Numer 1/2009(3)

Java eXPress

CZASOPISMO DLA DEWELOPERÓW JAVA

Dworzec główny: GWT dla początkujących

Bocznica: J2ME: Serializacja obiektów, cz.1

Dworzec główny: Nie ma nic za darmo – biblioteki Open Source

Konduktor: Wzorce projektowe: Temporal Object

- Kubek Kawy - czyli alternatywny kurs Javy, cz. III
- Testowanie metod prywatnych
- Programowanie bez słowa redeploy...

Maszynista

jep.setGraphics();

Zaczynając cały pomysł z czasopismem dotyczącym Javy nie miałem pojęcia jak ten temat zostanie przyjęty. Już od samego początku nie było łatwo: poszukiwania autorów, patronów itp. Niemniej jednak po wydaniu pierwszego numeru "coś zaskoczyło". Zaczęło się od zbiórki pieniędzy na hosting, później zgłoszenia potencjalnych autorów, a skończywszy na dwóch osobach, bez których JAVA exPress nie wyglądałby tak, jak wygląda. Mowa tutaj o Marku i Jakubie.

Marek Podsiadły zgłosił się do pomocy już dawno temu i dzięki niemu mamy artykuły w wersji html. A żeby nam wszystkim było łatwiej, napisał aplikację, której efektu widzicie na http://javaexpress.pl poniżej "W ostatnim numerze". Tak tak, to się generuje automagicznie dzięki Markowi. Co więcej, Marek kończy pisać aplikację (w Grails), która zastąpi mało efektywny arkusz i dokument na Google Docs, do naliczania brykietów i aukcji.

Jakub Sosiński natomiast, to nasz nadworny grafik. Pomógł dostosować logo Developers World oraz JAVA exPress, zrobił projekt roll-up, który mieliście okazję zobaczyć na COOLuarach i 4Developers, oraz przygotował nowa szatę graficzną naszego czasopisma. Mam nadzieję, że Wam się podoba.

Jeśli macie jakieś uwagi co do pisma, lub chcielibyście pomóc go tworzyć lub napisać artykuł, to zapraszam do kontaktu mailowego: kontakt@dworld.pl.

Do zobaczenia 1 czerwca...

P.S. Zdjęcie z okładki możecie znaleźć na
http://www.flickr.com/photos/nikonvscanon/504339356/

Pozdrawiam,
Grzegorz Duda

Plan podróży

4Developers Kraków 2009

7 marca odbyła się pierwsza edycja multikonferencji 4Developers. Czemu mutli? Ponieważ na jednej konferencji można było posłuchać o Javie, .Net, Zarządzaniu projektami i o językach specjalizowanych. Świetny pomysł, świetna organizacja i świetni prelegenci. Czego można więcej chcieć od konferencji? Wg. JAVA exPress najlepszy wykład poprowadził Neal Ford, ze swoim sztandarowym tematem - produktywność developera.

Ale to nie był jedyny dobry wykład. Zarówno Ted, Adam, Waldi, Jacek i przedstawiciele firmy e-point nie zamierzali zaniżać poziomu konferencji.

Już wkrótce na stornie konferencji (http://4developers.org.pl/) znaleźć będzie można prezentacje i materiały wideo. A więc zamiast się rozpisywać zapraszam do oglądania.

No i oczywiście na 4Develoeprs nie mogło zabraknąć JAVA exPress (jako patrona medialnego) i prowadzonego przeze mnie Java Underground. Ciekaw jestem jak tym razem podobały się Wam lightning talki.

COOLuary

2 otwarte dyskusje o Javie - Gdansk, 23-24 maja 2009

COOLuary - pierwsza Open Space Conference o Javie w Polsce już za nami (http://dworld.pl/cooluary/). Zebraliśmy sporo doświadczenia, pomysłów i pochwał. A więc nie pozostaje nic innego, jak zabrać się za organizację kolejnej edycji COOLuarów. Więcej informacji na stronie 8.

JAZOON09

THE INTERNATIONAL CONFERENCE ON JAVA TECHNOLOGY
JUNE 22–25, 2009 ZURICH

Jazoon zbliża się wielkimi krokami. Już od 22 czerwca przez 3 dni w Zurychu będzie głośno o Javie. Dla czytelników JAVA exPress specjalna 40% zniżka od cen na stronie. Zainteresowanych zapraszam do kontaktu: kontakt@dworld.pl. Oczywiście JAVA exPress objęło patronatem medialnym to niezwykłe wydarzenie.

GeeCON

Let's move the Java world!
May 7-8, 2009 Cracow, Poland

Jeśli za daleko Wam do Zurychu, a nie mogliście uczestniczyć w 4Developers, to jeszcze nic straconego. Już 7-8 maja, w Krakowie, odbędzie się w pełni międzynarodowa konferencja o Javie. GeeCON (http://geecon.org/), bo o nim mowa, będzie trwał 2 dni, a każdego dnia będzie można wybrać jedną z dwóch ścieżek. Na każdej ścieżce wiele wspaniałych prelegentów nie tylko z zagranicy, ale także naszych rodzimych – polskich. Rejestracja już rozpoczęta. Do końca marca zniżkowe ceny, a wśród osób zarejestrowanych do 15 marca rozlosowane zostaną licencje na IntelliJ IDEA oraz JavaRebel. Tak, Java exPress także objęło patronatem GeeCONa.

Java4People 04.04.2009 SZCZECIN

Jak widać ten rok obfituje w wydarzenia Javowe. Nie sposób tutaj nie wymienić kolejnych dwóch konferencji. Pierwsza z nich, Java4People, odbędzie się 4 kwietnia w Szczecinie. Druga natomiast, Javarsovia, planowana jest na 27 czerwca, ale więcej szczegółów dowiecie się w kolejnym numerze JAVA exPress. W tym momencie warto nadmienić, że obydwie te konferencje są darmowe.

Skład tekstu i wybór tematów: Grzegorz Duda
Grafika: Jakub Sosiński
Artykuły html i aplikacje: Marek Podsiadły
kontakt: kontakt@dworld.pl

Kubek Kawy - czyli alternatywny kurs Javy, cz. III

Bartek Kuczyński

Długo zastanawiałem się nad tym co powinniśmy zrobić na pierwszych prawdziwych zajęciach z Javy (czytaj będziemy w końcu programować). Weźmiemy się za programowanie, bo większą część teorii już przebrnęliśmy w dwóch poprzednich częściach, ale co będziemy programować.

Środowisko

Zanim napiszemy pierwszy program należy włączyć sobie jakieś pisadło. Na rynku jest wiele środowisk przeznaczonych do programowania. Generalna nazwa to IDE od angielskiego „Integrated Development Environment" - zintegrowane środowisko programowania. Narzędzie takie zapewnia zazwyczaj kilka podstawowych opcji jak:
- podświetlanie składni
- sprawdzanie poprawności składni
- automatyczna konfiguracja kompilatora
- debugger

Pierwsze dwa punkty są minimalnym minimum. Bez tego nie można mówić o IDE nawet jak o edytorze kodu. Trzeci punkt nie jest obowiązkowy, ale zazwyczaj dostarczany nawet z najprostszymi edytorami. Ostatni punkt też nie jest obowiązkowy, ale IDE aspirujące do miana poważnego narzędzia powinno posiadać debugger.

Od tego punktu zaczynamy prawdziwą alternatywność tego kursu. Zazwyczaj tradycyjne kursy prowadzone na uczelniach czy też w ramach wydawnictw książkowych zawierają magiczne zdanie „otwórz ulubiony edytor tekstu". Co pewnego razu kolega podsumował „Można pisać w Excelu?". Oczywiście, że można, pytanie brzmi czy nie łatwiej użyć do tego zadania odpowiedniego narzędzia? Ja nie powiem wam o ulubionym edytorze. Od dziś przez pewien czas waszym ulubionym edytorem jest Eclipse 3.4.1 Classic. Pobieramy go w następujący sposób:
- Wchodzimy na www.eclipse.org.
- Naciskamy „Download".
- Z listy wybieramy „Eclipse 3.4.1 Classic".
- Obok napisu „Download from" będzie link do pliku.
- Zapisujemy plik na dysku.
- Rozpakowujemy zawartość.
- Uruchamiamy.
- Zostaniemy poproszeni o wskazanie „Workspace" jest to katalog roboczy tu będą nasze projekty.
- Naciskamy ctrl+n i z listy wybieramy Java - Java Project. Można sobie pomóc wpisując na górze okna „java project"
- Nazywamy nasz projekt w jakiś rozsądny sposób, na przykład „Kubek Kawy fajny kurs java".
- Naciskamy „finish". Jak wyskoczą nam jakieś okienka to naciskamy „OK". Na tym etapie raczej nic jeszcze nie zepsujemy.
- Powinno ukazać się nam coś w stylu rysunku 1

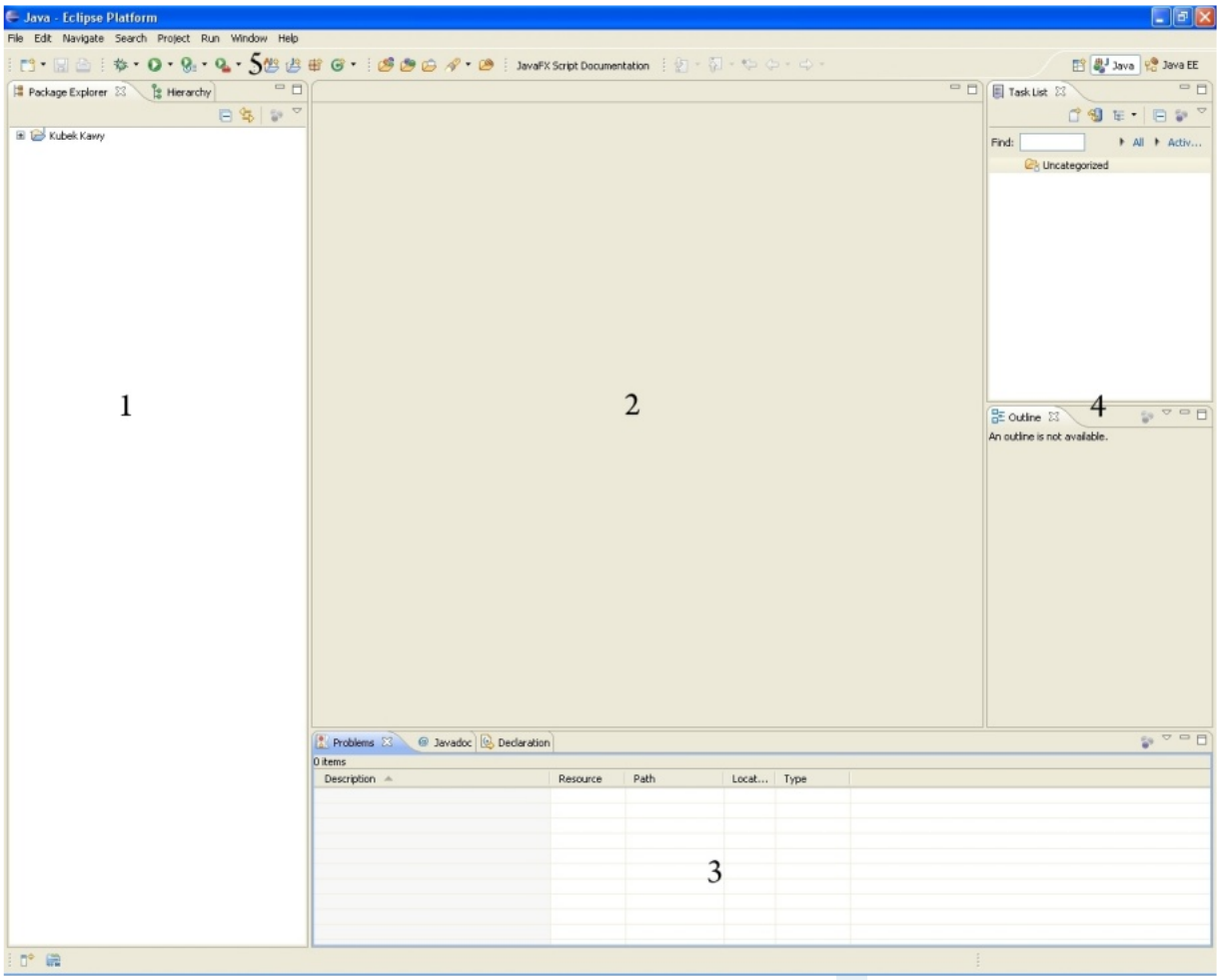
Teraz jeszcze kilka słów co my tu mamy i bierzemy się do pisania kodu.

1. Lista projektów, plików, bibliotek w projektach. W praktyce podgląd aktywnej zawartości przestrzeni roboczej.
2. Tu będą otwierać się pliki. Edytor.
3. Słówko „Problems" mówi samo za siebie. Tu też będzie podgląd konsoli.
4. Różne pomocne okna. Później sam odkryjesz co i jak.
5. Masa różnych przycisków. Pozwalają na tworzenie nowych plików, zapisywanie, drukowanie, ale też na uruchamianie kompilatorów czy debuggerów.

