarzana czy nie i co wpłynęło na jej wartość.

Załóżmy, że przewinęliśmy ekran tak, że widzimy następujący kod:

```
for (int i = 0; i < len; i++) {
  ExtendedBitSet extendedBitSet;
  if (this.get(i) && extendedBitSet.get(i)) {
    sum++;
  }
}
return sum % 2;
```

Rodzi się we mnie natychmiast pytanie – a co to za suma? Czy działo się z nią coś wcześniej? Czy może jej wartość została pobrana z zewnątrz?

W przypadku jednej zmiennej jeszcze to nie jest duży problem, ale wyobraźmy sobie sytuację, kiedy takich zmiennych jest pięć. Śledzenie logiki takiej metody będzie bardzo trudne.

Bardzo prosta operacja przeniesienia deklaracji nieco niżej spowoduje, że kod będzie dużo bardziej czytelny:

```
public int boolMultiply(
    ExtendedBitSet extendedBitSet) {
  int len = 0;
  if (this.fixedLength <
    extendedBitSet.fixedLength) {
    len = this.fixedLength;
  } else {
    len = extendedBitSet.fixedLength;
  }

  int sum = 0;
  for (int i = 0; i < len; i++) {
    if (this.get(i) && extendedBitSet.get(i)) {
      sum++;
    }
  }
  return sum % 2;
}
```

Nieco bardziej złożona sytuacja jest w metodzie toByteArray. Jest ona niesamowicie trudna w analizie. Mi zajęło około 20 minut zrozumienie zasady jej działania. Teraz sobie wyobraźmy projekt złożony z tysiąca klas, z których każda zawiera tego typu metody. Zapanowanie nad takim kodem będzie graniczyło z cudem. Spójrzmy na kod tej metody:

```
public byte[] toByteArray() {
  int bytesNumber = 0;

  if (fixedLength % 8 == 0) {
    bytesNumber = fixedLength / 8;
  } else {
    bytesNumber = fixedLength / 8 + 1;
  }
  byte[] arr = new byte[bytesNumber];
  for (int j = bytesNumber - 1, k = 0; j >= 0;
  j--, k++) {
    for (int i = j * 8; i < (j + 1) * 8; i++) {
      if (i == fixedLength) {
        break;
      }
      if (get(i)) {
        arr[k] += (byte) Math.pow(2, i % 8);
      }
    }
  }
  return arr;
}
```

Metoda toByteArray zwraca bajtową reprezentacje ciągów bitowych – czyli ciąg bitowy o długości 14 bitów można zaprezentować za pomocą 2 bajtów, zaś ciąg bitowy o długości 23 bity można zaprezentować za pomocą 3 bajtów.

Algrotym można opisać w następujących krokach:

- określ liczbę bajtów,
- dla każdej ósemki bitów (lub kilku bitów w przypadku niepełnych bajtów) wykonaj następującą operację:
- sprawdź wartość każdego bitu
- jeśli bit ma wartość 1, dodaj odpowiednią potęgę dwójki (wynikającą z pozycji bitu w bajcie) do wyniku danego bajtu.

Dlaczegóżby nie wyrazić tego algorytmu w formie programistycznej? Często o tym się zapomina. Jako programiści próbujemy w zwięzły sposób wyrazić nasze pomysły, co zazwyczaj prowadzi do bardzo nieczytelnych rozwiązań, których nie tylko inni ale i my sami nie jesteśmy w stanie zrozumieć w krótkim czasie. Ideałem jest dążenie do tego, aby jedno spojrzenie wystarczyło do odszyfrowania intencji twórcy. Nie potrzebujemy zagłębiać się w szczegóły realizacji algortmu, ważne byśmy wychwycili jego główną myśl. Spójrzmy na inną implementację metody toByteArray:

```
public byte[] toByteArray() {
  int bytesCount = computeBytesCount();
  byte[] byteArray = new byte[bytesCount];
  for (int i = 0; i < bytesCount; ++i) {
    int byteNumber = bytesCount - i - 1;
    byteArray[byteNumber] = computeByteValue(i);
  }
  return byteArray;
}
```

Najważniejsza zmiana, polega na bezpośrednim wyrażeniu algorytmu na ogólnym poziomie. Dzięki czemu jesteśmy w stanie w krótkim czasie zrozumieć intencję twórcy. Pomocnicza metoda computeBytesCount znajduje ilość bajtów nowej reprezentacji. W pętli dla każdego bajtu wykonywana jest operacja obliczania wartości bajtu i zapamiętywania wyniku. Czyż taki zapis nie jest dużo prostszy? Co z tego, że nie widać wszystkich elementów realizacji algorytmu. Bardziej dociekliwi zawsze mogą zajrzeć do metod computeBytesCount i computeByteValue.

Mogą one wyglądać następująco:

```
private byte computeByteValue(int byteNumber) {
  int firstBitPosition = byteNumber * 8;
  int lastBitPosition = (byteNumber + 1) * 8 - 1;

  byte byteValue = 0;
  for (int i = this.nextSetBit(firstBitPosition);
  i >= firstBitPosition
      && i <= lastBitPosition; i = this
      .nextSetBit(i + 1)) {
    int currentBitPosition = i - firstBitPosition;
    if (get(i) == true) {
      byteValue += (byte) Math.pow(2,
          currentBitPosition % 8);
    }
  }
  return byteValue;
}

private int computeBytesCount() {
  int bytesCount = 0;
  if (fixedLength % 8 == 0) {
    bytesCount = fixedLength / 8;
  } else {
    bytesCount = fixedLength / 8 + 1;
  }
  return bytesCount;
}
```

Warto zauważyć, że kod realizujący zadanie zbytnio się nie uprościł, ale dużo łatwiej jest go teraz przeanalizować i zrozumieć. Teoria refaktoringu nazywa tego typu działania wyłuskiwaniem metody (ang. extract method). Prawdę mówiąc tutaj poszliśmy nieco dalej, gdyż zmodyfikowaliśmy nieco implementację algorytmu, tak aby stała się bardziej czytelna. Zwróćmy uwagę na wiersze:

```
int firstBitPosition = byteNumber * 8;
int lastBitPosition = (byteNumber + 1) * 8 - 1;
```

Tak naprawdę są one wyodrębnieniem bardzo enigmatycznych wyrażeń z wiersza:

```
for (int i = j * 8; i < (j + 1) * 8; i++) {
```

Czyż intencja w takim przypadku nie staje się oczywista? Moje wieloletnie doświadczenia doprowadziły mnie do wniosku:

JAWNIE NAZYWAJ ZŁOŻONE ELEMENTY KODU

zamiast

```
j * 8
```

napisz

```
int firstBitPosition = byteNumber * 8;
```

Zasadę te rozszerzam do instrukcji warunkowych. Często w kodzie możemy spotkać wyrażenia, za którymi kryje się pewna logika. Warto bezpośrednio wyrazić naszą intencję. Czytelność niesamowicie wzrasta.

Zamiast

```
if (index >= 0)
```

napisz

```
boolean isIndexInRange = (index >= 0);
if (isIndexInRange) {
```

Kod zaczyna się czytać jak książkę! Przecież programowanie jest dla ludzi. Ułatwiajmy zatem sobie życie.

A oto najważniejsza zasada, będąca kwintesencją powyższych rozważań:

PISZ KOD W TAKI SPOSÓB, ABY CZYTAŁO SIĘ GO JAK POWIEŚĆ. UŻYWAJ JEDNOZNACZNYCH I JEDNOCZEŚNIE PROSTYCH NAZW. REALIZOWANE OPERACJE DZIEL NA LOGICZNE CZĘŚCI I KAŻDĄ IMPLEMENTUJ W OSOBNYCH METODACH.

Myślę, że jak na jeden raz, wystarczy. Wystarczy, żeby zaostrzyć apetyty i wzbudzić ochotę na więcej. Żeby oprowadzić nieco po ogrodzie refaktoringu i jego przyległościach. Zapewniam, że to niesamowite miejsce i daje niesamowicie wiele radości i satysfakcji.

Poniżej zamieszczam ostateczną wersję kodu, który przechodzi załączone testy (oczywiście ze zmianą uwzględniającą niestatyczność metod merge i boolMultiply).

Końcowa postać przykładowej klasy (warto ją porównać z postacią początkową):

```
public class ExtendedBitSet extends BitSet {
  private int fixedLength = 0;

  public ExtendedBitSet(int size, String str) {
    super(size);
    fixedLength = size;
    initializeBitSet(str);
  }

  public ExtendedBitSet(String str) {
    this(str.length(), str);
    initializeBitSet(str);
  }

  private void initializeBitSet(String str) {
    int strLength = str.length();
    for (int i = 0; i < strLength; ++i) {
      if (str.charAt(strLength - 1 - i) == '1') {
        set(i);
      }
    }
  }

  public void merge(ExtendedBitSet extendedBitSet)
{
    for (int i = extendedBitSet.nextSetBit(0);
    i >= 0; i = extendedBitSet
        .nextSetBit(i + 1)) {
      this.set(this.fixedLength + i);
    }
    this.fixedLength = this.fixedLength
        + extendedBitSet.fixedLength;
  }

  public int boolMultiply(
      ExtendedBitSet extendedBitSet) {
    int len = 0;
    if (this.fixedLength <
```

```
      extendedBitSet.fixedLength) {
    len = this.fixedLength;
  } else {
    len = extendedBitSet.fixedLength;
  }
  int sum = 0;
  for (int i = 0; i < len; i++) {
    if (this.get(i) && extendedBitSet.get(i)) {
      sum++;
    }
  }
  return sum % 2;
}

public byte[] toByteArray() {
  int bytesCount = computeBytesCount();
  byte[] byteArray = new byte[bytesCount];

  for (int i = 0; i < bytesCount; ++i) {
    int byteNumber = bytesCount - i - 1;
    byteArray[byteNumber] = computeByteValue(i);
  }
  return byteArray;
}

private byte computeByteValue(int byteNumber) {
  int firstBitPosition = byteNumber * 8;
  int lastBitPosition = (byteNumber + 1) * 8 - 1;

  byte byteValue = 0;
  for (int i = this.nextSetBit(firstBitPosition);
  i >= firstBitPosition
      && i <= lastBitPosition; i = this
      .nextSetBit(i + 1)) {
    int currentBitPosition = i - firstBitPosition;
    if (get(i) == true) {
      byteValue += (byte) Math.pow(2,
          currentBitPosition % 8);
    }
  }
  return byteValue;
}

private int computeBytesCount() {
  int bytesCount = 0;
  if (fixedLength % 8 == 0) {
    bytesCount = fixedLength / 8;
  } else {
    bytesCount = fixedLength / 8 + 1;
  }
  return bytesCount;
}

public String convertToBitString(int size) {
  char[] resultArray = new char[size];
  for (int i = 0; i < size; ++i) {
    resultArray[i] = '0';
  }

  for (int i = this.nextSetBit(0); i >= 0;
  i = this.nextSetBit(i + 1)) {
    resultArray[size - 1 - i] = '1';
  }
  return new String(resultArray);
}

public String convertToBitString() {
  return convertToBitString(this.fixedLength);
}
```

Kariera w e-point SA

e-point SA to wiodący software house, lider biznesowych zastosowań technologii Java.

Działalność

Już od ponad 10 lat e-point SA specjalizuje się w tworzeniu dedykowanych rozwiązań internetowych na potrzeby biznesu. Korzysta przy tym z własnego środowiska programistycznego OneWeb, uznanych standardów informatycznych (Java 2 Enterprise Edition, Linux) oraz oprogramowania dostarczanego przez międzynarodowych partnerów (IBM, Intel, Oracle).

e-point SA tworzy:

- **Portale korporacyjne** – serwisy www, wewnętrzne portale informacyjne (intranet, extranet), systemy e-learning. Ich podstawą jest autorska platforma Active Content, która nie tylko pełni funkcję systemu zarządzania treścią (*content management system*), ale również umożliwia integrowanie aplikacji internetowych z serwisami www.
- **Systemy e-commerce** – w oparciu o własne środowisko programistyczne OneWeb oraz komercyjne produkty pakietu IBM WebSphere firma dostarcza rozwiązania e-commerce pozwalające na sprzedaż produktów i usług. Są to przede wszystkim sklepy internetowe oraz systemy obsługi klientów.
- **Aplikacje dedykowane** – e-point SA projektuje i buduje dedykowane aplikacje usprawniające wybrane procesy biznesowe w organizacji. Są to głównie systemy ewidencji i zarządzania zasobami, wnioskami, zamówieniami, szkoleniami, jak również systemy Customer Relationship Management czy wsparcia sprzedaży.