W miarę posuwania się naszych prac odkryjesz zapewne do czego służy jakieś 90% opcji i przycisków. Na razie nie zawracaj sobie tym głowy. Możesz poeksperymentować, ale zakładam, że nie bardzo znasz się na programowaniu więc i tak niczego nie wykorzystasz.

Pierwszy program

Czas na pierwszy program. Naciskamy ctrl+n wybieramy „Java-package" i jako nazwę podajemy nazwę pakietu. Pakiety służą przede wszystkim do logicznej organizacji kodu. W praktyce organizują go też fizycznie na dysku. Działają



Rysunek 1. Coś co będzie wam się śniło po nocach

trochę jak adres internetowy, zapewniając nie tylko podział logiczny, ale też unikalność nazw. O tym wkrótce. Na razie piszemy pierwszy program. Ja mój pakiet nazwałem `pl.koziolekweb.javaexpress.kubekkawy.cz3`. Znowu naciskamy ctrl+n i wybieramy „java-class”. Nazywamy naszą nową klasę „PierwszyProgram” i zaznaczamy opcję „public static void main(String[] args)”. Otrzymamy coś takiego jak poniżej:

```
1.  package pl.koziolekweb.javaexpress.kubekkawy.cz3;
2.
3. public class PierwszyProgram {
4.
5.   /**
6.    * @param args
7.    */
8.      public static void main(String[] args) {
9.     // TODO Auto-generated method stub
10.
11.   }
12. }
```

Metoda main(String[] args) jest punktem od którego startuje każdy program w języku Java. Tyle trzeba obecnie o niej wiedzieć. Zamiast standardowego „Hello world!” sprawdźmy który dzień tygodnia dziś mamy. Program realizujący to zadanie wygląda tak:

```
1. package pl.koziolekweb.javaexpress.kubekkawy.cz3;
2.
3. import java.util.Calendar;
4. import java.util.GregorianCalendar;
5.
6. public class PierwszyProgram {
7.
8.    /**
9.     * @param args
10.    */
11.   public static void main(String[] args) {
12.     Calendar calendar = GregorianCalendar.getInstance();
13.     int dzień = calendar.get(Calendar.DAY_OF_WEEK);
14.     System.out.println("dziś mamy " + dzień + " dzień tygodnia");
15.   }
16. }
```

Niedziela jest pierwszym dniem tygodnia, poniedziałek drugim itd. W liniach 3 i 4 znajdują się

polecenia importu odpowiednich klas. Jeżeli chcemy użyć jakiejś klasy w programie musimy powiedzieć kompilatorowi czego chcemy używać. Java ma ścisłą kontrolę typów, ale też przymusowe dostarczanie implementacji klas i interfejsów w procesie kompilacji. Jeżeli spotkałeś się z językiem C/C++ to pamiętasz pewno, że dało się tam wymusić kompilację kodu bez sprawdzania czy istnieją odpowiednie zależności z poleceń include. W javie takiego przekrętu nie da się zrobić. Szkoda, ale z drugiej strony ma to zalety. Oczywiście nie wszystko trzeba importować. W 14 linii jest używana klasa System, ale nie ma importu. Kompilator automatycznie importuje wszystkie klasy z pakietu java.lang. Jeżeli mamy do zaimportowania kilka klas z jakiegoś pakietu to można zastąpić je zapisem „z gwiazdką”, czyli zamiast nazwy klasy wrzucamy '*' i tyle. Oczywiście IDE po formatowaniu kodu ograniczy to do wymaganych klas. Zresztą, czy to jest ważne na tym etapie? Moim zdaniem nie.

Naciskamy mały zielony przycisk na górze. W ten sposób zostanie uruchomiony nasz program. Na dole otworzy się podgląd konsoli, w którym powinniśmy zobaczyć:

```
dziś mamy 7 dzień tygodnia
```

No to na tyle jeśli chodzi o pierwszy program. Czas zająć się czymś zabawniejszym niż wypisywanie głupot w konsoli.

Interakcja z użytkownikiem

No nie do końca porzucimy konsolę. Na chwilę obecną bez sensu jest pchanie się w fajerwerki graficzne, jak nie do końca potrafimy programować proste rzeczy. Co za dużo to niezdrowo. Metoda `main()`, jako parametr przyjmuje tablicę obiektów `String`. Tablica ta reprezentuje argumenty przekazane do programu za pomocą linii poleceń. Czas zatem trochę skomplikować zadanie i sprawdzać czy dziś mamy dzień podany przez użytkownika. W tym celu musimy zrobić dwie rzeczy. Po pierwsze dopisać odpowiedni kod do naszego programu, ale to za chwilę. Po drugie umieć uruchomić go z parametrami. W tym celu wybieramy z górnego menu Run>Run Configuration> zakładka Arguments i w polu „Program arguments” wpisujemy interesujący nas numer dnia tygodnia. Pamiętamy, że tydzień ma siedem dni, numerowanych od 1, a pierwszym dniem jest niedziela. Klikamy „Run”, zaskoczyło to ok. Czas na programowanie. Ostrzegam, kod będzie niebanalny.

```
1. package pl.koziolekweb.javaexpress.kubekkawy.cz3;
2.
3. import java.util.Calendar;
4. import java.util.GregorianCalendar;
5. import java.util.regex.Matcher;
6. import java.util.regex.Pattern;
7.
8. public class PierwszyProgram {
9.
10.   /**
11.    * @param args
12.    */
13.   public static void main(String[] args) {
14.     if (args.length == 0) {
15.       System.out.println("Nie podano argumentów!");
16.       System.exit(0);
17.     }
18.     String dzieńPodanyPrzezUżytkownika = args[0];
19.
20.     Pattern pattern = Pattern.compile("^[0-9]+$");
21.     Matcher matcher = pattern.matcher(dzieńPodanyPrzezUżytkownika.trim());
22.
23.     if (!matcher.matches()) {
24.       System.out.println("Proszę podać liczbę jako parametr");
25.       System.exit(0);
26.     }
27.
28.     Calendar calendar = GregorianCalendar.getInstance();
29.     Integer dzień = calendar.get(Calendar.DAY_OF_WEEK);
30.     Integer dzieńPPUInteger = Integer.parseInt(dzieńPodanyPrzezUżytkownika);
31.
32.     System.out.println("Czy dziś jest " + dzieńPodanyPrzezUżytkownika
33.         + " dzień tygodnia? "
34.         + (dzień.equals(dzieńPPUInteger) ? "tak" : "nie"));
35.   }
36. }
37.
```

> *Fajne? Nie jakieś tam prościutkie wypisywanie parametrów jak w kursach bywa.*

Fajne? Nie jakieś tam prościutkie wypisywanie parametrów jak w kursach bywa. Linie 14 do 17 zawierają proste sprawdzenie, czy przekazano parametry. Jeżeli użytkownik nie przekazał parametrów, długość tablicy jest równa 0, to wyświetlany jest komunikat i program kończy działanie. W linii 18 tworzymy sobie zmienną, której jedynym zadaniem jest sprawienie, że kod będzie czytelniejszy. Można oczywiście pracować, ze zmienną args, ale może to spowodować zaciemnienie kodu. Pamiętajmy, że nazwy zmiennych powinny mieć nazwy znaczące. Patrząc na zmienną powinniśmy wiedzieć co reprezentuje.
Linie od 20 do 26 służą do sprawdzenia, czy podany parametr jest liczbą całkowitą większą od 0. Do przeprowadzenia tej operacji użyłem wyrażeń regularnych. Jest to narzędzie, które pozwala analizować tekst przy pomocy pewnych wzorców. Na razie nie trzeba o tym więcej wiedzieć, ale jak jesteś zainteresowany to odsyłam do Wikipedii. Podstawowe informacje zamieszczam w ramce.

Oczywiście pada pytanie, dlaczego analizę robimy za pomocą tak trudnego zagadnienia jakim są wyrażenia regularne. Odpowiedź brzmi ponieważ jest to prawidłowe podejście. Bardzo często jako przykład takiego sprawdzania jest podawany kod podobny do poniższego:

```
1. String liczbaString = "000";
2. try {
3.      Integer liczbaInteger = Integer.parseInt(liczbaString);
4. } catch (NumberFormatException e) {

5.      System.out.println("podana liczba jest nieprawidłowa!");
6. }
```

Ten kod to „tru zuo” w najczystszej postaci. Wywodzi się z czasów, gdy wiele rzeczy można było zaprogramować za pomocą obsługi przerwań systemowych. Mówi się też czasami o programowaniu sterowanym wyjątkami. My mamy Javę, która posiada odpowiednie mechanizmy i narzędzia wydajnie zastępujące pewne pomysły.
W linii 30 następuje zamiana napisu na liczbę. Wszystkie klasy reprezentujące liczby mają metody `parse()`, które pozwalają zamienić ciąg znaków na liczbę. Ostatnie linijki programu są dość skomplikowane, ale nikt nie powiedział, że będzie łatwo. Generalnie jeżeli wykonujemy dodawanie obiektów `String` i innych obiektów to domyślnie wywoływane są metody `toString()` lub przeprowadzana konwersja z typów prostych. W linii 34 wykorzystany jest dodatkowo operator trójargumentowy w celu uniknięcia dodatkowej instrukcji `if`.

Pełna interakcja

Na zakończenie tej części mam dla ciebie zadanie domowe. Poniżej przedstawiam program, który zamiast argumentów będzie pytał użytkownika o liczbę i mówił czy odpowiada ona aktualnemu dniu tygodnia. Na jego podstawie napisz program, który będzie odpowiadał na pytanie o datę urodzin. Czyli użytkownik będzie podawał datę, a program zdecyduje czy użytkownik ma dziś urodziny czy też nie.

W kolejnym odcinku

Przechowywanie danych. Interakcja z użytkownikiem ciąg dalszy. Dwie rzeczy na raz – bardzo podstawy wątków.

Wyrażenia regularne – regexp

Wyrażenia regularne służą do analizowania tekstu za pomocą wzorców. Tekst jest „przykładany" do wzorca i sprawdzany czy spełnia warunki. Tworzenie wzorców to skomplikowane zagadnienie, ale podstawowe wzorce można opanować w kilka minut. W Javie wzorzec jest podawany jako ciąg znaków. Istnieją też inne sposoby podawania wzorców w innych językach. Podstawowe wzorce:

Cyfry:
[0-9] lub \d – dowolna cyfra
\D lub [^0-9] – dowolny znak nie będący cyfrą
Białe znaki:
\s – dowolny znak biały
\S – dowolny znak nie będący znakiem białym
\t – znak tabulacji [Tab]
\n – znak nowej linii
\r – znak powrotu karetki
Litery:
[a-zA-Z] lub \w – dowolna litera
[^a-zA-Z] lub \W – dowolny znak nie będący literą
Znaki:
. – dowolny znak
Krotności:
Krotność określa ile razy znak może się powtórzyć.
x{n} – n-krotne wystąpienie znaku *x*
x{n,} – co najmniej n-krotne wystąpienie znaku *x*
x{0,n} – co najwyżej n-krotne wystąpienie znaku *x*
x{n,m} – co najmniej n-krotne i maksymalnie m-krotne wystąpienie znaku *x*
x+ – co najmniej jednokrotne wystąpienie *x*
x? – nie więcej niż jednokrotne wystąpienie *x* odpowiednik *x{0,1}*
*x** – możliwość wystąpienia *x* odpowiednik *x{0,}*
Znaki specjalne:
^ – jeżeli jest umieszczony na samym początku wzorca oznacza początek wzorca. W przeciwnym wypadku oznacza zaprzeczenie.
$ – umieszczony na końcu wzorca oznacza koniec wzorca. Nie należy mylić ze znakiem końca linii.
\ – znak „ucieczki" jeżeli znak za tym znakiem jest znaczący lub specjalny to jego cechy zostaną zignorowane i będzie traktowany jak zwykły tekst.

Należy pamiętać, że w Javie obiekt `String` ma dodatkowe właściwości dotyczące znaków specjalnych. Szczególnie należy uważać na ilości znaku backslash '\'. Jeżeli będzie go za mało lub za dużo to może okazać się, że wzorzec jest nieprawidłowy lub oznacza coś innego. Poniżej przykładowe wzorce, które różniąc się ilością znaków backslash oznaczają zupełnie różne rzeczy:

1. `"^\d$"` – nieprawidłowa sekwencja znaków. Program się nie skompiluje.
2. `"^\\d$"` – jedna cyfra. Znak ucieczki zastosowany tylko do znaku '\', działa w ramach obiektu `String`. Wyrażenie regularne „widzi" tylko jeden znak '\'.
3. `"^\\\d$"` – nieprawidłowa sekwencja. Program się nie skompiluje.
4. `"^\\\\d$"` – ciąg znaków „\d", a nie cyfra. Znak ucieczkowy zastosowany najpierw jak w przykładzie 2, a następnie dodatkowo w ramach wzorca. Przy czym należy pamiętać, że uciekamy też od znaku '\' we wzorcu. Swoista ucieczka do znaku ucieczki.

Na zakończenie do analizy pozostawiam prosty wzorzec. Zgadnij co opisuje:

```
"^[-]?[0-9]*\\.?[0-9]*$"
```

```
1. package pl.koziolekweb.javaexpress.kubekkawy.cz3;
2.
3. import java.util.Calendar;
4. import java.util.GregorianCalendar;
5. import java.util.Scanner;
6. import java.util.regex.Matcher;
7. import java.util.regex.Pattern;
8.
9. public class PierwszyProgram {
10.
11.     /**
12.      * @param args
13.      */
14.     public static void main(String[] args) {
15.             Scanner scanner = new Scanner(System.in);
16.
17.             System.out.println("Podaj liczbę do sprawdzenia");
18.             String dzieńPodanyPrzezUżytkownika = scanner.nextLine();
19.
20.             Pattern pattern = Pattern.compile("^[0-9]+$");
21.             Matcher matcher = pattern.matcher(dzieńPodanyPrzezUżytkownika.trim());
22.
23.             if (!matcher.matches()) {
24.                     System.out.println("Prosze podać liczbę");
25.                     System.exit(0);
26.             }
27.
28.             Calendar calendar = GregorianCalendar.getInstance();
29.             Integer dzień = calendar.get(Calendar.DAY_OF_WEEK);
30.             Integer dzieńPPUInteger = Integer.parseInt(dzieńPodanyPrzezUżytkownika);
31.
32.             System.out.println("Czy dziś jest " + dzieńPodanyPrzezUżytkownika
33.                             + " dzień tygodnia? "
34.                             + (dzień.equals(dzieńPPUInteger) ? "tak" : "nie"));
35.     }
36. }
```

COOLuary

2 otwarte dyskusje o Javie - Gdansk, 23-24 maja 2009

24 stycznia odbyła się pierwsza UnConference o Javie w Polsce. Wydarzenie to zostało bardzo dobrze odebrane w społeczności Javowej. Dlatego też już 23-24 maja odbędzie się druga edycja COOLuarów. Tym razem w Gdańsku. Termin i miejsce może jeszcze ulec zmianie (więcej informacji na http://dworld.pl/cooluray/).

Pietwszy dzień, to tradycyjna UnConference, zwana także jako OpenSpace Conference. Typ konferencji mocno promowany przez samego Bruce'a Eckela. A więc wiele technicznych dyskusji na wysokim poziomie.

Drugi dzień będzie czymś na pograniczu UnConference, a warsztatów. Podczas 3 godzinnych spotkań, gdzie punktem centralnym będzie laptop, uczestnicy prowadzeni przez doświadczonego prowadzącego, będą mieli okazję zapoznać się z daną technologią. 2 sesje: poranna i popołudniowa, oraz możliwość wyboru tematu "szkolenia" wraz z nieformalnych charakterem i duża interakcją pomiędzy uczestnikami i prowadzącym powoduje, że czas spędzony na "szkoleniu" będzie niezapomniany.

Więcej informacji na temat drugiej edycji COOLuarów będzie można wkrótce znaleźć na stronie http://dworld.pl/cooluary/

Free Spring 3.0 Talk with Arjen Poutsma

Using or planning to use Spring 3.0? The Warszawa JUG, the Polish JUG and SpringSource invite you to two evening talks designed to increase your knowledge of Spring 3.0 These talks are appropriate for developers and engineers who are looking for expert advice on how to optimize and manage Spring 3.0.

This seminar will include topics as:

* New features in Spring 3.0
* Performance tuning Spring 3.0
* An overview of Spring Projects
* Enterprise capabilities for Spring

Don't miss this exclusive opportunity to speak directly with one of the leading Spring committers as well as other Spring users.

Arjen Poutsma is a senior software engineer at SpringSource with fifteen years' experience in commercial software environments. During this time he has worked with both Java EE and Microsoft .NET. Three years ago, Arjen started to specialise in Web Services and Service Oriented Architectures. During this period, he has worked for some of the largest organizations in the world helping them better understand enterprise Java and how SOAs fit into their organizations.

In part from his experiences with these organizations, Arjen founded the open source Spring Web Services project and continues to lead the technical direction and development as the project lead for Spring Web Services. This project aims at facilitating development of document-driven web services. Arjen has also contributed to various other open source projects, including XFire, Axis2, and others. He is a regular speaker at Java and SOA conferences, including JavaPolis, The Spring Experience, JavaZone, W-JAX, and many others.

For more information or to register visit:

9 March, 17:00 PM - 20:00 PM - Warsaw: http://springtalkwarsaw.eventbrite.com/

2 April, 17:00 PM - 20:00 PM – Cracow: http://springtalkcracow.eventbrite.com/

Professional Spring Framework Training Cracow - Warsaw

SpringSource, the company that created, built and sustains Spring, will also be offering its flagship training course Core Spring in Warsaw and in Cracow. SpringSource has trained over 3500 people, from developers to lead architects. The hands-on courses are known worldwide as the best in the industry, and the SpringSource trainers develop and consult on Spring 100% of their time – bringing real world experience on the latest Spring technology to the class. In this four-day intensive course you will learn how to use the Spring Framework to create well-designed, testable and maintainable business applications.

You will learn to:

* Work with the Spring Inversion of Control (IoC) Container
* Effectively use JDBC and Hibernate for data access
* Use JUnit, Spring, stubs and mocking frameworks to effectively implement automated unit and integration tests
* Take advantage of Aspect-Oriented Programming (AOP) to keep code clean and maintainable
* Use Spring Security to secure web and standalone applications
* Manage live applications with ease using Spring's support for Java Management Extensions (JMX)
* and much more

This professional course will be provided by Arjen Poutsma, the cost for this professional training is zł 7500,- However the Warszawa JUG and the Polish JUG made a discount available of 15%. For more information or to sign up visit:

10-13 March – Warsaw: http://www.springsource.com/training/spr001/waw-20090310

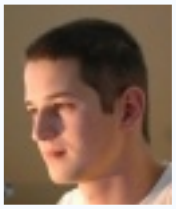
06-09 April - Cracow: http://www.springsource.com/training/spr001/krk-20090406

GWT dla początkujących

Paweł Stawicki

Co to jest GWT?

Na początku wypadałoby wyjaśnić co to takiego Google Web Toolkit. Bardzo ogólnie można by powiedzieć, że jest to kompilator, potrafiący przetworzyć (skompilować) kod Javy na JavaScript, mający też kilka innych ułatwiających życie funkcji. Dzięki temu pozwala na pisanie aplikacji webowych działających po stronie klienta, bez znajomości JavaScriptu. Oprócz tego daje nam kilka ułatwień, których nie mielibyśmy programując w czystym JavaScript. Jednym z nich jest obsługa różnych wersji JavaScriptu - niestety występują w nich różnice na różnych przeglądarkach. Poza tym łatwo tworzy się serwisy RPC (Remote Procedure Call), pozwalające aplikacji działającej po stronie klienta na komunikację z serwerem. Jeśli w Twojej głowie wyświetliło się słowo "AJAX", to bardzo słusznie. GWT to nic innego jak kolejne narzędzie do tworzenia aplikacji AJAXowych, jednak istotnie różniące się od "konkurencji".



Paweł Stawicki zajmuje się Javą od 2003 roku. Obecnie pracuje jako „software engineer” w firmie NCDC. Interesuje się także Ruby'm. Jest co-leaderem Szczecińskiego JUGa (http://szczecin.jug.pl) i autorem bloga http://pawelstawicki.blogspot.com. Z zamiłowania żeglarz :)

Prosta aplikacja kliencka

GWT dostarczane jest z narzędziami do tworzenia projektu. Tworzą one potrzebną strukturę katalogów i podstawowe pliki. Można też stworzyć projekt gotowy do zaimportowania w Eclipsie. Polecenie:

```
projectCreator -eclipse java-express -out java-express
```

Stworzy pusty projekt eclipse'a.

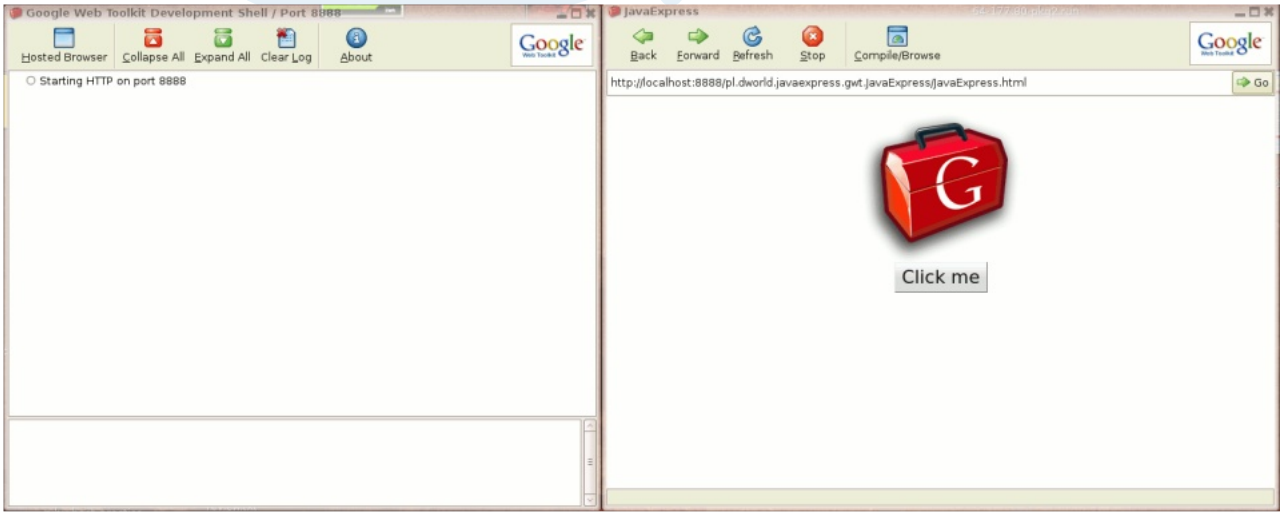
Polecenie

```
applicationCreator -out java-express -eclipse java-express pl.dworld.javaexpress.gwt.client.JavaExpress
```

Utworzy przykładową aplikację, ze skryptami do kompilowania i uruchamiania. Modyfikator

`-eclipse java-express` utworzy jeszcze konfigurację odpalania (plik *.launch) dla Eclipse'a.

Oba powyższe polecenia (zastosowane jedno po drugim) utworzą projekt z przykładową aplikacją, który będzie można zaimportować do Eclipse'a i uruchomić. Odpalenie w Eclipse aplikacji z zawartego w projekcie pliku *.launch uruchomi tak zwanego shella. Shell to takie GWTowe środowisko uruchamiania ze specjalną przeglądarką. Jest bardzo przydatne przy developmencie, testowaniu i debugowaniu. Jeśli mamy błąd w kodzie klienckim, po uruchomieniu skompilowanego kodu w przeglądarce dostaniemy tylko enigmatyczny komunikat błędu z JavaScriptu. W GWT Shellu dostaniemy pełen stack trace z Javy.



Rys 1. Po lewej stronie GWT Shell. Po prawej „shellowa” przeglądarka.

> *... w momencie budowania aplikacji GWT, kod Javy z pakietu client jest kompilowany do JavaScriptu.*

W tym momencie kod kliencki nie jest jeszcze skompilowany do JavaScriptu, można go skompilować wciskając przycisk "Compile/Browse". Teraz możemy już otworzyć naszą aplikację w dowolnej przeglądarce.

Co my tu mamy - czyli co jest w wygenerowanej aplikacji

W wygenerowanej aplikacji istotny jest pakiet `client`. W pakiecie tym mamy już przykładową klasę. Innych pakietów (pod pakietem `pl.dworld.javaexpress.gwt`) na razie nie mamy. Jeśli nasza aplikacja ma działać całkowicie po stronie klienta (np. prosty kalkulator), i nie potrzebuje komunikować się z serwerem, może tak pozostać. Jeśli jednak chcielibyśmy zrobić aplikację, która będzie wysyłała na serwer jakieś dane lub je pobierała, klasy z którymi klient będzie się komunikował muszą być umieszczone poza pakietem `client`.

Spójrzmy teraz na naszą prostą aplikację. Nie licząc plików konfiguracyjnych serwera, składa się ona z jednej klasy, jednej strony HTML i pliku modułu GWT (*JavaExpress.gwt.xml*). W module określony jest `EntryPoint`, czyli klasa, która ma "obsłużyć" ładowaną stronę html. O wszystkich klasach z pakietu `client` możemy myśleć jak o plikach JavaScript, tzn. są one wykonywane po stronie klienta, przez przeglądarkę. Rzeczywiście, w momencie budowania aplikacji GWT, kod Javy znajdujący się w tym pakiecie jest kompilowany do JavaScriptu. W klasie implementującej `EntryPoint` znajduje się następujący kod:

```
RootPanel.get().add(button);
RootPanel.get().add(label);
```

Te dwie linie dodają przycisk i etykietę do strony HTML. Zazwyczaj aplikacje GWT to jedna strona, początkowo praktycznie pusta, wypełniana potem dynamicznie przez JavaScript. Pewenym problemem w takiej sytuacji stają się zakładki do stron (*bookmarks*), jednak i na to jest sposób, napiszę o nim nieco później.