Ponadto firma zapewnia swoim klientom komplet usług niezbędnych do poprawnego funkcjonowania rozwiązań internetowych, a więc bieżącą obsługę i utrzymanie systemów oraz hosting.

e-point SA pracuje głównie dla międzynarodowych korporacji. Obecnie eksport usług stanowi ponad 50% przychodów firmy.

Profesjonalizm i innowacyjność przedsięwzięć realizowanych przez e-point SA były wielokrotnie nagradzane. W 2007 roku Międzynarodowe Stowarzyszenie Project Management (IPMA) przyznało firmie główną nagrodę w prestiżowym konkursie „Project Excellence Award Polska" za stworzenie platformy e-commerce dla europejskiego oddziału korporacji Amway.

Kariera

e-point SA jest firmą zorientowaną na projekty. Nie koncentruje się na tworzeniu sztywnych procedur i zasad. Zamiast tego stwarza warunki do współpracy dla ludzi o różnych umiejętnościach.

Firma poszukuje osób, które wniosą do niej wiedzę i doświadczenie, wykazując się przy tym kreatywnością, rzetelnością i zaangażowaniem. W zamian daje im możliwość udziału w ambitnych projektach i satysfakcję z efektów własnej pracy.

Swoim pracownikom e-point SA oferuje:

- kontakt z nowoczesnymi technologiami i najlepszymi praktykami inżynieryjnymi,
- pracę w młodym, kompetentnym i utalentowanym zespole,
- atrakcyjne wynagrodzenie,
- indywidualną ścieżkę rozwoju zawodowego, zgodnie z potrzebami, wiedzą i umiejętnościami,
- dobrą organizację, unikalną atmosferę i komfortowe warunki pracy,
- dogodną lokalizację biura na warszawskim Mokotowie, w pobliżu stacji Metra Wierzbno,
- opiekę medyczną (LIM Center),
- program rehabilitacji ruchowej w siedzibie firmy,
- wyjazdy i imprezy integracyjne.

Warto dodać, że w uznaniu dla prowadzonej polityki kadrowej firma e-point SA została wyróżniona godłem „Inwestor w Kapitał Ludzki".

Najczęściej oferowane stanowiska pracy w e-point SA:

- Programista aplikacji J2EE
- Web Developer
- Administrator Sieci i Systemów
- Kierownik Zespołu Programistów
- Producent
- Project Manager
- Account Manager
- Menedżer serwisowy
- Projektant Koncepcji / Architekt Informacji
- Architekt Systemów / Projektant Aplikacji

Szczegółowe zakresy obowiązków na poszczególnych stanowiskach, wymagania stawiane kandydatom, jak również aktualne oferty pracy znajdziecie na stronie internetowej:

- **www.e-point.pl/kariera**

OSGi Declarative Services

Jacek Pospychała

Wprowadzenie do OSGi

OSGi, coraz popularniejszy temat dyskusji wśród programistów Javy, wg. Burton group przeżywa obecnie okres największego rozkwitu. Należy jednak pamiętać że sam standard został zainicjowany już dawno, bo w 1999 roku. Powstałe wówczas OSGi Alliance zrzeszało na początku głównie firmy telekomunikacyjne. Misją OSGi jest utworzenie specyfikacji uniwersalnego mechanizmu modułów dla języka Java. Najnowsze wydanie specyfikacji, o numerze 4.1, posiada szereg implementacji zarówno komercyjnych (Knoplerfish Pro, Prosyst mBedded Server Professional), jak i darmowych (Apache Felix, Eclipse Equinox). Sama organizacja OSGi nie zajmuje się implementacją, a jedynie (lub aż!) opracowywaniem standardu – kompromisu, który skłonni są zaakceptować jej członkowie.

Mimo, że składnia języka Java przewiduje pojęcie pakietów (których zadaniem jest grupowanie klas powiązanych ze sobą logicznie), to we współczesnych, rozbudowanych systemach zapanowanie nad dziesiątkami, czy setkami pakietów jest nie lada wyzwaniem. W końcu pakiety klas nigdy nie działają w próżni, a cały czas komunikują się między sobą. Egzekwowanie, czy w ogóle śledzenie właściwych zależności między różnymi pakietami – tak by np. były one zgodne z przyjętą architekturą – jest nie jest łatwe i staje się coraz trudniejsze z kolejnymi wersjami produkowanego systemu.

OSGi upraszcza całe to piekiełko pakietów wprowadzając koncepcję modułu - zbioru klas Java (pogrupowanych w pakiety) i zasobów (np. pliki tekstowe, graficzne, itp.), opisanych plikiem manifestu. Moduły są dostarczane w postaci jednego pliku JAR i na pierwszy rzut oka nie wiele różnią się od typowych plików JAR znanych ze specyfikacji Javy. Kluczową różnicę stanowi jednak manifest. Opisuje on interfejsy komunikacyjne modułu, a dokładniej pakiety publikowane przez siebie oraz wymagane z innych modułów. Informacje te są przechowywane w tekście manifestu, a nie w kodzie Javy, by środowisko OSGi mogło spełnić wymagania każdego modułu, zanim maszyna wirtualna Javy zacznie ładować jego klasy. Ze względu na telekomunikacyjne korzenie standardu, jednym z kluczowych wymagań była zawsze możliwość nieprzerwanej pracy środowiska modułowego, a więc także w trakcie aktualizacji oprogramowania i wymiany modułów systemu. Z punktu widzenia programisty, oznacza to tyle że każdy moduł (klasy Java!) może w dowolnej chwili zniknąć, zostawiając na pastwę losu wszystkie pozostałe wymagające go moduły. To jest zupełnie nowe wymaganie, w porównaniu do dotychczasowych aplikacji pisanych w Javie.

Komunikacja w OSGi: Usługi

Najprostszy sposób komunikacji w OSGi to zwyczajna interakcja między klasami. Może być zaimplementowana dokładnie w taki sam sposób jak w przypadku komunikacji między różnymi plikami JAR, jednak nie mamy wówczas żadnej kontroli nad dynamiczną naturą modułów. W tej sytuacji próba zdefiniowania usług, które klasy z różnych modułów mogłyby świadczyć sobie nawzajem, szybko sprowadzi się jedynie do ograniczonego wzorca Singleton.

Z drugiej strony, moduły zainteresowane wybranymi usługami mogą uzyskać do nich dostęp poprzez rejestr usług (ang. service registry). W oferowanym przez OSGi rejestrze każdy moduł może ogłosić (zareklamować!) usługi, które świadczy. Usługa jest powiązana z interfejsem (lub wieloma interfejsami), który implementuje. W rejestrze może być wiele różnych implementacji tej samej usługi, co pozwala na bardziej abstrakcyjne projektowanie systemu (zamiast skupiania się na jednej implementacji). Korzystając z rejestru, można sprawdzić czy usługa jest cały czas dostępna, czy też nie zniknęła wraz z udostępniającym ją modułem.

Moduł rejestruje usługę poprzez interfejs `BundleContext`, do którego otrzymuje referencję zawsze przy starcie. Poniżej zarejestrowano przykładową usługę `org.example.MailOffice`. Implementacja usługi jest w klasie `MailOfficeImpl`. Z tą usługą nie są związane żadne dodat-

kowe własności – stąd trzeci argument to null.

```
ServiceRegistration reg = context
    .registerService("org.example.MailOffice",
        new MailOfficeImpl(), null);
```

Poniższy fragment kodu prezentuje w jaki sposób uzyskać dostęp do udostępnionej przed chwilą usługi:

```
ServiceReference ref = context
    .getServiceReference("org.example.MailOffice");
if (ref != null) {
  MailOffice office = (org.example.MailOffice)
    context.getService(ref);
  office.sendMessage("I'm using you, service!");
```

W pierwszej linii odpytujemy rejestr, czy zna usługę o typie `org.example.MailOffice`. Jeżeli jest taka zarejestrowana, to otrzymamy referencję do niej w postaci interfejsu `ServiceReference`. W kolejnym kroku, ponownie poprzez kontekst modułu, można uzyskać bezpośrednio klasę realizującą interesujący nas interfejs. W naszym przypadku jest to `org.example.MailExample`.

Usługa zostanie usunięta z rejestru w momencie zatrzymania modułu, lub poprzez bezpośrednie wywołanie `ServiceRegistration.unregister()`.

Korzystanie z usług w sposób przedstawiony powyżej ma niestety kilka wad. Moduł musi najpierw zostać uruchomiony, by uzyskać dostęp do kontekstu `BundleContext`. Podczas startu systemu z dużą ilością modułów może to znacznie wydłużyć czas uruchamiania. W przypadku dużej liczby usług, łatwo stracić nad nimi kontrolę, gdyż odwołania są zaszyfrowane bezpośrednio w kodzie programu. Jeżeli między usługami występują zależności, również samemu trzeba się zatroszczyć o ich wcześniejsze spełnienie. Sprawia to, że korzystanie z usług wiąże się z dużą ilością powtarzalnego kodu, który jest dodatkowo trudny w utrzymaniu ze względu na dynamizm całego środowiska. Główną zaletą jest pełna kontrola nad usługami.

Declarative Services

Zamiast programowo uruchamiać i zatrzymywać usługi oraz spełniać zależności między nimi, można przecież wykorzystać podobny fortel jak z samymi modułami! Zrzucimy tą czynność na środowisko, a całą konfigurację umieścimy w plikach tekstowych. W ten sposób usługa zostanie uruchomiona dopiero, gdy jej wszystkie wymagania zostaną spełnione, a zarządzanie zależnościami między usługami jest dużo prostsze. By aktywować usługi nie jest już konieczne uruchamianie wszystkich modułów – jedynie te usługi, które muszą być aktywne, zostaną aktywowane wraz ze swoimi modułami.

Aby móc korzystać z Declarative Services (w skrócie DS), należy się upewnić że w środowisku OSGi jest uruchomiony moduł z implementacją tej usługi. Przy tworzeniu tego artykułu wykorzystano moduł z Eclipse Equinox – `org.eclipse.equinox.ds`.

Wracając do przykładu z `MailService`, zamiast programowo rejestrować usługę poprzez odwołanie do kontekstu `BundleContext`, należy zmodyfikować plik manifestu – wskazać plik z opisem komponentu realizującego usługę:

```
Service-Component: mailOffice.xml
```

Plik `mailOffice.xml` zawiera definicję usługi. Usługa ta powinna być uruchamiana zaraz po starcie systemu i realizować interfejs `org.example.MailOffice`. Jak pamiętamy z wcześniejszego kodu, klasa implementująca tą usługę to `org.example.MailOfficeImpl`. Cała definicja wygląda następująco:

```
<?xml version="1.0" encoding="UTF-8"?>
<component immediate="true" name="mailOffice">
   <implementation class=
      "org.example.MailOfficeImpl" />
   <service>
      <provider interface="org.example.MailOffice"
/>
   </service>
</component>
```

Poprawnie zdefiniowana usługa jest dostępna poprzez rejestr i może zostać wykorzystana podobnie jak w poprzednim przykładzie – poprzez programowe odwołanie. Wówczas jednak powiązanie między modułami udostępniającym i wykorzystującym komponent `mailOffice` nadal nie będzie czytelne. Zdefiniujmy zatem drugi komponent, tym razem wykorzystujący `mailOffice`. Nadamy mu nazwę Person, a implementowany będzie przez klasę `home.Person`. Wymaga ona komponentu o nazwie `mailOffice` dostarczającego interfejs `org.example.MailOffice`, by móc wysyłać wiadomości do urzędu. By komponent Person został aktywowany potrzebna jest dokład-

nie jedna usługa `MailOffice`. Jeśli nasza osoba (Person) wykorzystywałaby wielu dostawców poczty (np. różne konta pocztowe), byłaby powiadamiana o wszystkich pojawiających się i znikających dostawcach.