Komunikacja z serwerem

Mamy zatem gotową aplikację, która działa całkiem po stronie klienta. Wszystko jest w JavaScripcie, serwer jest potrzebny tylko do dostarczenia JavaScriptu przeglądarce użytkownika. Nie jest to jeszcze AJAX, nie mamy tu żadnych odwołań do serwera, których wynik (dane) byłby dynamicznie prezentowany na stronie w momencie jego otrzymania przez klienta, bez potrzeby przeładowania całej strony. Oczywiście, w GWT można się do serwera odwołać. Aby to zrobić trzeba utworzyć kilka interfejsów i klas, zachowując określoną konwencję. Zanim to opiszę, krótkie wyjaśnienie co to są wywołania synchroniczne i asynchroniczne. Wywołania synchroniczne to takie, w których program nie wykonuje się dalej dopóki nie dostanie wyniku wywołania. Każde wywołanie zwykłej metody w Javie jest wywołaniem synchronicznym, np.

```
int a = getValue();
System.out.println("value is here");
```

jest wywołaniem synchronicznym. Druga linia nie wykona się dopóki nie zakończy się wykonywanie pierwszej. W aplikacjach AJAXowych natomiast mamy do czynienia z wywołaniami asynchronicznymi, np.

```
getValue(
    new AsyncCallback() {
      public void onSuccess(int result) {
        a = result;
}});
```

Wywołana jest metoda getValue(...), i zanim jeszcze wynik tego wywołania będzie znany, wykona się linia System.out.println("value is not here"); Wartość wywołania zostanie przypisana do zmiennej a w momencie kiedy będzie znana, może to być po kilku milisekundach, może być po kilku sekundach - nie ma to znaczenia. Nasz program wykonuje się dalej (i np. reaguje na działania użytkownika). Dobrym przykładem takiej AJAXowej aplikacji jest http://maps.google.com. Jak przesuwamy mapę, poszczególne

... mamy wszystko czego nam potrzeba ...

jej fragmenty się ładują, ale zanim jeszcze się załadują, nie jesteśmy "zablokowani", możemy dalej przesuwać mapę. W momencie załadowania danego fragmentu zostanie on wyświetlony.

A jak to się robi w GWT? Zaczniemy od synchronicznego interfejsu serwisu w pakiecie `client`:

```
public interface GwtService
extends RemoteService {
  String getData(String s);
}
```

Warto tu nadmienić jeszcze jedno. Część serwerowa "widzi" klasy części klienckiej, dla serwera cała aplikacja to po prostu klasy Javy. Część kliencka jednak nie widzi serwerowej, i jest to logiczne, po skompilowaniu do JavaScriptu, kod JavaScript, wykonywany w przeglądarce, nie może mieć dostępu do klas Javy. Dlatego jeśli chcecie przekazywać między serwerem a klientem jakieś własne obiekty, muszą one być umieszczone w pakiecie `client`.

Ok, mamy interfejs synchroniczny, zróbmy zatem jego implementację. W zasadzie każdy kod umieszczony poza pakietem client jest traktowany jako kod serwerowy. Na potrzeby tego artykułu utworzymy dla niego pakiet `pl.dworld.javaexpress.gwt.server`. W nim też umieścimy poniższą implementację:

```
public class GwtServiceImpl
    extends RemoteServiceServlet
    implements GwtService {
  public String getData(String s) {
    return getDataFromDb(s);
  }

  private String getDataFromDb(String s) {
    return s.toUpperCase() + " " +
      System.currentTimeMillis();
  }
}
```

Mamy interfejs, mamy implementację, czyli wszystko czego potrzeba po stronie serwerowej. Trzeba ją jeszcze "wystawić" tak, aby była dostępna dla klienta. Nasza implementacja GwtServiceImpl to nic innego jak servlet, odpowiadający na żądania POST. Jak zapewne zauważyłeś, drogi czytelniku, metoda getDataFromDb tak naprawdę nie pobiera niczego z bazy danych, na potrzeby tego artykułu jednak taki „fake" wystarczy. Wystawiamy dopisując do pliku JavaExpress.gwt.xml następującą linię:

```
<servlet path="/service" class="pl.
dworld.javaexpress.gwt.server.
GwtServiceImpl" />
```

To spowoduje wystawienie naszego servletu pod urlem o końcówce "*/service*".

Ale zaraz zaraz, przecież miało być asynchronicznie? I po stronie klienta tak będzie. Wszystkie pozostałe klasy będziemy tworzyć w pakiecie `client`. Potrzebny będzie asynchroniczny interfejs:

```
public interface GwtServiceAsync {
  public void getData(String s,
    AsyncCallback<String> callback);
}
```

Interfejs ten nieznacznie różni się od wersji synchronicznej. Różnice zaznaczone są na czerwono:

- Nazwa taka sama z dodanym słówkiem `Async`
- Każda metoda typu `void`
- Do każdej metody dodany parametr typu `AsyncCallback`
- Typ zwracany z metody interfejsu synchronicznego trafia do callback'u.

Ok, mamy już wszystko czego nam potrzeba. Teraz zobaczmy jak się tego używa. Napiszemy sobie prosty widget do tego celu. Widget to fragment część GUI, coś co można wstawić na stronę GWT. Oczywiście musi być w pakiecie `client`. Na początek bez części odwołującej się do serwera, za chwilę ją dodamy.

> *Dość częstym błędem jest przepisywanie rezultatu wywołania do jakiejś zmiennej klasy*

```
public class GwtServiceUser extends Composite {
  private VerticalPanel m_vPanel = new VerticalPanel();
  private Label m_label = new Label();
  private TextBox m_textBox = new TextBox();

  public GwtServiceUser() {
    m_vPanel.add(m_textBox);
    m_vPanel.add(new Button("Use service"));
    m_vPanel.add(m_label);
    initWidget(m_vPanel);
  }
}
```

Ok, dodajmy teraz ten widget do naszej strony. Nic prostszego, w klasie `JavaExpress` dodajemy na końcu:
`RootPanel.get().add(new GwtServiceUser());`
Dobra, pora naprawdę zapytać o coś serwer. Potrzebny do tego będzie obiekt implementujący interfejs GwtServiceAsync. Przekażemy go do naszego widgetu przez konstruktor, który teraz będzie wyglądał tak:

```
public GwtServiceUser(GwtServiceAsync service) {
  m_vPanel.add(new Button("Use service"));
  m_vPanel.add(m_label);
  initWidget(m_vPanel);
}
```

Skąd go wziąć? Utworzymy go jedną linią kodu, więc teraz ostatnie linie klasy JavaExpress będą wyglądać tak:

```
GwtServiceAsync service = (GwtServiceAsync)
GWT.create(GwtService.class);
RootPanel.get().add(new GwtServiceUser(service));
```

Istotna jest jeszcze adnotacja

`@RemoteServiceRelativePath("/service")`
Możemy ją dodać bezpośrednio przed wywołaniem GWT.create, możemy też (i tak będzie w tym przypadku) przed interfejsem `GwtService`. Będzie on więc wyglądał tak:

```
@RemoteServiceRelativePath("/service")
public interface GwtService
  extends RemoteService {
```

W `GWT.create(...)` zawiera się cała "magia" GWT. To właśnie ta metoda, korzystając z naszych interfejsów (asynchronicznego i synchronicznego) i adnotacji, utworzy nam implementację `GwtServiceAsync`, odwołującą się do servletu znajdującego się pod urlem "*/service*". Pełen kod widgetu korzystającego z tego serwisu wygląda tak jak poniżej.

Wywołanie serwisu RPC opiera się na tzw. callbackach. Interface `AsyncCallback` ma dwie metody: `onFailure` i `onSuccess`. Ich nazwy nie wymagają chyba wyjaśnień. W przypadku powodzenia wywołania RPC, jego wynik trafia do metody `onSuccess` i tam też musimy go obsłużyć. Ważne jest, żeby pamiętać że jest to wywołanie asynchroniczne, i reszta kodu klienckiego może się wykonywać w czasie oczekiwania na rezultat tego wywołania. Dość częstym błędem jest przepisywanie rezultatu wywołania do jakiejś zmiennej klasy i używanie jej w kodzie następującym bezpośrednio po wywołaniu, czyli w momencie gdy jeszcze rezultat wywołania nie jest znany (zmienna nie jest zainicjalizowana).

Tworzenie interfejsu użytkownika

Interfejs użytkownika w GWT opiera sie na tzw.

```
public class GwtServiceUser extends Composite {
  private VerticalPanel m_vPanel = new VerticalPanel();
  private Label m_label = new Label();
  private TextBox m_textBox = new TextBox();
  public GwtServiceUser(final GwtServiceAsync service) {
    m_vPanel.add(m_textBox);
    m_vPanel.add(new Button("Use service", new ClickListener() {
      public void onClick(Widget sender) {
        service.getData(m_textBox.getText(), new AsyncCallback<String>(){
          public void onFailure(Throwable caught) {
            Window.alert("Error!");
          }
          public void onSuccess(String result) {
            m_label.setText(result);
          }
        });
      }
    }));
    m_vPanel.add(m_label);
    initWidget(m_vPanel);
  }
}
```

widgetach. Są to elementy takie jak pola tekstowe, przyciski, napisy itp. Specjalnym rodzajem widgetów są panele. Panele służą do układania widgetów (także innych paneli) na stronie. Robi się to podobnie jak w Swingu. Panele są odpowiednikiem Swingowych layoutów, aczkolwiek zrealizowanym nieco inaczej. I tak jest np. `VerticalPanel`, na którym komponenty są jeden pod drugim, jest `HorizontalPanel`, jest `FlowPanel`, w którym komponenty umieszczane są poziomo, a jak miejsce się kończy, to w kolenym rzędzie pod spodem. Odpowiednikiem `BorderLayout` ze Swinga jest `DockPanel`. Jest także `TabPanel`, pokazujący zawartość w zakładkach. Bardziej zaawansowane przykłady to `Grid` i `FlexTable`.

Tworzenie własnych widgetów

Klasa `GwtServiceUser`, którą właśnie utworzyliśmy, to nic innego jak nasz własny widget. Jak widać rozszerza on klasę `Composite`. Jest to standardowy sposób pisania własnych widgetów dla GWT. Jedynym warunkiem jest wywołanie metody `initWidget()` w konstruktorze. W metodzie tej podajemy `m_vPanel`, czyli panel pionowy z przyciskiem i napisem. W ten sposób wstawiając na stronę `GwtServiceUser` wstawiamy od razu napis i przycisk, odpowiednio już oprogramowany.

Nie wszystko złoto co się świeci

Nie każdy kod Javy da się skompilować do JavaScriptu za pomocą GWT. Kompilowalne są prymitywy i podstawowe klasy i interface'y, takie jak `Collection`, `List`, `Map` (i ich standardowe implementacje), popularne wyjątki i klasy użytkowe (`StringBuffer`, `Math` itp.). Pełna lista jest pod adresem *http://code.google.com/docreader/#t=RefJreEmulation*
Oczywiście można pisać własne klasy kompilowalne do JavaScriptu, wystarczy tylko żeby wszystko czego będziemy w nich używać (pola, zmienne) też było kompilowalne.
Może się to wydawać mało, ale naprawdę da się wykorzystując tylko ten zestaw klas pisać zaawansowane aplikacje w GWT. Poza tym trzeba pamiętać, że ograniczenie to dotyczy tylko kodu części klienckiej, po stronie serwera możemy używać wszystkich dostępnych klas.

> *GWT ma tę zaletę, że opiera się na pewnych standardach*

Zakładki i przyciski "Back" i "Forward"

Pewnym problemem w GWT są zakładki (*bookmarks*). Skoro wszystko dzieje się na jednej stronie, a zmienia się tylko dynamicznie jej zawartość, to cała aplikacja ma jeden adres. Trudno w takiej sytuacji zrobić zakładkę do strony innej niż startowa. Jednak i na to jest sposób. Po URLu w przeglądarce można dodać dowolny tekst po znaku "#" (tzw. *token*). Nie jest to już część adresu. Można więc tworzyć linki i przyciski "przekierowujące" na tę samą stronę, tylko z innym tekstem po znaku "#". W GWT jest klasa `History`, do której możemy podłączyć `HistoryListener`. W ten sposób jesteśmy informowani o wszelkich zmianach URLa (a więc także tokena) w przeglądarce, także tych wywołanych przez wciskanie przycisków `Back` i `Forward`. Za pomocą klasy `History` możemy też programowo odkładać na stos historii nowe adresy, więc możemy to zrobić w dowolnym momencie, nie tylko przekierowując na taką stronę. Możemy na przykład dołożyć element na stos historii w momencie kiedy użytkownik kliknie przycisk na stronie.

Moduły

Aplikacje GWT składają się z modułów. W prostych zastosowaniach (a właściwie często w skomplikowanych też) wystarczy jeden moduł. Moduł GWT to taka GWTowa biblioteka. Można go spakować do jara i używać w GWTowych projektach. Może mieć zarówno kod kliencki jak i serwerowy. Jednak jeśli zawiera kod kliencki, musi zawierać jego źródła. Pamiętajmy, że GWT kompiluje źródła Javy do JavaScriptu. Źródła, nie bytecode. Każdy moduł ma swoją konfigurację. W naszej prostej aplikacji znajduje się ona w pliku *JavaExpress.gwt.xml*. Moduły mogą po sobie dziedziczyć. W naszej aplikacji w pliku *JavaExpress.gwt.xml* znajdziemy linię

```
<inherits name="com.google.gwt.user.User"/>.
```

Oznacza to, że nasz moduł ma też funkcjonalność modułu `com.google.gwt.user.User`. Jest to moduł GWT dołączany standardowo do każdej aplikacji. Z kolei w module `com.google.gwt.user.User` znaleźlibyśmy linię `<inherits name="com.google.gwt.core.Core" />`, więc w naszm module niejawnie mamy także funkcjonalność z modułu Core.

W module określony jest także tzw. `EntryPoint`. Jest to klasa, która obsługuje daną stronę. To tam znajduje się kod wyszukujący na stronie odpowiednie miejsce i modyfikujący w tym miejscu drzewo DOM dokumentu HTML.

GWT ma tę zaletę, że opiera się na pewnych standardach. Np. kod kliencki, ten kompilowany do JavaScriptu, standardowo znajduje się w pakiecie `client`. Możemy w pliku xml modułu skonfigurować inną lokację, ale nie musimy tego robić jeśli standard jest zachowany. Podobnie rzecz się ma z plikami "publicznymi". Są to wszystkie pliki które serwer udostępnia przeglądarce. Możemy tam wrzucić podstrony w HTMLu, obrazki, pliki css itd. Domyślnie powinny się one znajdować w katalogu `public`. Jeśli chcemy trzymać te pliki w innym katalogu - proszę bardzo, wystarczy tylko skonfigurować odpowiednio moduł.

Kolejną istotną rzeczą w module są servlety. Przykład mamy już w naszej aplikacji:

```
<servlet path="/service" class="pl.dworld.javaexpress.gwt.server.GwtServiceImpl" />
```

Oznacza to, że serwis GWT zaimplementowany w klasie `pl.dworld.javaexpress.gwt.server.GwtServiceImpl` będzie wystawiony pod urlem "*/service*". Oczywiście możemy wystawiać wiele serwisów.

W module możemy też zdefiniować pliki CSS i JavaScript, które chcemy dodać do strony.

Istotną rzeczą o której należy pamiętać jest fakt, że servlety skonfigurowane w pliku modułu są brane pod uwagę tylko przy uruchamianiu aplikacji GWT z tzw. shella. To znaczy, że jeśli już

> *Kod wykonywany w metodach testowych w GWTTestCase traktowany jest tak jakby był odpalany po stronie klienta.*

skompilujemy naszą Javę do JavaScriptu, i będziemy chcieli zdeployować na jakimś servlet containerze (np. Tomcacie), to te wszystkie servlety będziemy musieli wpisać do pliku *web.xml*.

Testowanie GWTTestCase

Jeśli piszemy aplikację po stronie klienta serwerowi zostawiając tylko to co rzeczywiście MUSI być po stronie serwera, to dużą część (jeśli nie całość) logiki biznesowej też będziemy mieli po stronie klienta. Warto dla niej napisać testy. Nie ma problemu jeśli są to "zwykłe" klasy, nie korzystające z mechanizmów specyficznych dla GWT (np. `GWT.create()`). Wtedy możemy napisać "normalny" test JUnit. Jeśli jednak nasze klasy korzystają z takich mechanizmów, możemy wykorzystać `GWTTestCase`. Klasa ta dziedziczy po klasie `TestCase` z JUnit 3, zatem nie możemy wykorzystać adnotacji `@Test`, `@Before` itd. Nasze metody testowe muszą być publiczne i typu `void`, a ich nazwy muszą zaczynać się od "test". Ponieważ klasa `GWTTestCase` nadpisuje metody `setUp` oraz `tearDown` i przygotowuje w nich test GWT oraz "sprząta" po nim, nie powinniśmy ich nadpisywać. Twórcy GWT przygotowali dla nas analogiczne metody `gwtSetUp` i `gwtTearDown`. Kod wykonywany w metodach testowych w `GWTTestCase` traktowany jest tak jakby był odpalany po stronie klienta. Tzn. stawiany jest serwer, do którego możemy się odwoływać przez GWT RPC. Działają też takie mechanizmy jak np. `RootPanel.get()`.

Odpowiedzi do Express killers

Przykład pierwszy:

W metodzie log() użyto klasy StringBuffer, która pozwala optymalnie dodawać łańcuchy znakowe. Niestety zastosowano metodę append() tejże klasy, która wywołuje metodę toString() obiektu przekazanego jako referencję. W konsekwencji metoda log() wywołuje w pętli for metodę toString(), która znowu wywołuje metodę log(). Powstaje rekurencja nieskończona. Uruchomienie zakończy się wyjątkiem StackOverFlow().

Przykład drugi:

Kompilator nie pozwoli skompilować tak napisanego kodu, gdyż operator warunkowy będzie niejawnie rzutował wynik swojej operacji na pierwszy wspólny typ, jaki znajdzie w hierarchii dziedziczenia. Wynika to z faktu, że dopiero na etapie uruchomienia wiadomo będzie, co będzie wynikiem operacji, a tym samym, jaki typ będzie zwrócony przez operator. Kompilator jest zbyt leniwy i nie policzy wyniku tej operacji (mimo, że mógłby) – decyzję pozostawi maszynie wirtualnej.

Przykładowo wspólnym przodkiem dla klasy Float i Integer jest klasa Number, dla klasy String i Serializable interface Serializable, ale klasa String i Integer nie mają wspólnego przodka, zatem rzutowanie będzie na klasę Object.

Biorąc pod uwagę, że pierwszym, a zarazem jedynym przodkiem klas MyThrowable i MyException jest klasa Object, ten typ będzie właśnie wynikiem operacji. I tutaj pojawia się problem – klasa Object nie implementuje metody echo(). Zatem zapis można by rozbić na dwie linijki:

```
Object o = true ? new MyThrowable() :
  new MyException();
System.err.println(o.echo());
```

Teraz widać dokładnie, dlaczego kompilator zgłosił błąd. Gdyby klasa MyException rozszerzała klasę MyThrowable, wtedy kompilator nie zgłosiłby błędu, a wynikiem operacji byłaby wspomniana liczba 2.

W zaprezentowanym przykładzie w prosty sposób można wykryć błąd, gdyż zrobił to kompilator. Nadużywanie operatora warunkowego może jednak spowodować, że program nie zachowa się w sposób, który byłby od niego oczekiwany.

Nie ma nic za darmo - biblioteki Open Source

Tomasz Skutnik

Po co nam Open Source?

Stosowanie bibliotek o otwartym kodzie źródłowym (*open source*) podczas programowania w języku Java jest codzienną praktyką. Przeciętny system zbudowany w technologii J2EE zawierać może od kilku do kilkudziesięciu bibliotek pomocniczych. Duża część z nich to biblioteki *open source*, które można zgrać bezpośrednio z Internetu i użyć we własnym projekcie – zwykle bez dodatkowych opłat bądź ograniczeń licencyjnych.

Zalety stosowania bibliotek o otwartym kodzie źródłowym są dość oczywiste:

- brak bądź niewielkie koszty nabycia biblioteki,
- wygoda programowania – szczególnie debugowania,
- możliwość szybkiego naprawiania błędów ujawniających się podczas wdrożenia i eksploatacji systemu,
- lepsze zrozumienie sposobu działania systemu jako całości.

Z doświadczeń firmy e-point SA w tworzeniu i utrzymaniu systemów informatycznych wynika, że główną korzyścią stosowania bibliotek *open source* jest posiadanie kodu źródłowego. Przede wszystkim ze względu na możliwość szybkiej analizy i usuwania błędów wykrywanych podczas eksploatacji systemu. Pozostałe korzyści mają charakter drugorzędny.

Nie zawsze jednak decydując się na użycie w projekcie biblioteki o otwartym kodzie źródłowym zdajemy sobie w pełni sprawę z konsekwencji, jakie taka decyzja pociąga za sobą. Szczególnie jeśli uwzględni się problemy występujące podczas utrzymania i konserwacji systemów informatycznych w dłuższym horyzoncie czasowym.

Tomasz Skutnik
Dyrektor ds. Badań i Rozwoju w e-point SA

Śledzi na bieżąco rozwój wykorzystywanych przez firmę komponentów i narzędzi. Planuje i prowadzi działania rozwojowe w zakresie wewnętrznych komponentów oprogramowania. Zainteresowania zawodowe: języki programowania, metaprogramowanie, generatory kodu, kompilatory.

Przegląd problemów związanych ze stosowaniem bibliotek *open source*

Musimy zdawać sobie sprawę, że decyzja o użyciu konkretnej biblioteki ma najczęściej charakter ostateczny. Zjawisko to, kojarzone zwykle z dostawcami oprogramowania komercyjnego, określa się mianem „vendor lock-in". Oznacza to, że również w przypadku oprogramowania *open source*, potencjalny koszt wymiany dowolnej biblioteki pomocniczej **po wdrożeniu systemu** jest bardzo duży. Dla istotnych komponentów – na przykład warstwy prezentacji czy warstwy dostępu do danych (O/R mapping) – wymiana taka jest zupełnie **nieopłacalna** z biznesowego punktu widzenia.

Jednym z najważniejszych zagadnień utrzymania systemów krytycznych biznesowo jest możliwość szybkiej analizy i usuwania błędów napotkanych we wdrożeniach produkcyjnych. Często zdarza się, że błąd ujawnia się nie w oprogramowaniu wytworzonym przez nas, lecz w bibliotece pomocniczej. Posiadanie w takiej sytuacji jej kodu źródłowego jest niezbędne do szybkiego znalezienia i usunięcia problemu.

Podstawowym grzechem popełnianym przez programistów i kierowników projektów jest nadmierne zaufanie pokładane w **binarnych wersjach** bibliotek o otwartym kodzie źródłowym. W sytuacjach awaryjnych, kiedy potrzeba dostępu do kodu źródłowego staje się paląca, pośpieszne szukanie go w Internecie może zakończyć się niepowodzeniem. Niefrasobliwe uzależnienie projektu od binarnej wersji biblioteki, bez jej minimalnej choćby weryfikacji, powodować może różnorakie problemy.

Najbardziej oczywistym z nich jest brak kodu źródłowego do biblioteki, której używamy. Zwykle po dołączeniu biblioteki do projektu programiści przestają śledzić jej dalszy rozwój. Z czasem, gdy publikowane są kolejne wersje, zarządzający projektami *open source* mają zwyczaj

> *potencjalny koszt wymiany dowolnej biblioteki pomocniczej po wdrożeniu systemu jest bardzo duży.*

kasować wersje archiwalne (zarówno binarne i źródłowe) ze stron WWW projektu. Wtedy zdobycie lub odtworzenie kodu źródłowego do binarnej wersji biblioteki używanej w naszym projekcie (o ile zapobiegliwie nie zgraliśmy go wcześniej) nie jest ani łatwe, ani szybkie.

Jeżeli na stronach projektu nie można znaleźć źródeł, to pierwsze kroki kierujemy do repozytorium kodu źródłowego. Tu jednak czekać na nas mogą niezbyt miłe niespodzianki. Pierwszą z nich jest brak samego repozytorium. Dzieje się tak zazwyczaj, gdy autorzy projektu *open source* decydują się na migrację kodu źródłowego pomiędzy systemami kontroli wersji (np. z CVS do SVN). Jeżeli wersja, której używamy, jest na tyle wiekowa, że była rozwijana jeszcze w starym repozytorium, może się zdarzyć, że nie jest ono już nigdzie dostępne – co w połączeniu z brakiem wersji źródłowej na stronach WWW projektu istotnie utrudnia odnalezienie źródeł. Kolejnym problemem bywa brak w repozytorium tagu odpowiadającego interesującej nas wersji oprogramowania. Najczęściej jest to spowodowane błędem ludzkim, tj. autorzy projektu *open source* czasem zapominają nadać właściwy tag w repozytorium.

Przyjmijmy, że udało nam się jednak zdobyć wersję źródłową projektu. Nie oznacza to końca naszych problemów. Bywa, że zgrany przez nas kod źródłowy po prostu się nie kompiluje. Zwykle błędy kompilacji wynikają z niezgodności wersji JDK lub są to trywialne błędy syntaktyczne (np. brak średnika itp.). Zagadką pozostaje wtedy, kto i w jaki sposób zbudował binarną wersję biblioteki. Na pewno nie powstała ona ze źródeł, które mamy do dyspozycji.