Poniżej przykład definicji komponentu:

```
<?xml version="1.0" encoding="UTF-8"?>
<component immediate="true" name="person">
   <implementation class="home.Person" />
   <reference cardinality="1..1"
      interface="org.example.MailOffice"
      name="MailOffice" policy="static" />
</component>
```

Jak tylko w systemie pojawią się usługi `MailOffice`, zostanie utworzony nowy obiekt `Person`, a referencja do `MailOffice` zostanie przekazana poprzez metodę `Person.activate(ComponentContext ctx)`. Korzystając z kontekstu na bieżąco można odwoływać się do usługi i sprawdzać czy ona istnieje (strategia typu lookup).

Jest także drugi sposób odwoływania się do wymaganej usługi. DS umożliwiają automatyczne przekazywanie referencji do usługi za każdym razem, gdy nowa się pojawia w systemie, oraz gdy któraś z implementacji znika (strategia zdarzeniowa, z ang. event based). W tym podejściu trzeba uzupełnić znacznik `<reference>` w definicji komponentu o dwa nowe argumenty – bind oraz unbind:

```
<reference cardinality="1..1"
  interface="org.example.MailOffice"
  name="MailOffice" policy="static"
  bind="setMail" unbind="unsetMail" />
```

Z kolei klasę `Person` uzupełniamy o metody:

```
protected void setMail(MailOffice o);
protected void unsetMail(MailOffice o);
```

Wraz ze zmianą widoczności referencji `MailOffice`, metody te na bieżąco będą wywoływane przez środowisko.

Poza tym prostym przykładem biblioteka DS oferuje jeszcze dwa inne rodzaje komponentów: delayed component oraz component factory. Komponent typu „delayed” (oraz cały deklarujący go moduł) zostanie uruchomiony dopiero gdy inny komponent będzie go wymagał. Mechanizm ten przypomina dobrze znaną koncepcję lazy-loading. To rozwiązanie jest możliwe dlatego, że moduł DS rejestruje usługi na rzecz innych modułów bez ich uruchamiania – tworzy proxy, a prawdziwa implementacja uruchamiana jest dopiero przy pierwszym odwołaniu.

Z kolei fabryka komponentów ma zastosowanie wszędzie tam gdzie potrzebne jest wiele instancji danego komponentu. Chociażby komponent `mailOffice` – zdefiniowany z dodatkową opcją `factory=<nazwaFabryki>` spowoduje utworzenie usługi typu `org.osgi.service.component.ComponentFactory`, która pozwala tworzyć nowe instancje komponentu (kolejne usługi `mailOffice`) poprzez metodę `newInstance(Dictionary)`. W każdym wywołaniu tej metody można konfigurować dodatkowe opcje poprzez jej argument.

Podsumowanie

DS to jednak tylko jedno z istniejących rozwiązań zarządzania usługami pomiędzy modułami OSGi. Specyfikacja ta pojawiła się dopiero w ostatniej wersji standardu, pozostawiając czas projektom korzystającym z OSGi na utworzenie swoich własnych mechanizmów abstrakcji usług. Jednym ze zbliżonych narzędzi są punkty rozszerzeń znane programistom wtyczek w Eclipse. Innym, opisanym także w tym numerze, jest Spring-DM, który najprawdopodobniej wkrótce także stanie się częścią standardu OSGi w wersji 4.2. Każdy zainteresowany przetestowaniem DS powinien zajrzeć do Eclipse w wersji developerskiej (3.5), by wypróbować powstające tam narzędzia do konfiguracji komponentów. W znaczący sposób upraszczają one pracę z DS.

Java Team

dołącz do nas www.isolution.pl

Maven 2 - jak ułatwić sobie pracę, cz. I

Rafał Kotusiewicz

Zachęcony własnymi doświadczeniami z Mavenem 2 zapragnąłem podzielić się z Wami wiedzą, którą w ostatnich miesiącach zdobyłem na temat tego wspaniałego narzędzia. Dodam, że jeszcze rok temu byłem gorącym zwolennikiem Anta. Dzięki temu opiszę również sposób migracji projektu z Anta na Mavena – jeśli ma to oczywiście sens.

Założenie co do wiedzy czytelnika

Przystępując do czytania tego tekstu czytelnik powinien znać system operacyjny na tyle, żeby ustawić w nim zmienne środowiskowe i posłużyć się narzędziami do rozpakowania archiwów (zip lub tgz). Zakładam oczywiście, że wiedza ta jest na tyle niemagiczna, że każdy z Was ją posiada a jeśli nie to może posiąść w ciągu najbliższych kilku minut (jednak nie jest to przedmiotem naszych rozważań).

Instalacja Mavena

Aby zainstalować Maven w środowisku developerskim nie musi ono spełnić właściwie żadnych specjalnych wymagań poza dwoma podstawowymi. Po pierwsze do uruchomienie potrzebna jest Java w wersji 1.4 lub nowsza oraz wystarczająca ilość miejsca na dysku aby przechowywać repozytorium z bibliotekami. Dodam, że repozytorium w moim środowisku zajmuje około 200MB, myślę więc, że zarezerwowanie około 300-400MB z jednej strony zabezpieczy nasze potrzeby a z drugiej nie będzie stanowić zbyt dużego obciążenia dla dysku (tym bardziej, że powierzchnia dysku jest stosunkowo tania). Jeśli maszyna, na której zamierzacie zainstalować Mavena spełnia te niezbyt wygórowane wymagania czas zabrać się za instalację.

Czynność to niemal oczywista więc poświecimy jej tak niewiele miejsca jak to możliwe. Ze strony http://maven.apache.org/download.html ściągamy archiwum z Maven 2.0.9 odpowiednie dla używanego systemu operacyjnego. Plik ma około 2MB więc powinno zająć to krótką chwilkę w zależności od łącza którym dysponujemy. Ściągnięty plik rozpakowujemy do jakiegoś wygodnego dla nas katalogu (warto wybrać taki żeby unikać - kłopotliwych czasami - spacji w ścieżce), np. *C:\apache-maven-2.0.9*. Teraz należy zająć się ustawieniem zmiennych środowiskowych. Jeśli nie macie jeszcze ustawionej zmiennej JAVA_HOME to najwyższy czas to uczynić. Warto oczywiście dodać katalog $JAVA_HOME/bin (lub %JAVA_HOME%\bin) do zmiennej $PATH (lub %PATH%), choć nie jest to oczywiście koniecznie dla poprawnej pracy z Mavenem. Mając już pewność, że JAVA_HOME ustawiona jest poprawnie i wskazuje na główny katalog z JDK możemy ustawić dwie kolejne. Zmienna M2_HOME (lub MAVEN_HOME) musi wskazywać na główny katalog Mavena, czyli w naszym przykładzie *C:\apache-maven-2.0.9*. Teraz uaktualniamy zmienną PATH dodając do niej katalog M2_HOME\bin (lub MAVEN_HOME\bin). Sprawdzamy efekt poprzez ponowne otwarcie terminala (jeśli pracujemy z Windows) lub przeładowania środowiska (*nix, Mac OS X).

```
C:\Documents and Settings\Rafał>mvn --version
Maven version: 2.0.9
Java version: 1.5.0_16
OS name: "windows xp" version: "5.1" arch: "x86" Family: "windows"
```

Jeśli na waszych komputerach efekt jest taki sam to znaczy, że wszystko się udało. Jeżeli jednak są jakieś problemy należy sprawdzić poprawność ustawienia zmiennych środowiskowych (za pomocą polecenia echo $ZMIENNA w terminalu). Jeśli problem objawia się wyświetleniem długiego stosu wyjątków to być może archiwum z Mavenem zostało uszkodzone podczas transportu. Warto wtedy ściągnąć je ponownie wybierając inny serwer źródłowy. Jednak z doświadczenia wiem, że na tym etapie problemy jeśli się zdarzają to wynikają raczej z błędnych ustawień środowiska – tam też należy szukać ich przyczyn.

Pierwsze kroki

Aby zobaczyć jak łatwe jest korzystanie z Mavena w kilka sekund utworzymy szkielet naszego pierwszego projektu, najpierw go utworzymy po-

tem zaś zastanowimy się dokładnie co się stało i dlaczego. Wpisujemy w terminalu co następuje (zaznaczone pogrubionym tekstem):

```
C:\Dev\JaveExpress\Maven>mvn archetype:create \
-DgroupId=com.javaexpress \
-DartifactId=HelloMaven
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix:
'archetype'.
[INFO] ----------------------------------------
[INFO] Building Maven Default Project
[INFO]    task-segment: [archetype:create] (aggrega-
tor-style)
…
…
[INFO] BUILD SUCCESSFUL
[INFO] ----------------------------------------
[INFO] Total time: 7 seconds
[INFO] Finished at: Wed Sep 24 22:16:42 CEST 2008
[INFO] Final Memory: 8M/14M
[INFO] ----------------------------------------
```

Najpierw należy pewnie wyjaśnić czego zażądaliśmy... Poprosiliśmy Mavena aby utworzył dla nas szkielet standardowej aplikacji (pakowanej do jara) wraz z szablonem pliku opisującego nasz projekt. W sumie dużo i trwało to kilka sekund (za pierwszym razem może trwać nieco dłużej, ponieważ Maven potrzebne biblioteki ściąga ze zdalnego repozytorium do repozytorium lokalnego). Spójrzmy zatem co się pojawiło w wyniku naszych działań. W katalogu, w którym znajdowaliśmy się pojawił się katalog zgodny z wartością parametru `artifactId`, który podaliśmy tworząc projekt.

```
C:\Dev\JaveExpress\Maven>dir
...
2008-09-24  22:16    <DIR>          .
2008-09-24  22:16    <DIR>          ..
2008-09-24  22:16    <DIR>          HelloMaven
0 plik(ów)              0 bajtów
3 katalog(ów)  50 311 876 608 bajtów wolnych
```

HelloMaven to nasz projekt. Gdy zajrzymy do katalogu, w którym się znajduje, natrafimy na plik *pom.xml* (do którego zawartości będziemy wielokrotnie wracać) oraz katalog *src*. Pełna zawartość aktualnie wygląda tak:

```
|- pom.xml
`- src
|- main
|  `- java
|     `- com
|        `- javaexpress
|           `- App.java
`- test
`- java
`- com
`- javaexpress
`- AppTest.java
```

Wstępnie wyjaśnię, że katalog *src* zawiera – jak można się domyśleć – źródła aplikacji i testów (zgodnie z założeniem, ze testy to część aplikacji) podzielone wg przeznaczenia ze wstępnie zarysowaną strukturą pakietów zgodną z wartością parametru `groupId` podaną podczas tworzenia szkieletu. Zanim przejdziemy dalej poddajmy drobnej edycji plik *App.java*. Zamieńmy znajdujący się tam wiersz:

```
System.out.println( "Hello World!" );
```

na bliższy duchowi naszej przykładowej aplikacji:

```
System.out.println( "Hello Maven!" );
```

Teraz znajdując się w katalogu aplikacji (tego, w którym znajduje się plik *pom.xml*) wywołujemy polecenie:

```
C:\Dev\JaveExpress\Maven\HelloMaven>mvn package

[INFO] Scanning for projects...
[INFO] ----------------------------------------
[INFO] Building HelloMaven
[INFO]    task-segment: [package]
[INFO] ----------------------------------------
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered re-
sources.
[INFO] [compiler:compile]
[INFO] Compiling 1 source file to C:\Dev\Jave-
Express\Maven\HelloMaven\target\classes
[INFO] [resources:testResources]
[INFO] Using default encoding to copy filtered re-
sources.
[INFO] [compiler:testCompile]
[INFO] Compiling 1 source file to C:\Dev\Jave-
Express\Maven\HelloMaven\target\test-classes
[INFO] [surefire:test]
[INFO] Surefire report directory: C:\Dev\Jave-
Express\Maven\HelloMaven\target\surefire-reports

-------------------------------------------------
T E S T S
-------------------------------------------------
Running com.javaexpress.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0,
Time elapsed: 0.093 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] [jar:jar]
[INFO] Building jar: C:\Dev\JaveExpress\Maven\Hel-
loMaven\target\HelloMaven-1.0-SNAPSHOT.jar
[INFO] ----------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ----------------------------------------
[INFO] Total time: 5 seconds
[INFO] Finished at: Thu Oct 16 11:43:39 CEST 2008
[INFO] Final Memory: 7M/13M
[INFO] ----------------------------------------
```