Czasami zdarza się, że budowana przez nas ze źródeł biblioteka kompiluje się prawidłowo, lecz wykonanie dołączonych do niej automatycznych testów kończy się niepowodzeniem. Teoretycznie moglibyśmy zastąpić pierwotnie używaną przez nas wersję biblioteki nową, skompilowaną ze źródeł. Pojawia się jednak w takiej sytuacji pytanie – czy na pewno źródła, z których skompilowaliśmy własną wersję binarną, są prawidłowe? Kto ma rację? Czy to kod źródłowy działa prawidłowo, a testy jednostkowe są nieaktualne, czy może to autorzy projektu opublikowali nową wersję biblioteki z wprowadzonym błędem, bez wcześniejszego sprawdzenia czy dotychczasowe testy kończą się prawidłowo?

Bywa, że projekt *open source* buduje się z wersji źródłowej, testy jednostkowe kończą się powodzeniem, natomiast wygenerowany artefakt binarny nie odpowiada oczekiwanej przez nas wersji. Przykładowo: paczka źródłowa oznaczona jako 2.3 buduje artefakt oznaczony jako 2.4. Bardzo trudno w takiej sytuacji dociec, gdzie leży problem – czy w niewłaściwym nazwaniu paczki źródłowej, czy w niespójności źródeł.

Istnieje też cała klasa mniej krytycznych problemów utrudniających zbudowanie prawidłowej wersji binarnej z posiadanych źródeł, takich jak np. brak domyślnego mechanizmu budowania. O ile w przypadku prostej biblioteki własnoręczne napisanie prostego skryptu budującego nie stanowi problemu, to dla bardziej złożonych projektów – kompilujących klasy warunkowo, bądź modyfikujących kod wynikowy – uzyskanie własnej wersji binarnej może stanowić duże wyzwanie. Dzieje się tak zwykle, gdy autorzy projektu nie mają sformalizowanego, zewnętrznego mechanizmu budowania i publikacji biblioteki, a polegają jedynie na konfiguracji własnego, lokalnego IDE.

Kiedy już zbudujemy wersję binarną ze źródeł, testy jednostkowe kończą się sukcesem oraz, na pierwszy rzut oka, artefakt binarny jest tym, co chcieliśmy uzyskać – wciąż nie musi oznaczać to końca naszych kłopotów.

Czasami, po zamianie oryginalnej wersji binarnej biblioteki na wersję skompilowaną ze źródeł, aplikacja przestaje działać sygnalizując awarię wyjątkami typu `ClassNotFoundException` lub `IncompatibleClassChangeError`. Przyczyną takiego stanu rzeczy okazują się najczęściej obce klasy (tj. pochodzące z innych projektów) dołączone do pierwotnego artefaktu binarnego . I znowu mamy do rozwiązania zagadkę: kto, w jaki sposób i z jakich źródeł zbudował wersję binarną biblioteki, której używamy,

> *Część bibliotek nie określa jawnie, na jakiej licencji można ich używać*

oraz w jaki sposób znalazły się tam klasy z innych bibliotek.

Najbardziej nieprzyjemnym i najtrudniejszym do zdiagnozowania problemem jest sytuacja, w której (pomimo prawidłowej kompilacji, testów i uruchomienia we własnym projekcie) okazuje się, że zbudowana przez nas biblioteka zachowuje się inaczej niż wersja pierwotnie zgrana z Internetu. Przykładowo: dla określonych parametrów wejściowych jakaś funkcja zamiast zwracać wartość `null` rzuca wyjątek. Takie niespodziewane zmiany kontraktu mają tendencję do nakładania się na błędy, które właśnie próbujemy diagnozować, znacząco komplikując proces naprawy.

Oprócz opisanych powyżej problemów, mających charakter techniczny, projekty *open source* mają również szereg innych, mniej uciążliwych wad – głównie w sferze zarządzania. Szczególnie dotyczy to małych, jednoosobowych projektów. Zdarza się bowiem, że gdy projekt spoczywa na barkach jednego człowieka – w sytuacji, gdy ten znajdzie mocno angażującą go pracę, zmieni zainteresowania zawodowe (np. ulubiony język programowania), itp. – rozwój biblioteki zamiera. Nie pojawiają się nowe wersje, błędy nie są naprawiane, a poprawki przesyłane przez użytkowników nie są przeglądane i dołączane do istniejącego kodu źródłowego.

Przed użyciem każdej biblioteki *open source* (zwłaszcza w zastosowaniach komercyjnych!) należy również bezwzględnie sprawdzić licencję, która dołączona jest do projektu. Część bibliotek nie określa jawnie, na jakiej licencji można ich używać – **nagłe** pojawienie się licencji, której warunki wykluczają użycie jej w naszym projekcie – w momencie, gdy mamy już system wdrożony produkcyjnie – może narazić nas na dużą i kosztowną modyfikację istniejącego już i przetestowanego oprogramowania.

Decydując się na wybór biblioteki warto również zwrócić uwagę na jakość dokumentacji, aktywność użytkowników i autorów oprogramowania (listy dyskusyjne, wiki itp.). Co prawda, posiadanie i dobra znajomość kodu źródłowego biblioteki, której chcemy użyć, pozwala nam uniezależnić się częściowo od dokumentacji i autorów. Niemniej dobra dokumentacja techniczna, samouczki i przykłady użycia znacznie ułatwiają pracę z konkretną biblioteką.

Jak sobie radzić w przypadku problemów?

Opis wszystkich powyższych problemów nie byłby kompletny bez próby oszacowania ich skali i zasugerowania możliwych środków zaradczych. W e-point SA przeprowadziliśmy analizę części aktywnych projektów pod kątem użycia zewnętrznych bibliotek o otwartym kodzie źródłowym, ze szczególnym uwzględnieniem problemów opisanych w poprzedniej sekcji. Przegląd obejmował 22 najbardziej aktywne projekty, rozwijane na 43 gałęziach rozwojowych, które korzystały ze 173 różnych bibliotek o otwartym kodzie źródłowym w 253 różnych wersjach.

Wnioski z analizy były pesymistyczne: **33% binarnych wersji bibliotek o otwartym kodzie źródłowym jest obarczona którymś z wcześniej omówionych problemów.**

Co to oznacza? Oznacza to, że w przypadku długoterminowego utrzymania systemów IT, natknięcie się na któryś z powyższych problemów jest jedynie kwestią czasu. Pytanie nie brzmi „czy może nas to spotkać?” lecz „kiedy nas to spotka?”.

A co jeżeli jednak nie możemy znaleźć źródeł wcześniejszej wersji biblioteki na stronach projektu? Gdzie ich szukać? Jak sobie radzić, jeśli pomimo poszukiwań nie uda nam się ich znaleźć?

Pierwszym, często najskuteczniejszym krokiem który można wykonać, jest poszukiwanie źródeł w wyszukiwarkach internetowych, takich jak np. Google, Yahoo. Bywa, że gdzieś w Internecie ktoś posiada archiwalną wersję źródeł, których szukamy, a które nie są już dostępne na stronach projektu. Jeżeli takie poszukiwania zawiodą, można również spróbować sprawdzenia

> *jedynym sposobem na uniknięcie komplikacji jest kompilowanie bibliotek ze źródeł*

w serwisie archive.com. Przechowuje on archiwalne strony WWW i możliwe, że wśród nich znajdziemy także wersję archiwalną biblioteki. Oczywiście, jeżeli jest dostępne publiczne SCM, to jest ono najlepszym miejscem aby zdobyć kod źródłowy. Można zgrać z niego źródła na podstawie oznaczenia tagiem odpowiadającego interesującej nas wersji (pod warunkiem, że taki tag istnieje i jest prawidłowy).

Jeżeli żaden z powyższych sposobów nie doprowadził do znalezienia źródeł, można skorzystać z pomocy dekompilatorów. Nie zawsze uda nam się tak zdekompilować klasę, aby można było w niej naprawić błąd, a następnie ponownie ją skompilować i zaktualizować archiwum biblioteki. Ale można na podstawie zdekompilowanego kodu spróbować przynajmniej znaleźć przyczynę błędu i skonstruować jego tymczasowe obejście.

Jeżeli znaleźliśmy się w sytuacji, w której musimy naprawić błąd w bibliotece, której źródeł nie mamy, możemy również podjąć jedno z dwóch dodatkowych działań rozwiązujących problem: próbować zmigrować oprogramowanie do nowszej wersji biblioteki, której kod źródłowy mamy, bądź całkowicie usunąć daną zależność z projektu. Które z nich wybrać? Zależy to od tego, z jak dużej liczby funkcji danej biblioteki korzysta nasz system oraz potencjalnych niezgodności z nowszymi wersjami biblioteki. Jeżeli korzystamy z niewielu funkcji biblioteki zewnętrznej (w stosunku do całkowitej wielkości naszego systemu), to najkorzystniejsze może być napisanie tych kilku funkcji bibliotecznych na nowo i dołączenie ich do własnego kodu źródłowego.

Jak wobec tego najlepiej zabezpieczyć się przed wszystkimi opisanymi wcześniej kłopotami albo przynajmniej zminimalizować ich niekorzystny wpływ na nasz projekt? Z doświadczeń firmy e-point SA wynika, że jedynym naprawdę skutecznym sposobem na uniknięcie powyższych komplikacji jest **kompilowanie ze źródeł wszystkich używanych w projekcie bibliotek *open source*.**

Oczywiście, nie jest to ani tak proste, ani tak łatwe jak zgranie i bezpośrednie użycie skompilowanej wersji binarnej biblioteki *open source*. Początkowy (wcześniej niemal zerowy) koszt dołączenia biblioteki do naszego rozwiązania nagle zaczyna być zauważalny. Jeśli zamierzamy skorzystać z kilkunastu bibliotek i każdą z nich chcemy skompilować własnoręcznie, to wtedy – w zależności od szybkości budowania poszczególnych bibliotek i napotykanych po drodze przeszkód – czas spędzony przez programistów na dołączaniu bibliotek do projektu zaczyna być zauważalny w budżecie i harmonogramie.

W przypadku większej liczby projektów, czy prowadzenia i utrzymywania wielu projektów na wielu gałęziach rozwojowych równocześnie, bez **mechanizmów automatyzujących** budowanie i centralne zarządzanie lokalnie skompilowanymi bibliotekami nie da się w zasadzie zapewnić odpowiednio szybkiej reakcji na błędy mogące ujawniać się w systemach produkcyjnych. Ich zaprojektowanie, wdrożenie, utrzymanie i udokumentowanie procesu posługiwania się nimi jest jednak zdaniem długotrwałym i dość kosztownym. Dlatego każdy z zespołów tworzących oprogramowanie sam musi sobie odpowiedzieć na pytanie: w jaki sposób ma zarządzać źródłami zależności projektu oraz jaką część budżetu i harmonogramu może na to poświęcić.

Testowanie metod prywatnych

Tomasz Kaczanowski

Wstęp

Nie wdając się w długie rozważania, przyjmijmy na potrzeby tego tekstu, że testy jednostkowe piszemy, aby upewnić się, że stosunkowo mały fragment kodu (najczęściej pojedyncza klasa) nie zawiera błędów. Zadaniem każdej klasy jest służenie innym, tj. udostępnienie im pewnych wartościowych z ich punktu widzenia funkcjonalności, zatem i w testach skupiamy się na tym, czy owe dostępne z zewnątrz funkcjonalności są realizowane poprawnie (tj. zgodnie z dokumentacją i/lub ze zdrowym rozsądkiem).

I tutaj żadnych wątpliwości nie ma - takie funkcjonalności muszą zostać przetestowane. Wątpliwości pojawiają się, gdy developer odczuwa potrzebę przetestowania metod prywatnych, a więc takich, które z zewnątrz nie są bezpośrednio dostępne.

Do rozpatrzenia są tu dwa zagadnienia. Po pierwsze, czy testowanie metod prywatnych jest w ogóle dopuszczalne. Po drugie zaś, jak można to zrobić. W tym artykule postaram się udzielić odpowiedzi na oba te pytania.

Czy wypada?

Poglądy przeciwników testowania metod prywatnych podsumować można zwięzłym stwierdzeniem: "w kodzie obiektowym nietaktem jest zaglądanie obiektom w ich szczegóły implementacyjne". Jest to niewątpliwie słuszna uwaga. Osoby posługujące się tym argumentem, wskazują na słabość designu klasy, który nie pozwala programiście testować jej funkcjonalności jedynie poprzez wywołania metod publicznych.

Znani nam wszystkim pragmatyczni programiści (Dave Thomas & Andy Hunt) w książce Pragmatic Unit Testing we fragmencie poświęconym testom metod prywatnych, zauważają że:

W większości przypadków, klasę powinno się dać przetestować poprzez wywołanie jej metod publicznych. Jeżeli za prywatną lub chronioną metodą kryje się duża funkcjonalność, to być może jest to sygnał ostrzegawczy, który mówi ci, że w rzeczywistości ukryta jest tam odrębna klasa, która chciałaby wydostać się na zewnątrz.

Innymi słowy, uważają oni testowanie metod prywatnych za próbę rozwiązywania niewłaściwego problemu - prawdziwym problemem jest zły design klasy, nie zaś niemożność testowania jej metod prywatnych. Najwyraźniej podobnego zdania są twórcy JUnit, którzy nie zdecydowali się wprowadzić do swojego frameworku ułatwień do testowania metod prywatnych. Prawdopodobnie podejmując taką decyzję mieli na uwadze jedną z reguł XP odnoszącą się do testów:

Jeżeli coś jest zbyt trudne do przetestowania, to prawdopodobnie masz do czynienia z "code smell" - zmień to.

Jako przyczyny pisania testów dla metod prywatnych wskazuje się niekiedy programowanie typu code-first. Takie pisanie kodu sprawia, że przystępując do tworzenia testów programista ma już w głowie szczegóły implementacyjne klasy i zamiast myśleć o jej eksponowanym na zewnątrz zachowaniu, siłą rzeczy skupia swoją uwagę na jej wnętrzu. Jako odtrutkę na takie problemy przywołuje się Test Driven Development (TDD), a w szczególności kodowanie test-first.

Argumentem praktycznym wysuwanym przeciwko testowaniu metod prywatnych jest ich częstsza zmienność niż ma to miejsce w przypadku publicznego API, a co za tym idzie kruchość testów wymagających zmian w odpowiedzi na zmiany wprowadzane w implementacji klasy.

Argumenty drugiej strony są nie mniej ważkie, a ich najkrótsze podsumowanie stanowić może kolejne motto ze świata XP: Testuj

O autorze

Nazywam się Tomek Kaczanowski. Jestem developerem w firmie Software Mind. Interesuje mnie pisanie oprogramowanie wysokiej jakości. W tzw. wolnym czasie dzielę się ze światem moimi przemyśleniami na blogu poświęconym Javie i testom, lub też buduję wieżę z klocków, gotuję kaszki i rysuję motylki.

Testuj wszystko, co może nie działać!

wszystko, co może nie działać. Ponadto zwolennicy testowania metod prywatnych przedkładają kod przetestowany nad kod zgodny z pewnymi, mniej ich zdaniem istotnymi dla jakości kodu, zasadami programowania obiektowego. W dyskusji starają się zapędzić w kozi róg przeciwnika stawiając go przed alternatywami typu: "co jest ważniejsze dla ciebie - testy czy enkapsulacja?". W ten sposób sugerują, że w niektórych przypadkach osiągnięcie obu tych celów jest niemożliwe, a wybór pomiędzy nimi może dać tylko jeden rezultat - kod musi być przetestowany i basta.

Z tym wywodem nie zgadzają się przeciwnicy testowania metod prywatnych (choć wielu z pewnością zgadza się ze stwierdzeniem, że testy są ważniejsze niż cała reszta dobrych praktyk) i jako lekarstwo przywołują wspominane już wcześniej TDD. Testy tworzone przy pomocy TDD sprawdzają całość stworzonego kodu jedynie poprzez wywołanie metod należących do jego API, zaś metody prywatne powstają w wyniku refaktoringu istniejących metod i jako takie, są również przetestowane. Scenariusz takiego developmentu, wygląda następująco:

- piszemy test dla metody A
- piszemy metodę A
- refaktorujemy
- piszemy test dla metody B
- piszemy metodę B
- refaktorujemy - zauważamy wspólną część metoda A i B, korzystając z refaktoringu extract method wyodrębniamy ją w postaci metody C

Skoro prywatna metoda C powstaje w wyniku refaktoringu w pełni przetestowanych metod A i B, to siłą rzeczy jest ona w całości przetestowana testami testującymi te (publiczne) metody, w związku z czym nie ma potrzeby pisać dla niej odrębnych testów.

Na to można usłyszeć, że owszem, że TDD jest bardzo przyjemną techniką, ale pomocną tylko wtedy, gdy tworzymy kod od podstaw. Tymczasem pisanie kodu od zera jest rzadkim luksusem, a proponowanie developerowi zmagającemu się z tysiącami linii legacy code techniki TDD nie jest żadnym wyjściem z sytuacji, i że wówczas testowanie metod prywatnych może być bardzo pomocne. W zetknięciu z wielką, rozdętą, wieloliniową i wszystkorobiącą klasą, developer ma prawo czuć się bezradny, zwłaszcza, że staje przed typowym problemem "jajka i kury":

- klasa ewidentnie powinna zostać zrefaktorowana, ale nie można tego zrobić, bo nie ma testów, które ustrzegą przed wprowadzeniem błędów podczas refaktoringu
- koniecznie trzeba napisać testy dla tej klasy, ale nie jest to możliwe bez refaktoringu

W tym momencie na forach internetowych rozlega się chór głosów pouczających co należy zrobić w takich wypadkach, począwszy od odesłania do literatury (zwłaszcza do "Working Effectively with Legacy Code" Michaela Feathersa) po różne mniej lub bardziej konkretne pomysły, które generalnie sprowadzają się do kuracji w postaci stopniowego pisania testów i refaktoryzacji aż do osiągnięcia zdrowego designu i w pełni przetestowanego kodu. Od tej chwili dyskusja już kompletnie odrywa się od początkowego tematu, pojawiają się wypowiedzi w stylu "w projekcie, w którym pracuję, nie mamy na to czasu", po czym rozmowa dryfuje w rejony technik tworzenia kodu i designu klas, zarządzania projektami IT, przeznaczaniem czasu na testy, narzekaniem na niedobrych project managerów itd.

Najwyraźniej w tym momencie obu stronom sporu o testowanie metod prywatnych kończy się amunicja, możemy więc pokusić się o małe podsumowanie:

- nikt nie twierdzi, że testowanie metod prywatnych to świetny pomysł - ale niektórzy twierdzą, że czasami jest to jedyne wyjście,
- wielu przedkłada kod przetestowany nad kod zgodny z OO - ale są i tacy, którzy akceptują wyłącznie kod, który jest przetestowany i zgodny z OO zarazem,
- tworząc kod od zera mamy luksus takiego pisania go, by mógł zostać w całości przetestowany

> ... choć lepiej tego nie robić, to jednak czasami życie zmusza nas do testowania metod prywatnych

jedynie poprzez wywołania jego metod publicznych - technika TDD może być pomocna w osiągnięciu takiego stanu,
- pracując z legacy code czasami nie mamy wyjścia - skoro przeciwnik (kod) gra nie fair, to i z naszej strony "wszystkie chwyty dozwolone" - włącznie z testowaniem metod prywatnych.

Załóżmy że wypada, to jak mam to zrobić?

Przed chwilą ustaliliśmy, że choć lepiej tego nie robić, to jednak czasami życie zmusza nas do testowania metod prywatnych. Nadszedł więc czas by zająć się techniczną stroną zagadnienia i opisać jak można to zrobić.

W zasadzie istnieją dwie popularne i rozsądne techniki, oraz grono dość dziwnych (i na szczęście niepopularnych) pomysłów.

Metoda I - Zaduma, to znaczy refleksja

Najpopularniejszą chyba techniką jest zmiana modyfikatora dostępu poprzez mechanizm refleksji. Prywatna metoda tylko na czas testu staje się dostępna. Zaletą tej techniki jest brak konieczności dokonywania jakichkolwiek zmian w kodzie testowanej klasy. Fragment kodu testu odpowiedzialnego za "upublicznienie" i wywołanie metody wygląda tak:

```
Method method = targetClass.
  getDeclaredMethod(methodName, argClasses);
method.setAccessible(true);
return method.invoke(targetObject, argObjects);
```

Jak widać zamiast prostego wywołania metody, wyciągamy obiekt klasy Method ją reprezentujący, ustawiamy mu dostęp, a następnie wywołujemy na nim metodę invoke.

Ten sposób dostępu do metod prywatnych jest na tyle popularny, że już dawno temu doczekał się swojej realizacji w postaci specjalnego dodatku do JUnit - JUnit-addons. Jego wadą jest wywoływanie metody poprzez podanie jej nazwy jako Stringu, co powoduje kłopoty w przypadku refaktoringu testowanej klasy.

Metoda II - Domyślny modyfikator dostępu

Drugi pomysł opiera się na wykorzystaniu domyślnego modyfikatora dostępu. Zakładając, że testy leżą w takim samym pakiecie co testowana klasa (co jest dość powszechnie przyjętą praktyką), będą one miały dostęp do metod opatrzonych tym modyfikatorem. Zmieniamy więc modyfikator dostępu metody z prywatnego na domyślny i już możemy bez trudu pisać dla niej testy. Pytanie czy w ten sposób klasa nie odsłania zbytnio swojej implementacji? Sprawa jest dyskusyjna, ale biorąc pod uwagę, że domyślny modyfikator pozwala na dostęp do metody tylko klasom z tego samego pakietu, wiele osób nie uważa takiej zmiany za szkodliwą.

Słabością tej techniki jest konieczność modyfikacji kodu źródłowego - nie skorzystamy z niej, jeżeli nie mamy możliwości jego zmiany.

Pozostałe metody

Dla tych, którzy odczuwają niedosyt rozwiązań, podaje inne pomysły (zaprawdę, pomysłowość ludzka nie zna granic...):
- AspectJ – przy pomocy AspectJ można zrobić z grubsza wszystko, więc oczywiście można go użyć i w tym przypadku. Osobiście uważam to za słaby pomysł, przede wszystkim dlatego, że osób znających AspectJ ze świecą szukać, więc testy napisane z jego wykorzystaniem będą nieczytelne (a więc i nie do przerobienia) w przypadku gdy jedyny specjalista AspectJ w całej firmie pójdzie na L4. Inna sprawa, że użycie AspectJ wydaje mi się tu przysłowiowym pójściem "z armatą na wróble".
- umieszczenie testów w statycznej klasie wewnętrznej testowanej klasy - owszem, taka klasa będzie miała dostęp do prywatnych metod, więc nadaje się do naszego celu. Ponieważ jednak nie chcę by klasa testowa trafiła do wynikowego JARa oraz nie podoba mi się

> *Testowanie metod prywatnych w wikszości przypadków jest świadectwem słabości designu*

pomysł mieszania kodu z testami, więc ten pomysł zdecydowanie odradzam.
- dla prywatnej metody foo() dodajemy metodę foo_FOR_TESTING_ONLY(), która ją wywołuje - i oczywiście umawiamy się z kolegami, że wolno z niej korzystać wyłącznie na potrzeby testów... eee... nie, dziękuję, testy w moim przekonaniu powinny służyć poprawianiu jakości kodu, a tu ewidentnie zaśmiecamy go. Poza tym na godzinę przed deadline nikt nie będzie przejmował się żadnymi ustaleniami i z metody foo_FOR_TESTING_ONLY() ochoczo skorzysta, gdy tylko będzie tego potrzebował ...[po chwili zastanowienia] sam bym tak zrobił byle nie siedzieć po nocy :P.

Podsumowanie

Celem tego artykułu było omówienie zagadnienia testowania metod prywatnych. Przedstawiłem w nim główne argumenty wysuwane za i przeciw, jak również techniczne środki realizacji. Na koniec pozwolę sobie przedstawić własne zdanie na ten temat.

Krótko rzecz ujmując: nie powinieneś tego robić. Testowanie metod prywatnych w większości przypadków jest świadectwem słabości designu i może być wyeliminowane poprzez jego poprawę. Są jednak nieliczne sytuacje – zaliczam do nich np. przejściowe użycie testów metod prywatnych umożliwiające refaktoring szczególnie niechlujnie napisanej klasy – w których testy metod prywatnych uważam za całkowicie uzasadnione. Dlatego powinno się znać podstawowe techniki pisania takich testów – z użyciem refleksji i zmiany modyfikatora dostępu.

Linki

W artykule opierałem się na licznych blogach, wypowiedziach z forów internetowych i list mailingowych, fragmentach książek, tudzież własnych przemyśleniach, doświadczeniach i rozmowach, jakie miałem okazję prowadzić z kolegami po fachu. Poniżej przytaczam niekompletną listę źródeł, z której korzystałem w trakcie pisanie tekstu.