Tym razem zostawimy cały listing, żeby wyjaśnić krok po kroku co się stało. Na początek Maven poszukuje pliku projektu (pom.xml), na podstawie którego może kontynuować swoje dzia-

łanie. Znalazł go ,więc informuje o przeprowadzanych czynnościach. Widzimy, że dostrzegł nasze pragnienie spakowania projektu, a żeby to zrobić musi przeprowadzić po kolei kilka faz swojego działania (o tym napiszę szerzej już za chwilę). Najpierw próbuje przetworzyć wszystkie zasoby, potem kompiluje pliki źródłowe Javy, potem przetwarza zasoby wymagane przez testy i kompiluje testy. Następnie uruchamia wszystkie testy i jako, że nie wystąpiły po drodze żadne problemy, pakuje całość do pliku jar. Aby upewnić się, że wszystko jest jak należy spróbujmy uruchomić program. Ale gdzie on jest?! – zapytacie. Otóż cały efekt pracy Mavena podczas budowania aplikacji znajduje się w katalogu target/ (tworzonym przez niego automatycznie) położonym na równi z katalogiem src. Tam należy szukać katalogu classes ze skompilowanymi plikami javy oraz pliku *.jar o nazwie dość rozbudowanej. Do nazwy wrócimy również niebawem. Tymczasem spróbujmy uruchomić program.

```
C:\Dev\JaveExpress\Maven\HelloMaven>java -cp .;target\HelloMaven-1.0-SNAPSHOT.jar com.javaexpress.App

Hello Maven!
```

Działa jak należy, możemy więc wrócić do przeanalizowania prostego póki co pliku *pom.xml*. Jego aktualna zawartość wygląda mniej więcej tak:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>com.javaexpress</groupId>
<artifactId>HelloMaven</artifactId>
<packaging>jar</packaging>
<version>1.0-SNAPSHOT</version>
<name>HelloMaven</name>
<url>http://maven.apache.org</url>
<dependencies>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.1</version>
<scope>test</scope>
</dependency>
</dependencies>
</project>
```

Jak widać na listingu jest to plik w formacie XML i opisuje nasz projekt. W aktualnej wersji zawiera naprawdę niezbędną ilość informacji (pozostałe, jak katalog ze źródłami itp. Maven czerpie ze znanej sobie konwencji) potrzebną do zbudowania projektu. Zacznijmy od góry. Elementy `groupId` oraz `artifactId` zawierają wartości podane przez nas podczas tworzenia projektu. Zatrzymajmy się więc by dokładnie wyjaśnić czym jest jedno i drugie. Grupa (reprezentowana przez `groupId`) jest ogólnym określeniem dla zestawu produktów, bibliotek czy też aplikacji, stanowiących razem pewną całość. Artefakt zaś (reprezentowany oczywiście przez element `artefactId`) to nazwa produktu w obrębie grupy. Zastanówmy się co to znaczy, by w przyszłości skorzystać z wniosków. Jeśli mamy aplikację, która składa się z trzech części: zestawu klas biznesowych (model aplikacji) oraz zestawu interfejsów DAO jako pojedynczej części, implementacji wspomnianych interfejsów oraz części klienckiej, np. aplikacji www, a całość stanowi pewną grupę logiczna, jeden właściwy produkt, to warto wszystkie te projekty umieścić w obrębie tej samej grupy (np. `com.company.crm`) a poszczególne części jako artefakty o nazwach np. `crm-api`, `crm-dao-impl`, `crm-web`. Kolejnym elementem jest wersja i wydaje mi się, że na tym etapie nie wymaga absolutnie żadnego komentarza (choć z wersjami związana jest pewna tradycja i funkcjonalność samego Mavena i warto wrócić później do tematu.). Ważne natomiast jest, że efektem spakowania projektu będzie plik o nazwie domyślnie składającej się z elementów `artefactId`, `version` oraz `packaging`. W naszym wypadku:

```
HelloMaven-1.0-SNAPSHOT.jar
```

Zawartość elementu `name` pojawia się w konsoli, gdy uruchamiamy maven'a aby zidentyfikować aktualnie przetwarzany projekt. Element url jest informacją o stronie domowej projektu. We właściwym czasie zobaczymy jak z tego skorzystać. Domyślnie jako strona domowa ustawiana jest strona Mavena. W tej chwili możemy ją tak pozostawić.

Ostatnim elementem, któremu przyjrzymy się teraz będzie wykaz zależności projektu. Wszystkie zależności objęte są wspólnym elementem `dependencies`, który zawiera wiele elementów `dependency`. Zależności to oczywiście biblioteki, z których korzystamy pisząc nasz kod. We wstępnie wygenerowanym pliku pom.xml widać jedną zależność, jest to biblioteka `junit`, którą wszyscy pewnie dobrze znają. Potrzebna jest tylko podczas testowania co jest zaznaczone w elemencie `scope`. Zanim wyjaśnię czym jest `scope`, powiem skąd się biorą biblioteki i dlaczego nie ma katalogu *lib* w projekcie.

Zależności projektu

Każdy projekt, nad którym pracujemy (poza tymi naprawdę najprostszymi) posiada jakieś zewnętrzne zależności, np. biblioteki do obsługi skrótów, logowania itp. Nie inaczej będzie tym razem, naszą główną klasę wyposażymy w mechanizm logowania. Możemy korzystać z `System.out.println(...)`, ale jak powszechnie wiadomo nie jest to dobry zwyczaj. Dlatego też wykażemy nieco finezji i wykorzystamy Logger dostępny na stronach apache.org. Jak zatem dołączyć bibliotekę do projektu? Właśnie w tym celu skorzystamy ze wspomnianego już elementu `dependency`. Za pomocą edytora otwieramy plik pom.xml i dodajemy następujący fragment (wytłuszczony):

```
…
<dependencies>
<dependency>
<groupId>commons-logging</groupId>
<artifactId>commons-logging</artifactId>
<version>1.1.1</version>
</dependency>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.1</version>
<scope>test</scope>
</dependency>
</dependencies>
…
```

Skąd jednak wziąć te informacje? Skąd Maven będzie wiedział jak dołączyć wymaganą bibliotekę? Gdzie fizycznie znajduje się plik biblioteki? Mnóstwo pytań – trzeba na nie odpowiedzieć. Najpierw rozwieję wątpliwości dotyczące tego skąd się wspomniane biblioteki biorą. Otóż istnieją rozsiane po sieci repozytoria takich bibliotek, każde z nich jest uporządkowane właśnie zgodnie z podaną hierarchią. Najpierw według grup, potem według artefaktów i na koniec według wersji. Można przyjrzeć się repozytoriom np. pod adresem http://repo1.maven.org/maven2 lub http://mirrors.ibiblio.org/pub/mirrors/maven2. Zróbmy chwilkę przerwy i zajrzyjcie pod te adresy. Miło popatrzeć ile wspaniałej pracy programistycznej zgromadzono w każdym z tych miejsc. Bezpośrednie przeglądanie tych repozytoriów jest jedną z metod poszukiwania zależności. Nie jest jednak metodą najlepszą. Znacznie wygodniej jest skorzystać z jednej z wyszukiwarek bibliotek. Jedną z dostępnych jest http://mvnrepository.com. Wyszukując logger do naszego przykładu (wiemy, że szukamy biblioteki commons-logging) w wyszukiwarce wpiszmy *commons-logging*. Biblioteka, której szukamy znajduje się na czwartej pozycji.

```
4. commons-logging » commons-logging
```

Spróbujmy objaśnić wynik tym bardziej, że na piątej pozycji mamy niezwykle podobny tekst.

```
5. commons-logging » commons-logging-api
```

Oczywiście pod każdym z wierszy wyniku znajduje się krótki opis pozycji. Nieprzypadkowo jest to zawartość elementu <description> z pliku pom.xml projektu (tutaj commons-logging). Cóż jednak odczytać z wierszy wyniku? Otóż po lewej stronie znaku » znajduje się nazwa grupy (groupId), po prawej zaś nazwa artefaktu (artifactId) elementu z listy dopasowanych do kryteriów wyszukiwania. Patrząc na opis jesteśmy w stanie zorientować się, że chodzi nam raczej o pozycję czwartą niż piątą. Kliknijmy więc w nazwę artefaktu aby przenieść się na listę dostępnych jego wersji. Na kolejnej stronie widać więcej szczegółów artefaktu oraz spodziewaną listę z wersjami (a także statystyki) i szablon elementu <dependency> gotów do skopiowania i wstawienia do pliku pom.xml naszego projektu. Jest to szablon sugerujący użycie najnowszej dostępnej wersji. Jeśli z jakichś powodów chcemy korzystać z wersji starszej możemy kliknąć w jedną ze znajdujących się na liście by przenieść się na kolejną stronę – tym razem z informacjami o tej wersji i otrzymać właściwy jej szablon.

Ostatnim dziś omawianym elementem jest `scope`. Definiuje on dwie rzeczy, po pierwsze dostępność klas z biblioteki dla klas projektu podczas kompilacji, testowania i uruchomienia oraz w zależności od sposobu pakowania projektu lub zastosowanych pluginów determinuje to czy zależność będzie dołączona do projektu czy też nie. Poniższe zestawienie - mam nadzieję - rozwieje wszystkie, lub chociaż większość, wątpliwości.

compile

Kompilacja, testowanie i uruchomienie. Biblioteka dostępna zawsze i dołączana do wynikowego artefaktu. Jest to domyślna wartość elementu scope ustawiana jeśli zostanie on pominięty przy definicji zależności

provided

Kompilacja i testowanie. Zakładamy, że biblioteka będzie dostarczona przez środowisko uruchomieniowe (kontener) nie jest więc dołączana (np. ServletAPI).

test

Kompilacja testów i testowanie. Dostępne tylko podczas kompilacji i uruchomienia testów.

runtime

Testowanie i uruchomienie. Biblioteka dostępna podczas uruchamiania testów oraz wymagana do działania aplikacji, jest więc dołączana do artefaktu wynikowego.

O konfigurowaniu zależności można napisać więcej, jednak myślę, że na tym etapie ta ilość informacji jest wystarczająca (co sugeruje, że w przyszłości będziemy do tego tematu wracać).

Teraz obiecana odpowiedź na pytanie „gdzie fizycznie znajdują się pliki bibliotek?" - bo przecież nie korzystamy przy każdej kompilacji z plików znajdujących się na zdalnych serwerach. Otóż nie, nie korzystamy z nich za każdym razem, tylko za pierwszym, czyli przy pierwszej potrzebie wykorzystania biblioteki. Odbywa się to tak - Maven poszukuje w lokalnym repozytorium (utworzonym podczas pierwszej próby utworzenia projektu) wskazanej biblioteki. Jeśli jej nie odnajdzie ściąga z któregoś ze zdalnych repozytoriów i instaluje w przestrzenie naszego lokalnego. Katalog z lokalnym repozytorium znajduje się zwykle w katalogu *.m2/repository* znajdującym się w naszym domowym katalogu. Tam też można odnaleźć strukturę podobną do tej ze zdalnych repozytoriów.

Kolejna ważna sprawa dotycząca samych zależności. Otóż, nie wszystkie biblioteki są dostępne w publicznych repozytoriach. Powody są rożne, czasem kwestie licencyjne a czasem są to zakupione od naszych dostawców fragmenty tworzonego systemu. A czasami po prostu korzystamy z jakiegoś leciwego kodu, którym po prostu nie ma się kto zająć i nie trafił dotąd do żadnego publicznego repozytorium. My jednak potrzebujemy tej biblioteki a jak wspominałem nie mamy katalogu *lib* w projekcie (pomijam projekt webowy i katalog *WEB-INF/lib* bo tam raczej się nie wrzuca żadnych plików, a nawet gdybyśmy to zrobili to nie ma jak poinformować Mavena żeby tam czegoś szukał). Musimy taką bibliotekę zainstalować w naszym lokalnym repozytorium. Robimy to za pomocą Mavena i jego pluginu do instalacji artefaktów. Spójrzmy na przykład. Do projektu chcemy dołączyć bibliotekę `super-tajny.jar` (kosztowała 1999$ i licencja nie pozwala umieścić jej w publicznym repozytorium). Kopiujemy wspomniany plik jar do bieżącego katalogu (dla wygody, bo przecież równie dobrze można posłużyć się pełną ścieżką wskazując aktualne miejsce pobytu jara) i wykonujemy następujące polecenie (pamiętając czym jest groupId, artifactId oraz version dla tego pliku):

```
mvn install:install-file -Dfile=super-tajny.jar
-DgroupId=com.dostawca.tajny
-DartifactId=super-tajny -Dversion=1.0
```

Teraz plik możemy już usunąć z bieżącego katalogu, przenieść się do katalogu z naszym projektem i wyedytować plik pom.xml dodające następujący kod (wytłuszczony):

```
<dependencies>
...
<dependency>
<groupId>com.dostawca.tajny</groupId>
<artifactId>super-tajny</artifactId>
<version>1.0</version>
<scope>runtime</scope><!-- lub inny w zalżności od
potrzeb -->
</dependency>
…
</dependencies>
```

Fazy budowania projektu

Teraz skoro już wiemy jak stworzyć projekt, jak dołączyć do niego biblioteki i jak go zbudować, spróbujemy dowiedzieć się jak ten proces wygląda. Na początek ustalmy jedną istotną rzecz – twórcy Mavena założyli, że z jednego projektu można zbudować jeden plik (pewną odmianą tej zasady są projekty wielomodułowe, ale o nich opowiem w następnym numerze). Mając to w pamięci zastanówmy się co można zrobić z projektem. Po pierwsze możemy zbudować ze źródeł plik wynikowy, na pewno też powinniśmy móc wyczyścić katalog roboczy przed kolejną próbą zbudowania projektu, na przykład po wprowadzeniu zmian w kodzie. Dodam, że w Mavenie dostrzeżona została również potrzeba tworzenia dokumentacji (nie tylko API) dla projektu, i ta potrzeba została opisana jako trzecia możliwość. Każda z nich, czyli budowanie, czyszczenie katalogu roboczego projektu oraz tworzenie dokumentacji zostały nazwane cyklami życia projektu (*ang. build lifecycle*). Te trzy zostały opisane cykle są podstawowymi dostępnymi w Mavenie. Każdy z takich cykli składa się oczywiście z mniejszych części zwanych fazami (*ang. phase*).

Jeśli dobrze przyglądaliście się wynikom dotychczasowej pracy i jej przebiegowi zwróciliście zapewne uwagę na charakterystyczne komunikaty Mavena takie jak `task-segment: [package]`, `[compiler:compile]` itp. Są to - w terminologii Mavena - *fazy budowania* projektu. Żeby w pełni je zrozumieć należy je rozpatrywać w kontekście

cykli budowy i *celów*. Aby problem skomplikować wspomnę jeszcze, że należy także pamiętać o typie projektu (znacznik `packaging` z pliku pom.xml). Brzmi to wszystko dość tajemniczo, ale przecież każdy zgodzi się, że źródła projektu zanim zamienią się w gotowy plik jar (lub inny) przechodzą pewne fazy. I właśnie te kilka etapów, które doprowadzają do gotowego produktu, połączone razem są *cyklem*. Każdy z etapów to oczywiście *faza* a ich ilość i rodzaj zależny jest od projektu, nad którym pracujemy.

Zacznijmy więc od początku. Twórcy Mavena przewidzieli trzy różne cykle pracy z projektem. Pierwszy to cykl budowy, składający się z faz, które po kolei uruchomione na końcu pozostawią po sobie wynikowy plik – jaki to plik to zależy od typu projektu. Drugi cykl to tworzenie dokumentacji w postaci nadającej się do publikacji w sieci www. Trzeci cykl to czyszczenie katalogu roboczego po pakowaniu lub tworzeniu strony.

Dla przykładu zastanówmy się co trzeba zrobić, żeby z kilkunastu plików źródłowych javy otrzymać bibliotekę w postaci pliku jar. Pewnie oprócz kodu javowego mamy zestaw jakichś plików `*.properties` zawierających tłumaczenia komunikatów lub jakiegoś rodzaju konfigurację. Pliki te wymagają wstępnego przetworzenia. Nazwijmy tę fazę budowy *process-resources*. Potem oczywiście kompilujemy klasy – niech ta faza nazywa się *compile*. Kolejnym etapem jest oczywiście przygotowanie do testów, oczywiście również testy mogą wymagać przygotowania pewnych plików `*.properties`, więc kolejna faza to *process-test-resources*. Teraz warto oczywiście skompilować klasy zawierające kod testów – nazwijmy tę fazę *test-compile*. Kolejnym krokiem jest oczywiście wykonanie testów, czyli *test*. Jeśli nasz kod bezbłędnie przechodzi testy możemy spakować klasy do pliku jar. I ta faza dostanie nazwę - a jakże! - *package*. Dwie kolejne rzeczy, które możemy zrobić z biblioteką, która została poprawnie zbudowana to zainstalować ją w lokalnym repozytorium oraz zainstalować w repozytorium publicznym. Każda z tych czynności jest dla nas oddzielną fazą i nazwiemy je odpowiednio *install* oraz *deploy*. Podsumujmy teraz proces budowy biblioteki. Poniżej widzimy jak on przebiega.

```
process-resources
compile
process-test-resourcess
test-compile
test
package
install
deploy
```

Widoczne tutaj fazy tworzą kompletny *cykl budowy* pakietu wynikowego dla projektu typu **jar**. Aby rozwiać wszelkie wątpliwości od razu wyjaśnię, że pakietem może być plik jar, aplikacja webowa w postaci pliku war, zestaw komponentów EJB w pliku jar, czy też archiwum EAR.

Skoro wiemy już czym jest cykl budowy i czym jest faza to jesteśmy gotowi na poznanie kolejnego terminu – cel (ang. goal). Czym jest? Otóż cel to najmniejsze zadanie do wykonania w cyklu życia. Oczywiście takich zadań może przypadać na cykl więcej niż jedno i jest to dość częsta sytuacja. Domyślnie jednak sytuacja wygląda znacznie prościej – na jedną fazę przypada jeden cel z konkretnego pluginu.

Poniżej rozpisane są poszczególne fazy dla projektu typu *jar* wraz z informacją jaki plugin jest dla każdej z nich uruchamiany. Zaraz wyjaśnię też co z tej wiedzy dla nas wynika.

Faza / Plugin / Cel (goal)
process-resources / maven-resources-plugin / resources
compile / maven-compiler-plugin / compile
process-test-resources / maven-resources-plugin / testResources
test-compile / maven-compiler-plugin / testCompile
test / maven-surefire-plugin / test
package / maven-jar-plugin / jar
install / maven-install-plugin / install
deploy / maven-deploy-plugin / deploy

Każdy z tych pluginów podczas właściwej sobie fazy cyklu jest wykonywany z domyślną konfiguracją. Jednak może nam ona z rożnych powodów nie odpowiadać. Warto więc ją nieco zmienić. Zrobimy to troszkę angażując znowu nasz przykładowy projekt. Sięgnijmy do pliku App.java i zmieńmy go dodając w sekcji importów dwa wiersze:

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
```

i nieco dalej w klasie App kolejny wiersz ze zmienną statyczną do logowania:

```
private final static Log log = LogFactory.getLog(App.class);
```

i wewnątrz pętli tuż pod naszym dumnym

„Hello Maven!” dość ekstrawagancką pętlę for w stylu Javy5:

```
for(String arg : args) {
log.info("arg => " + arg);
}
```

Teraz spróbujmy skompilować aplikację:

```
C:\Dev\JaveExpress\Maven\HelloMaven>mvn package
… (pominięty wydruk)
[ERROR] BUILD FAILURE
[INFO] ------------------------------------------
--------------------------
[INFO] Compilation failure
C:\Dev\JaveExpress\Maven\HelloMaven\src\main\ja-
va\com\javaexpress\App.java:[16,23]  for-each  loops
are not supported in -source 1.3
(try -source 1.5 to enable for-each loops)
for(String arg : args) {

C:\Dev\JaveExpress\Maven\HelloMaven\src\main\ja-
va\com\javaexpress\App.java:[16,23]  for-each  loops
are not supported in -source 1.3
(try -source 1.5 to enable for-each loops)
for(String arg : args) {
```

Od razu widać, że Maven zrozumiał o co nam chodzi, ale domyślna konfiguracja pluginu do przeprowadzania kompilacji zakłada, że kod jest zgodny z wersja 1.3 Javy (co nawet nie pozwala stosować wbudowanych *asercji*!). Należy więc skonfigurować moment kompilacji. Ale wciąż nie wiemy jak! Na stronie http://maven.apache.org/plugins/index.html znajduje się wykaz wbudowanych pluginów dostarczonych razem z Mavenen. Każdy z nich ma swoją stronę, na której można odnaleźć informacje o jego przeznaczeniu, działaniu, użyciu i konfiguracji – czyli wszystko czego nam potrzeba. Odnajdźmy więc plugin *compiler*. Na głównej stronie mamy już taką dawkę informacji, która na pewno rozwieje wszystkie nasze wątpliwości związane z tym pluginem. W części *Goals Overview* znajduje się informacja (podana w tabeli nr 1) o tym, który cel (*ang. goal*) kiedy jest uruchamiany. Teraz bardziej zainteresuje nas cześć *Usage*. Tam właśnie znajduje się instrukcja według której należy plugin skonfigurować. Pełne możliwości konfiguracji dostępne są po wybraniu w lewej części menu *Goals* a potem z listy *compiler:compile* (plugin:goal).

Wróćmy więc do naszego pliku pom.xml. Tuż za całą sekcją

```
<dependencies>
…
</dependencies>
```

tworzymy nową (build) i wprowadzamy to co widać na poniższym wydruku:

```
<build>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-compiler-plugin</artifactId>
<version>2.0</version>
<configuration>
<source>1.5</source>
<target>1.5</target>
</configuration>
</plugin>
</plugins>
</build>
```

Cały powyższy fragment jest dla nas nowy, ale najbardziej interesujące są wyróżnione części. Po pierwsze widać, że wszystkie pluginy są takimi samymi artefaktami jak pozostałe zależności projektu i dzięki zachowaniu tej samej hierarchii i zasady nazewnictwa możemy łatwo zorientować się do czego odnoszą się poszczególne fragmenty. Drugi wyróżniony fragment to część zawierająca konfigurację pluginu *maven-compile-plugin*. Wskazujemy jasno jaki poziom zgodności źródeł i klasy nas interesuje. Spróbujmy teraz zbudować projekt. Wśród wierszy, które odnajdziemy na ekranie terminala znajdzie się również taki:

```
[INFO] BUILD SUCCESSFUL
```

co znaczy, że tym razem Maven doskonale zrozumiał nasze intencje. Skoro udało się zbudować projekt (możemy to sprawdzić testując obecność pliku jar w katalogu *target/*) to czas na kilka kolejnych porcji informacji.

Idąc tym tropem możemy skonfigurować każdy dostępny plugin i użyć go w naszym projekcie. Zbliżając się ku końcowi tej części opowiem jeszcze gdzie należy szukać informacji o fazach w cyklach projektu. Według mnie najlepszym miejscem i – wbrew pozorom – najłatwiejszym do odczytania i nie pozostawiającym żadnych wątpliwości, są źródła Mavena. W drzewie kodu znajduje się plik *maven-core/src/main/resources/META-INF/plexus/components.xml*, opisujący dokładnie wszystkie cykle dla wbudowanych celów, na przykład wspomniany *jar-lifecycle*. Ciekawy jest cykl *clean*. Widać, że składa się z trzech faz,

```
pre-clean
clean
post-clean
```

ale tylko jedna jest w domyślnej konfiguracji wykorzystywana. Sprawdźmy czy rzeczywiście tak właśnie jest. Zaglądamy do katalogu naszego

projektu i wykonujemy poniższe polecenie:

```
C:\Dev\JaveExpress\Maven\HelloMaven>mvn clean
[INFO] Scanning for projects...
[INFO] ------------------------------------------
-------------------------
[INFO] Building HelloMaven
[INFO]    task-segment: [clean]
[INFO] ------------------------------------------
-------------------------
[INFO] [clean:clean]
[INFO] ------------------------------------------
-------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------
-------------------------
[INFO] Total time: 1 second
[INFO] Finished at: Wed Oct 29 23:55:40 CET 2008
[INFO] Final Memory: 3M/5M
[INFO] ------------------------------------------
-------------------------
```

Na powyższym wydruku widać, że zażądaliśmy wykonania cyklu *clean*. Ale z trzech wymienionych faz widać tylko jedną i uruchomiony skonfigurowany dla niej plugin. Co z pozostałymi dwoma? No cóż, pozostają do naszej dyspozycji. Możemy samodzielnie uruchomić wykonanie jakiejś czynności poprzez odpowiednią konfigurację. Poniżej przykładowe uruchomienie zadania Anta w pierwszej fazie cyklu *clean*:

```
...
<plugin>
<artifactId>maven-antrun-plugin</artifactId>
<executions>
<execution>
<phase>pre-clean</phase>
<configuration>
<tasks>
<echo>"Ant uruchomiony w Mavenie :)"</echo>
</tasks>
</configuration>
<goals>
<goal>run</goal>
</goals>
</execution>
</executions>
</plugin>
…
```

Oczywiście powyższe linie należy dodać do naszego roboczego pliku pom.xml, np. tuż za konfiguracją pluginu do kompilacji projektu. Gdy wykonamy w katalogu projektu *mvn clean* w konsoli pojawi się spodziewany komunikat:

```
C:\Dev\JaveExpress\Maven\HelloMaven>mvn clean
…
[INFO] Building HelloMaven
[INFO]    task-segment: [clean]
[INFO] ------------------------------------------
-------------------------
[INFO] [antrun:run {execution: default}]
[INFO] Executing tasks
[echo] "Ant uruchomiony w Mavenie :)"
[INFO] Executed tasks
[INFO] [clean:clean]
[INFO] ------------------------------------------
-------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------
-------------------------
...
```

czyli, dokładnie to czego się spodziewaliśmy. W podobny sposób możemy konfigurować dodatkowe czynności podczas każdej fazy dowolnego cyklu życia, możemy także zamieniać domyślnie skonfigurowane pluginy swoimi własnymi. Na tym właśnie polega ogromna elastyczność Mavena. W domyślnej konfiguracji jest niezwykle przydatny jednak skrzydła rozwija dopiero, gdy dodamy do niego szczyptę fantazji.

Kilka słów więcej o magii pliku pom.xml

Nie można pisać o zarządzaniu procesem budowania projektu za pomocą Mavena, nie opisując pliku `pom.xml`. To, że plik ten pełni kluczową rolę to już na pewno zauważyliście. Jego minimalną postać również – pojawia się ona po utworzeniu projektu. Ale na tym nie kończą się ani jego możliwości ani rola. Podsumowując tę część postaram się w miarę dokładnie (ale bez przesady, gdyż wiedzę tę będziemy rozwijać w kolejnych częściach cyklu o Mavenie) zapoznać Was ze strukturą pliku, oraz jego rolą w dystrybucji informacji o projekcie (zarówno podczas tworzenia dokumentacji jak i podczas wyszukiwania artefaktów za pomocą wspomnianych już narzędzi).

Spróbujmy podzielić cały plik na kilka części (jednak nieco inaczej niż w podręczniku dostępnym na stronach http://maven.apache.org/) zbierając elementy konfiguracji według ich przeznaczenia. Najpierw elementy identyfikujące projekt:

- *groupId* – informuje o grupie produktów, w której znajduje się projekt, zwykle (choć nie zawsze) składa się - na podobieństwo javowych pakietów - z domeny, dostawcy oraz nazwy całej grupy produktów
- *artifactId* – to bezpośrednia nazwa projektu w obrębie grupy, np. `crm-api`, `dao-jdbc` itp. (patrz uwaga wyżej)
- *version* – określenie numeru wersji, za pomocą liczb lub liczb mieszanych z opisem, np. 1.0, 2.1, 2.2-BETA, 2.2-RC1, itp. (wersjom przyjrzymy się dokładnie w kolejnym numerze)
- *packaging* – sposób pakowania projektu, który jednocześnie określa także fazy cyklu budowy dla projektu (inne nieco dla różnego rodzaju pro-

jektów)

Kolejną grupą będą elementy opisujące projekt, dostawcę oraz twórców:

- *name* – nazwa dla projektu, nieco bardziej opisowa niż zbitek składający się z `grupId` oraz `artifactId`, nazwa ta będzie używana w kilku miejscach, warto wybrać taką, która będzie kojarzyć się z projektem, ale nie będzie długa, np. SuperCRM.API – dla klasa i interfejsów stanowiących API projektu

- *description* – opis projektu, może być dowolnie długi, ale warto zachować umiar, gdyż jeśli do wyszukiwania artefaktów będących wynikiem naszej pracy będziemy używać wyszukiwarki podobnej do opisanej na początku artykułu (oczywiście przeszukującej nasze firmowe lub prywatne repozytorium) to zawartość sekcji `description` pojawi się w wynikach wyszukiwania

- *url* – tutaj można umieści adres witryny przechowującej informacje o projekcie, (adres w wersji pełnej np. http://dev.supercrm.com/ lub http://supercrm.com

- *organization* – element zbiorczy zawierający nazwę i adres strony domowej (zwyczajowo) organizacji zajmującej się rozwijaniem projektu, czyli – w przypadku klasycznej działalności komercyjnej - po prostu firmy, która zajmuje się jego tworzeniem; składa się z dwóch zagnieżdżonych elementów:

- *name* – nazwa organizacji

- *url* – adres strony domowej organizacji

- *inceptionYear* – rok rozpoczęcia prac nad projektem, wartość ta zostanie wykorzystana w notce o prawach autorskich w dokumentacji generowanej przez Mavena

- *licenses* – licencje na rozwijany produkt (może być ich oczywiście więcej niż jedna), każdą licencję podajemy w oddzielnym elemencie `<licence></licence>`

- *developers* – podobnie jak w przypadku licencji jest to elementy zbiorczy, zawierający jeden lub więcej elementów `developer`, z których każdych składa się z następujących elementów

- *id* – identyfikator developera, pozwalający np. powiązać go z informacjami o zmianach w systemie kontroli wersji bądź systemie zgłaszania błędów itp.

- *name* – imię i nazwisko (lub pseudonim) developera

- *email* – sprawa wydaje się oczywista :)

- *organization* – jeśli developer pracujący nad projektem na stałe zatrudniony jest (w przypadku działalności komercyjnej) lub związany (w przypadku działalności non profit) z organizacją inną niż rozwijając produkt to można podać tu jej nazwę

- *organizationUrl* – j.w., ale zawieraj informacje o stronie domowej macierzystej organizacji develepera

- *roles* – element zbiorczy zawierający pojedyncze elementy `role` informujące o roli developera w projekcie (np. jee-developer, architect, webmaster, itp.)

- *contributors* – informacje podobne jak te zawarte w elemencie `developers`, tutaj jednak znajdują się z założenia informacje o osobach zgłaszających błędy (nie uczestniczące w samym procesie wytwarzania jako dostawcy kodu), dane poszczególnych osób (dokładnie te same co w przypadku developerów) otaczane są elementem `contributor`.

Sporo już się dowiedzieliśmy o części informacyjnej pliku pom.xml. Spróbujmy zamienić tę wiedzę na rzeczywiste korzyści. Otwórzmy więc ten opisujący nasz projekt i uzupełnijmy jak poniżej:

```
...
<name>HelloMaven</name>
<url>http://javaexpress.pl/hellomaven</url>
<description>
Projekt przykładowy. Pomaga zrozumieć działanie Ma-
vena.
</description>
<organization>
<name>JaveExpress</name>
<url>http://javaexpress.pl</url>
</organization>
<inceptionYear>2008</inceptionYear>
<licences>
<licence>
<name>Apache 2</name>
<url>http://www.apache.org/licenses/LICENSE-
2.0.txt</url>
<distribution>repo</distribution>
<comments>Ulubiona licencja organizacji</comments>
</licence>
</licences>
<developers>
<developer>
<id>jjhop</id>
<name>Rafal Kotusiewicz</name>
<email>r.kotusiewicz@javaexpress.pl</email>
<url>http://javaexpress.pl/author/jjhop/</url>
<organization>JavaExpress</organization>
<organizationUrl>http://javaexpress.pl</organiza-
tionUrl>
<roles>
<role>architect</role>
<role>developer</role>
</roles>
<timezone>-1</timezone>
<properties>
<picUrl>http://javaexpress.pl/author/jjhop/mi-
ni.png</picUrl>
</properties>
</developer>
</developers>
<contributors>
<contributor>
```

```
<!-- tutaj odpowiednie dane -->
</contributor>
</contributors>
<dependencies>
...
```

Kolejna grupa jest dość szeroka, łączy bowiem informacje o tym gdzie szukać brakujących zależności jeśli nie ma ich w lokalnym repozytorium (*repositories) z informacjami porządkującymi dane projektu. Zaczniemy od konfiguracji repozytoriów, w których Maven będzie szukał brakujących *artefaktów*. Składają się na nią dwa elementy głównej konfiguracji wraz z elementami podrzędnymi:

- *repositories* – repozytoria, w których Maven będzie szukał głównych zależności projektu, czyli tych wymienionych wewnątrz elementu `dependencies`, są to jednocześnie repozytoria, do których w razie potrzeby może „wysłać" efekt naszej pracy (o tym więcej w przyszłości)

- *pluginRepositories* – to dodatkowe repozytoria, w których będą poszukiwane dodatkowe pluginy służące np. do specjalnego sposobu budowania (pakowania) projektu, do szczególnego typu raportowania itp...

Każdy z tych elementów zawiera elementy podrzędne – odpowiednio `repository` i `pluginRepository` – składające się z informacji pozwalających zidentyfikować repozytorium oraz „zajrzeć" do jego zawartości. Wspomniane informacje to:

- *id* – identyfikacja repozytorium, jeśli jest chwilowo niedostępne to dzięki temu identyfikatorowi Maven przez jakiś czas nie będzie próbował nawiązać z nim połączenia

- *name* – przyjazna nazwa repozytorium pojawiająca się podczas logowania informacji o przebiegu procesu budowania (lub którego z dwóch pozostałych) projektu

- *url* – pełen adres repozytorium (razem z protokołem), najczęściej w stylu http://repo.xxx.org/maven2, ale może także zaczynać się od ftp://

Bardziej szczegółowa konfiguracja repozytoriów zostanie opisana w kolejnych częściach wraz z odpowiednimi przykładami tak, aby nie zakłócać normalnego toku poznawczego :). Cztery kolejne elementy informacyjne to:

- *issueManagement* – wskazuje system zarządzania błędami i potrzebami tworzonego oprogramowania, element składa się z dwóch podrzędnych (patrz → przykład użycia w naszym projekcie)

- *ciManagement* – konfiguracja systemu CI (ang. *continuous integration*) obsługującego projekt (np. Continuum – najchętniej wykorzystywany wraz z Mavenem, Cruise Control), główny element zawiera elementy podrzędne zawierające dokładne informacje (patrz → przykład użycia w naszym projekcie)

- *mailingLists* – informacje o listach mailingowych związanych z projektem, główny zawiera elementy podrzędne opisujące konfigurację poszczególnych list, z których każda zawieraj następujące dane:

- *name* – nazwa listy wysyłkowej

- *subscribe* – adres email, pod który należy wysłać wiadomość by zapisać się na listę

- *unscubscribe* – j.w., ale z odwrotnym efektem

- *post* – adres email, na który wysyła się wiadomości na listę

- *archive* – URL wskazujący archiwum listy dostępne przez www (jeśli istnieje taka możliwość)

- *otherArchives* – zbiór pojedynczych elementów `otherArchive` zawierających informacje tożsame z tymi z elementu `archive`

- *scm* - informacje o systemie kontroli wersji, w którym przechowywane są źródła oraz inne pliki projektu (patrz → przykład użycia w naszym projekcie)

Kolejnym ważnym elementem jest *distributionManagement*, którego zadaniem jest precyzyjne opisanie sposobu (oraz statusu) dystrybucji artefaktów będących wynikiem budowy projektu a także adresy repozytoriów, w których będą instalowane. Także tutaj podajemy sposób i miejsce, w którym umieszczona będzie strona informacyjna projektu (to jak ją stworzyć będzie ostatnim elementem tej części artykułu) – mam oczywiście na myśli raport o stanie projektu w wersji html.

W wypadku tego elementu na razie nie będziemy posługiwać się przykładem i wrócimy do niego w jednej z kolejnej części artykułu, gdy zajmiemy się zarządzaniem procesem budowy.

Ostatnią częścią opisującą magię pliku pom.xml będzie dość dokładna konfiguracja budowania projektu. Proces budowania opisany jest – a jakże! - w ramach elementu o nazwie `build`. Tak naprawdę to domyślna konfiguracja jest dość dobra. Większość domyślnych wartości można zaaprobować bez żadnych zmian, czasami jednak trzeba coś zmienić. Wtedy właśnie do akcji wkracza konfiguracja w obrębie elementu `build`. Trzy poniższe elementy wydają się dobrym przykładem na początek:

- *finalName* – tutaj możemy zdefiniować własną wartość dla nazwy pliku wynikowego (np. su-

percrm.war zamiast supercrm-1.0-SNAP-SHOT.war)

- *defaultGoal* – domyślny cel procesu budowania, jeśli zdefiniujemy go tutaj, nie będzie trzeba podawać go w wierszy poleceń, oczywiście jeśli mimo zdefiniowania go, wywołamy Mavena z inną wartością to właśnie ona zostanie wzięta pod uwagę

- *directory* – katalog, w którym będą przechowywany wszystkie wyniki – także pośrednie – procesu budowania (domyślnie ma wartość *${basedir}/target*, przy czym *${basedir}* to katalog projektu)

- *sourceDirectory* – katalog, zawierający pliki źródłowe (*.java) projektu

- *testSourceDirectory* – katalog, w którym znajdują się pliki źródłowe (*.java) zawierające testy

- *outputDirectory* – katalog, do którego trafią skompilowane klasy projektu

- *testOutputDirectory* – katalog, do którego trafią skompilowane klasy testowe

Kolejnym ważnym elementem, który tu możemy skonfigurować są katalogi z zasobami projektu. Pamiętamy, że domyślnie takim miejscem jest katalog *${basedir}/src/main/resources* (oraz *${basedir}/src/test/resources*), czasami zachodzi jednak potrzeba wskazania więcej niż jednego katalogu jeśli z jakichś powodów chcemy np. rozdzielić różne ich rodzaje. Katalogi zasobów głównych obejmujemy elementem zbiorczym `resources`, katalogi dla procesu testowania elementem `testResources`. W każdym z nich możemy zdefiniować jeden lub więcej elementów `resource` o następującej strukturze:

- *directory* – wskazuje katalog zawierający interesujące nas zasoby

- *targetPath* – katalog, do którego po przetworzeniu trafią zasoby z directory

- *filtering* – (true lub false) informacja o tym, czy pliki zasobów mają podlegać przetwarzaniu przez mechanizm podstawiania zmiennych

- *includes* – zawiera jeden lub więcej elementów include wskazujących wprost lub za pomocą znaków wieloznacznych, które elementy z katalogu directory mają podlegać przetworzeniu i włączeniu do wyników procesu budowania

- *excludes* – jw. lecz zawiera elementy exclude i dotyczy wyłączenia z procesu budowania.

Poniżej przykładowa konfiguracja zawierająca wspomniane elementy:

```
...
<build>
<defaultGoal>install</defaultGoal>
<directory>${basedir}/target</directory>
<finalName>supercrm.war</finalName>
<sourceDirectory>${basedir}/src/main/java</source-
Directory>
<testSourceDirectory>${basedir}/src/test/java</te-
stSourceDirectory>
<outputDirectory>${basedir}/target/classes</output-
Directory>
<testOutputDirectory>${basedir}/target/test-clas-
ses</testOutputDirectory
<resources>
<resource>
<directory>${basedir}/src/main/psd_images</directo-
ry>
<targetPath>img</targetPath>
<filtering>false</filtering>
<includes>
<include>**/*.jpg</include>
</includes>
<excludes>
<exclude>**/*.psd</exclude>
</excludes>
</resource>
</resources>
...
</build>
…
```

Kolejną rzeczą, którą chcemy konfigurować w tym miejscu to oczywiście *pluginy*, które obsługują proces budowania. Wspominałem już dość szeroko o powiązaniu konkretnych pluginów z fazami budowy projektu, wspominałem także o tym, że można je konfigurować (oczywiście zakres i rodzaj konfiguracji zależy od konkretnych pluginów). Widzieliśmy już, że można konfigurując plugin obsługujący kompilację wskazać mu poziom zgodności źródeł (czasami wręcz nie można się bez tej możliwości obyć). Można także konfigurować inne jego opcje (np. kodowanie znaków w kodzie źródłowym) a także różnego rodzaju zachowanie innych pluginów. Do tego celu został przewidziany element `plugins` w `build`. Zawiera jeden lub więcej elementów plugin, z których każdy oprócz elementów identyfikujących (`groupId`, `artifactId`, `version`) zawiera inne sterujące jego działaniem. Oto one:

- *extensions* – (*true* lub *false*) informuje o tym czy plugin jest rozszerzeniem Mavena czy też nie (w jednej z kolejnej części opiszę jak zbudować własny tym projektu, jak np. jar lub war, wtedy też użyjemy w tym miejscu wartości *true*, poza tym rzadko tu zmienia się wartość domyślną, która jest oczywiście *false*)

- *inherited* - (*true* lub *false*) ma zastosowanie tylko w przypadku projektów wielomodułowych (o których opowiem w kolejnym odcinku), decyduje o tym czy konfiguracja jest dziedziczona przez moduły (konkretnie przez pliki pom.xml w poszczególnych modułach

- *configuration* – zestaw elementów, których

nazwy określają rodzaj konfiguracji o zawartość jej wartość (np. <encoding>utf-8</encoding> - encoding jest zmienną konfiguracyjną a utf-8 jest wartością, coś na kształt encoding=utf-8), elementów może być wiele i w zależności od modułu mogą być zagnieżdżone

- *dependencies* – zależności modułu, rzadko wykorzystywana chyba, że uruchamiamy jakieś zadanie ant'a, które potrzebuje dodatkowych bibliotek, ponad te, które dostarczane są z Antem

- *executions* – każdy plugin może być uruchomiony podczas więcej niż jednej faz i może zostać wykonany więcej niż jeden jego cel (ang. *goal*), można więc skonfigurować każde wykonanie oddzielnie; element ten zawiera jeden lub więcej elementów *execution*

Tym razem to już wszystko o konfiguracji w elemencie *build*. Przykład nieco uproszczony pojawił się na stronie 10 więc wydaje się, że nie warto go powtarzać. Więcej pojawi się jednak z czasem, kiedy będziemy rozbudowywać nasz projekt w trakcie poznawania Mavena. Teraz czas na ostatnią część artykułu poświęcona tworzeniu strony informacyjnej projektu.

Wizytówka projektu, czyli www w minutę

Na sam koniec zostawiłem coś, co niezwykle podoba mi się, gdy korzystam z Mavena. Otóż, gdy projekt rozwija się (nieważne czy jest to projekt komercyjny czy opensource) nadchodzi taki moment, że należy pewne informacje dotyczące go umieszczać w ogólnie dostępnym miejscu – najczęściej jest to sieć www (być może firmowy intranet). Zespół pracujący nad rozwojem oprogramowania pragnie podzielić się swoimi osiągnięciami, czasami po prostu potrzeba raportów na temat jakości kodu podanych w przystępnej formie. Powodów tworzenia stron tego typu jest wiele. A co z tym wspólnego ma Maven? Jednym z trzech cykli budowy projektu jest właśnie tworzenie jego dokumentacji do postaci plików html – myślę oczywiście o cyklu *site*.

Zaglądamy więc do katalogu projektu i próbujemy to w następujący sposób:

```
C:\Dev\JaveExpress\Maven\HelloMaven>mvn site
[INFO] Scanning for projects...
[INFO] ------------------------------------------------
---------------------------
[INFO] Building HelloMaven
[INFO]    task-segment: [site]
[INFO] ------------------------------------------------
---------------------------
…
[INFO] ------------------------------------------------
---------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------
---------------------------
…
```

Po chwili w katalogu target/site pojawi się mnóstwo plików *.html a wśród nich index.html. Właśnie od niego zacznijmy. Otwórzcie go w przeglądarce, powinien wyglądać mniej więcej tak jak widać to na rysunku.

Przyglądając się stronie zauważycie znajome informacje, począwszy od tych dotyczących ogólnych informacji o projekcie (element *description* z pliku pom.xml), które pojawiają się tu w części *About* poprzez wszystkie dane konfiguracyjne jak adres repozytorium, adres systemu kontroli wersji czy adresy list wysyłkowych związanych z projektem aż po skład zespołu rozwijającego projekt. Zachęcam do przejrzenia tego co wynika wprost z najprostszego zastosowania cyklu *site* i zmierzenia się z kolejną porcją możliwości skonfigurowania Mavena według własnych potrzeba. Sprawdźmy zatem co możemy jeszcze zrobić w kwestii strony informacyjnej projektu. Obsługę cyklu *site* konfigurujemy w obrębie elementu *reporting*, który jest elementem równorzędnym do *build* i innych. W jego obrębie możemy skonfigurować:

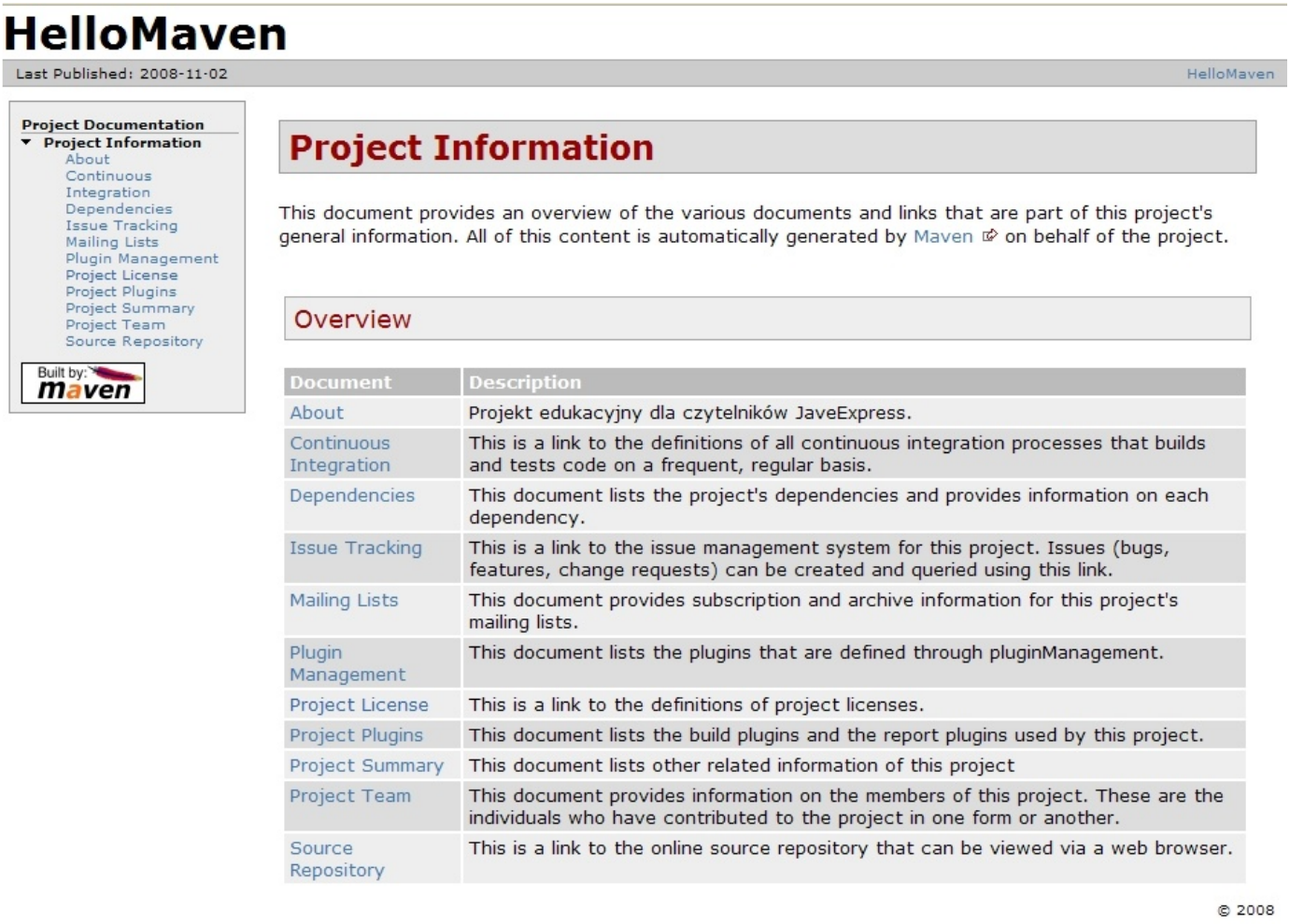
- *outputDirectory* – katalog, w którym znajdzie się wygenerowana strona

- *plugins* – kolekcja pluginów (w postaci elementów *plugin*), które zostaną uruchomione podczas tworzenia dokumentacji, są to pluginy generujące różnego rodzaju raporty o projekcie

Kilka przydatnych pluginów jest dostępnych w repozytoriach Mavena. Ich opis oraz konfigurację możemy znaleźć po adresem http://maven.apache.org/plugins/index.html w sekcji ***Reporting plugins***. Ich wykorzystanie jest niezwykle proste. Wystarczy umieścić dane identyfikacyjne wybranego pluginu w obrębie elementu *plugins*.

```
…
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-changelog-plugin</artifactId>
</plugin>
</plugins>
…
```

Korzystając z linków na wspomnianej stronie i dokumentacji dołączonej do każdego pluginu można dostosować konfigurację do własnych potrzeb. Nie będziemy teraz opisywać szczegółów, myślę, że każdy z Was uzbrojony w informacje zawarte w



artykule chętnie samodzielnie poeksperymentuje z raportami.

Obiecuję, że przy okazji kolejnego spotkania pokaże jak skonfigurować projekt tak, by maksymalnie skorzystać z możliwości jakie daje automatyczne tworzenie dokumentacji projektu.

Co w kolejnej części?

Kolejna część (czyli już w marcu) wprowadzi nas w tajniki składania projektu z kilku części. Zobaczymy jak korzystać z Mavena w naszych ulubionych środowiskach – Netbeans, IntelliJ Idea oraz Eclipse. Podłączymy nasz projekt do repozytorium, systemu śledzenia błędów oraz system CI (skorzystamy oczywiście z Continuum). Rozbudujemy także nieco stronę projektu. I przede wszystkim – mam nadzieję – będziemy się dobrze bawić!

O autorze

Od przełomu wieków zajmuję się rozwojem oprogramowania, aktualnie jako architekt - programista w firmie Aegis Media Sp. z o.o.. W swojej pracy zwykle wykorzystuję Javę i technologie z nią związane, jednak bardzo chętnie z domieszką języków skryptowych (Perl, Python, Ruby, Groovy). Wśród wielu języków, z którymi spotkałem się przez lata mojej pracy największe wrażenie zrobił na mnie (i robi do dziś) Ruby, z którym jestem blisko związany już od blisko sześciu lat. Nieco hobbystycznie *kolekcjonuję* stare języki (Algol, PL/I, Fortran) a właściwie ich dokumentacje i kompilatory, przyglądam się jak zmieniał się ich świat na przestrzeni lat.

Express killers, cz. I

Damian Szczepanik

Z przyjemnością witam w nowym cyklu *Express killers*. Dział będzie prezentował ciekawe (nie koniecznie zalecane!) przypadki użycia języka Java. Jestem zwolennikiem korzystania z IDE i wszystkich jego możliwości, które oferuje, dlatego problemy tutaj poruszane nie będą pytaniem, dlaczego poniższy kod się nie komplikuje?

```
public class Object {
  Object o = new Object() {
    public String toString() {
      return null;
    }
  }
}
```

Nie będę także czytelników zanudzał pytaniami o wynik takiego wyrażenia:

```
int i = 0x12<<3+04^5>>06>>>7|8&9;
```

bo przecież po to są nawiasy, żeby sobie życie ułatwiać oraz internet, żeby nie trzeba było pamiętać, który operand będzie pierwszy brany do wyliczenia.

Co zatem będzie tematem? Najlepsze są przykłady z życia wzięte. Ot np. taki:

```
import java.util.Vector;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Pool {

  private static Logger logger = Logger
      .getLogger(Pool.class.getName());
  private Vector<String> messages;
  private Integer status;

  public boolean add(String message) {
    logger.log(Level.ALL, "addding:" + this);
    return messages.add(message);
  }

  public String get(int index) {
    logger.log(Level.ALL, "getting:" + this);

    return messages.get(index);
  }

  public String toString() {
    StringBuffer sb = new StringBuffer();
    int i = 1;
    do {
      sb.append(get(i)).append(",");
    } while (++i < messages.size());
    sb.append(status);
    return sb.toString();
  }

  public static void main(String[] args) {
    Pool p = new Pool();
    p.add("hello");
    System.out.println(p.get(0));
  }
}
```

Biorąc pod uwagę, że powyższy kod się kompiluje, które z poniższych odpowiedzi dotyczą sytuacji po jego uruchomieniu?

- Uruchomienie zakończy się błędem `java.lang.ArrayIndexOutOfBoundsException`, gdyż w metodzie `toString()` nieprawidłowo inicjalizowana jest pętla.
- Gdyby zmienna `status` była prymitywem, program nie zakończyłby się wyjątkiem.
- Program zakończy się poprawnie, a na wyjście zostanie wypisany napis `hello`.
- Wyjątek `java.lang.NullPointerException` zostanie rzucony w pętli metody `toString()`.
- Żadna z powyższych odpowiedzi nie jest prawidłowa.

Odpowiedź na to pytanie pozostawiamy czytelnikowi, który może uruchomić powyższy kod w swoim ulubionym środowisku programistycznym. Może także poszukać jej na kolejnych stronach Java exPress.

A oto i drugie zadanie. Jego specyfika nie przeszkadza, by wkleić następującą metodę do swojego IDE i sprawdzić, jak się zachowuje po uruchomieniu. Oto kawałek kodu pewnej klasy:

```
public static void main(String[] args)
    throws Exception {
  CatchNull test = new CatchNull();
  if (test == null) {
    throw new NullPointerException("null object");
  }
  if (!(test instanceof CatchNull)) {
```

```
    throw new UnknownError(
        "not an instanceof CatchNull");
  }
}
```

W jakim przypadku uruchomienie powyższego kodu spowoduje rzucenie jednego z dwóch wyjątków? Odpowiedź, podobnie, jak do poprzedniego pytania, na dalszych stronach. Gorąco także zachęcam do poszukania jej samemu!

Serdecznie zapraszam wszystkich, by pochwalili się z czytelnikami swoimi interesującymi kawałkami kodu. Najlepsze będą publikowane i nagradzane na łamach czasopisma. Mogą to być także kawałki kodu z cyklu *WTF*. Praca codziennie dostarcza interesujące przykłady, po przeczytaniu których programista zaczyna sprawdzać elementarne podstawy swojej wiedzy. Niech zaczną ją też sprawdzać inni za pośrednictwem Java exPress.

Oferta dla sponsorów

JAVA exPress powstaje dzięki bezinteresownej pomocy autorów artykułów, którzy poświęcili swój czas, aby pomóc zrealizować pomysł stworzenia pierwszej gazety o Javie. Gazety, która jest dostępna za darmo, dla każdego, kto chciałby poszerzyć swoja wiedzę o języku Java.
Jeśli więc chciałbyś, aby gazeta nadal się ukazywała, a artykuły były coraz lepszej jakości możesz wspomóc naszą inicjatywę.
Niezależnie czy jesteś osobą prywatną, czy przedstawicielem firmy, skontaktuj się z nami pod adresem kontakt@dworld.pl.
Pomóż rozwijać zdolności swoich obecnych i przyszłych pracowników.

Zostań autorem

Czasopismo JAVA exPress ukazuje się co 3 miesiące. Poszukujemy osoby, które chciałyby zostać autorami artykułów do gazety lub pomóc w jej rozwoju (np. poprzez przygotowanie graficzne i skład).
Jeśli chcesz zostać autorem lub w inny sposób pomóc gazecie, wyślij swoja propozycję na adres kontakt@dworld.pl.
Każdy, kto pomaga tworzyć JAVA exPress otrzymuje "brykiety", które później może wymienić na cenne przedmioty: książki, wejściówki na konferencje, licencje, czy inne gadżety Javowe.
Więcej informacji na stronie http://java-express.pl

Wspomóż JAVA exPress

Jeśli podoba Ci się czasopismo i chciałbyś, żeby nadal się ukazywało i podnosiło swój poziom zarówno merytoryczny jak i wizualny, możesz nam pomóc. Pomóż nam w składzie tekstu, garfice lub korekcie.
Możesz także wspomóc nas finansowo. Każdy grosz się liczy. I pamiętaj, zanzacz w treści przelewu, czy chciałbyś, aby Twoja nazwisko pojawiło się na liście dobrodziejów.
Więcej szczegółów na stronie:
http://dworld.pl/wspomoz-java-express/

Odpowiedzi do Express killers:

W pierwszym przykładzie w metodzie `main(String[])` jest wywołanie metody `get(int)` co spowoduje zapisanie tej informacji w logach. Użycie słowa kluczowego `this` zakończy się wywołaniem metody `toString()`. W metodzie `toString()` następuje iteracja po wszystkich elementach kolekcji. Chociaż pętla nie jest napisana poprawnie, to przy pierwszej iteracji nie zostanie zgłoszony żaden wyjątek. Zostanie natomiast wywołana metoda `get(int)`. W konsekwencji wygenerowany zostanie wyjątek o przepełnieniu stosu `java.lang.StackOverflowError`.

W drugim przypadku żaden wyjątek nie zostanie rzucony. Pierwszy z nich byłby rzucony, gdyby zmienna `test` nie została poprawnie stworzona. Miałoby to miejsce, gdyby konstruktor rzucił wyjątek lub gdyby maszyna wirtualna nie posiadała wystarczającej ilości pamięci do stworzenia nowego obiektu. Gdyby tak się stało, to rzucony wyjątek nie zostałby chwycony, a zatem pierwsza instrukcja warunkowa nie zostanie wykonana. W takim przypadku także druga instrukcja warunkowa nie zostanie sprawdzona.

Druga instrukcja warunkowa przy deklaracji zmiennej test jako `CatchNull` nie ma sensu. Kompilator (lub maszyna wirtualna, jeśli zastosować rzutowanie) nie dopuści, aby do zmiennej typu `CatchNull` został przypisany obiekt, który nie jest typem `CatchNull` lub pochodnym. Jedynym przypadkiem, by warunek drugi był prawdziwy jest, by zmienna `test` była równa `null`, co według wcześniejszego wywodu nie jest możliwe.