- http://tech.groups.yahoo.com/group/junit/ - lista mailingowa projektu JUnit
- http://junit.sourceforge.net/doc/faq/faq.htm - JUnit FAQ - How do I test private methods?
- http://beust.com/weblog/archives/000303.html - Otaku, Cedric's weblog - Testing private methods? You bet.
- http://agiletips.blogspot.com/2008/11/testing-private-methods-tdd-and-test.html - Paolo Caroli - Testing private methods, TDD and Test-Driven Refactoring
- http://www.jroller.com/CoBraLorD/entry/junit_testing_private_fields_and - Arne Vandamme's weblog - JUnit testing private fields and methods
- http://www.artima.com/suiterunner/private.html - Bill Venners - Testing Private Methods with JUnit and SuiteRunner
- http://www.comp.mq.edu.au/units/comp229/resources/testingprivatemethods.html - COMP229 Object-Oriented Programming Practices
- http://javaworks.wordpress.com/2007/09/06/q-how-do-you-unit-test-private-methods-using-junit/ - Q : How do you Unit Test private methods using JUnit
- http://www.khussein.com/2008/tdd-testing-private-methods-using-aop/ - Khaled Hussein - Test Driven Development: Testing Private Methods Using Aspect Oriented Programming

Java Team

dołącz do nas www.isolution.pl

J2ME: Serializacja obiektów, cz. I

Adam Dec

Wstęp

„Pickling is the process of creating a serialized representation of objects.”
Antonio J Sierra,
„Recursive and non-recursive method to serialize object at J2ME”

Mechanizm serializacji (*Java™Object Serialization Specification*, **[1]**) pozwala na przechowywanie stanów obiektów pomiędzy kolejnymi wywołaniami programu, a także na przesyłanie obiektów przez sieć. Kluczem do zapisywania i odczytywania obiektów jest taka reprezentacja ich stanu, aby możliwa była późniejsza rekonstrukcja. Działa to nawet w sieci, co oznacza, że mechanizm serializacji automatycznie kompensuje różnice między systemami operacyjnymi. Serializacja obiektów umożliwia zaimplementowanie „lekkiej trwałości” (*lightweight persistence*, „Thinking in Java” ch.12). Serializacja została dodana do języka, aby umożliwić zdalne wywoływanie metod (RMI) oraz jest mechanizmem niezbędnym w przypadku komponentów JavaBeans. Stan serializowanego obiektu reprezentowany jest najczęściej w postaci strumienia bajtów. Klasa obiektu, który ma być przechowywany/przesyłany, musi być serializowalna, tzn. musi implementować interfejs java.io.Serializable, jest to interfejs znacznikowy (marker interface, **[2]**), tzn. nie posiada żadnych metod, tylko informuje wirtualną maszynę Java'y, JVM, że obiekt może zostać zapisany. Ważną cechą serializacji jest zapis całej „sieci obiektów”, z która połączony jest pojedynczy obiekt. Dostajemy graf serializowanych obiektów. Procesem serializacji można sterować implementując interfejs java.io.Externalizable, który de facto i tak rozszerza `java.io.Serializable`. Do dyspozycji mamy dwie metody, które wywoływane są automatycznie podczas serializacji/deserializacji:

```
public void readExternal(java.io.ObjectInput in)
throws IOException, ClassNotFoundException;
```

```
public void writeExternal(
java.io.ObjectOutput out)
throws IOException;
```

Interfejs ten jest wtedy użyteczy gdy chcemy mieć pełną kontrolę nad serializacją i deserializacją tworzonych obiektów.

Sun documentation:
„...Externalizable interface are implemented by a class to give the class complete control over the format and contents of the stream for an object and its supertypes...”

Jeżeli nie podoba się komuś perspektywa implementowania interfejsu `java.io.Externalizable` to zawsze można pozostać przy `java.io.Serializable`, „dodając” metody:

```
private void writeObject(
java.io.ObjectOutputStream out)
throws IOException;

private void readObject (
java.io.ObjectInputStream in)
throws IOException, ClassNotFoundException
```

...i tym samym jesteśmy w stanie przejąć kontrolę nad zapisywaniem i odczytywaniem stanu obiektu. W tym wypadku obiekt implementujący interfejs `Serializable` „pytany” jest, przy zastosowaniu mechanizmu refleksji, czy implementuje własną metodę `writeObject()`. Do zapisu stanu obiektu możemy użyć metod `defaultWriteObject()` lub `writeFields()`. Analogiczna rzecz dzieje się w przypadku `readObject()`;

J2ME, czyli Java 2 Platform, Micro Edition, jest edycją Java'y przeznaczoną na pisanie aplikacji pod urządzenia o ograniczonych zasobach oraz małym budżetem pamięciowym (np. telefony komórkowe, PDA). Ze względu na ograniczenia techniczne takich

> *mechanizm serializacji obiektów nie działa w przypadku J2ME*

urządzeń, czyli wolniejsze procesory i mniejsza pamięć, Java ME posiada własny, okrojony w stosunku do JSE, zestaw klas. Specyfikacja J2ME powstała w odpowiedzi na zapotrzebowanie ze strony urządzeń elektronicznych z tzw. "oprogramowaniem wbudowanym". Wiele pakietów z J2SE nie trafiło do J2ME a te, które zostały tu przeniesione ze standardowego zestawu pakietów, zostały mocno „okrojone". Architektura J2ME składa się z konfiguracji, profili oraz pakietów opcjonalnych. Na konfigurację składa się wirtualna maszyna Javy (KVM [3] – Kilobyte Virtual Machine, CVM [4], Monty VM [5], IBM's J9 VM [6], BlackBerry Motions's VM [7]) oraz podstawowy zestaw klas (tzw. klasy core'owe). Każdy producent urządzeń mobilnych implementuje swoją własną maszynę wirtualną. Jej budowa jest zależna od platformy systemowej urządzenia. Klasy core'owe zapewniają podstawową funkcjonalność urządzeniom o podobnych charakterystykach, czyli dostępna pamięć, moc obliczeniowa, rodzaj wyświetlacza, sposób wprowadzania danych czy też możliwość dostępu do sieci. Aktualnie istnieją dwie konfiguracje J2ME:

CLDC (*Connected Limited Device Configuration*)
CDC (*Connected Device Configuration*)

Konfiguracja urządzenia musi zostać powiązana z profilem. Najczęściej spotykane połączenie to CLDC 1.1 + MIDP 2.0 (*Mobile Information Device Profile* JSR 118). Na uwagę na pewno zasługuje NTT DoCoMo Java, zaimplementowana np. w telefonie Sony Ericsson k550i. Telefon ten korzysta z konfiguracji CLDC 1.1 oraz DoJa 2.5 API [8] DoJa jest to alternatywą dla profilu MIDP, oferuje między innymi interfejs do obsługi kamery, wysyłania/odbierania SMS'ów czy też API do obsługi grafiki 2D/3D (*Mascot Capsule Micro3D*). Na bazie wybranej konfiguracji i profilu tworzone są aplikacje zwane MIDletami, wśród których wyróżnić możemy gry lub inne aplikacje wykorzystywane w biznesie. Instalacja aplikacji na urządzeniu mobilnym sprowadza się do skopiowania pliku JAR (*Java Archive*) i JAD (*Java Application Descriptor*) przy pomocy kabla, IRDy, Bluetooth lub technologi GPRS na dane urządzenie. Nasze możliwości co do „pisania" programów niewątpliwe rosną wraz z użyciem pakietów opcjonalnych. Pakiety te rozszerzają platformę J2ME o dodatkowe funkcjonalności (API), z których programiści w ramach danego urządzenia mogą korzystać. Należy mieć na uwadze iż nie wszystkie pakiety są standardowo dostępne na każdym telefonie. Sony Ericsson k550i nie obsługuje np. Location-API oraz SIP-API. Ogólnie pakiety opcjonalne pozwalają np. na dostęp do baz danych (JSR 169), multimediów (JSR 135), protokołu bluetooth (JSR 82), web serwisów (JSR 172) czy nawet grafiki 3D (JSR 184).

Przedstawienie problemu

Niestety mechanizm serializacji obiektów nie działa w przypadku J2ME, żadna z konfiguracji CLDC 1.0 (JSR 30) oraz CLDC 1.1 (JSR 139) nie obsługuje mechanizmu zapisu stanu obiektów czy też refleksji.

Sun documentation:
„Consequently, a JVM supporting CLDC does not support … object serialization..."

Próba kompilacji np. klasy `MyObject` implementującej `java.io.Serializable /Externalizable` lub dodanie do *classpath* artefaktu zawierającego tę klasę w fazie preweryfikacji (badanie klas, które mają zostać wczytane przez maszynę wirtualną zgodną z CLDC, np. KVM) kodu skutkuje wyrzuceniem następującego błędu:

```
Error preverifying class
com.sonic.test.MyObject
java/lang/NoClassDefFoundError:
java/io/Serializable
```

> *możemy stworzyć własny interfejs albo poszukać gotowych rozwiązań*

Problem ten można rozwiązać implementując własny mechanizm serializacji. Przykłady właśnie takich podejść można znaleźć na stronach **[9]** i **[10]**. Mając na uwadze to iż nie możemy skorzystać z `java.io.Serializable/Externalizable` możemy stworzyć własny interfejs albo poszukać gotowych rozwiązań. Ja wybrałem interfejs udostępniany wraz z framework'iem J2ME Polish **[11]** o nazwie de.enough.polish.io.Externalizable. Po ściągnięciu można go znaleźć w artefakcie '`enough-j2me-polish-client.jar`', który znajduje się w katalogu: `${polish.home}/lib/`

Interfejs ten udostępnia dwie metody:

```
public void write(
final    java.io.DataOutputStream    dos)
throws java.io.IOException

public void read(
final    java.io.DataInputStream    dis)
throws java.io.IOException
```

Obiekty DataOutputStream/DataInputStream będą reprezentować strumień danych, do którego będziemy pisać i czytać. Strumień jest pojęciem abstrakcyjnym. Oznacza tor komunikacyjny pomiędzy dowolnym źródłem danych, a ich miejscem przeznaczenia. Strumienie danych umożliwiają posługiwanie się różnymi typami danych. Filtrują połączone z nimi strumienie bajtów, pozwalając na zapis i odczyt prostych typów danych.

```
Sun documentation:
„...A data input stream lets an application read primitive Java data types
from an underlying input stream in a machine-independent way...”
```

W ramach tych klas posługiwać będziemy się takimi metodami jak:

Pisanie do strumienia:

```
void writeX(x var);
```

gdzie X – nazwa typu danych. Na przykład:

```
void writeDouble(double var);
```

Czytanie ze strumienia:

```
x readX();
```

gdzie X – nazwa typu danych. Na przykład:

```
double readDouble();
```

Aby nasz system był uniwersalny będziemy musieli zatroszczyć się o przypadki, gdy w grę wejdą obiekty (np. kolekcje). Używając tych interfejsów, nie mamy metod typu `readObject()/writeObject()`. Na szczęście z pomocą przychodzi nam biblioteka SerME **[12]**, która udostępnia prosty mechanizm serializacji i perzystencji danych do bazy (RMS, `javax.microedition.io.file`) lub też narzędzie z tego samego framework'a o (*J2ME Polish*) nazwie: `de.enough.polish.io.Serializer`.

Użycie tego ostatniego sprowadza się do wywołania dwóch statycznych metod:

```
Serializer.serialize(MyObject, dos);
MyObject myObject = (MyObject) Serializer.deserialize(dis);
```

W tym momencie łatwo wyobrazić sobie naszą przykładową klasę MyObject (listing 1) zawierająca dwa prywatne pola *'name'*, *'myObject2'* oraz metody get/set. Pole *'name'* jest typu `String`, więc nie będzie problemu z serializacją. W przypadku pola *'myObject2'*, jeżeli tylko klasa MyObject2 implementuje interfejs: `de.enough.polish.io.Externalizable` to wystarczy, że użyjemy klasy Serializer :)

Listę typów danych obsługiwanych przez Serializer można zobaczyć pod adresem: *http://www.j2mepolish.org/cms/leftsection/documentation/programming/serialization.html.*

W tym przypadku sprawa była bardzo prosta, mieliśmy tylko dwa pola, więc proces tworzenia implementacji do metod `read()/write()` był szybki i trywialny. Co w przypadku, gdybyśmy mieli 100 takich klas, a każda np. po

> *Chcielibyśmy stworzyć taki system, który sam tworzyłby cały kod*

```
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;

public class MyObject implements de.enough.polish.io.Externalizable {

  private String name = "";

  // Klasa imlementująca de.enough.polish.io.Externalizable
  private MyObject2 myObject2 = null;

  public MyObject() { }

  public MyObject(String name, MyObject2 myObject2) {
    super();
    this.name = name;
    this.myObject2 = myObject2;
  }

  public void read(DataInputStream dis) throws IOException {
    this.name = dis.readUTF();
    this.myObject2 = (MyObject2)de.enough.polish.io.Serializer.deserialize(dis);
  }

  public void write(DataOutputStream dos) throws IOException {
    dos.writeUTF(this.name);
    de.enough.polish.io.Serializer.serialize(this.myObject2, dos);
  }

  public String getName() { return name; }

  public void setName(String name) { this.name = name; }

  public MyObject2 getMyObject2() { return myObject2; }

  public void setMyObject2(MyObject2 myObject2) { this.myObject2 = myObject2; }
}
```

20 pól różnego typu? Pisanie kodu zajęłoby dużo czasu, łatwo o pomyłki i późniejsze błędy typu `java.lang.ClassCastException` występujące już w trakcie działania programu. Musielibyśmy też za każdym razem pamiętać o kolejności zapisywania i odczytywania danych w metodach `read()` i `write()`. Jeżeli w metodzie `read()`, z listingu nr. 1, zamienimy kolejność odczytywania danych ze strumienia, to dostaniemy błąd:

```
java.io.EOFException
at java.io.DataInputStream.readFully(DataInputStream.java:178)
at java.io.DataInputStream.readUTF(DataInputStream.java:565)
at java.io.DataInputStream.readUTF(DataInputStream.java:522)
at MyObject.read(MyObject.java:23)
at MySerializationTest.main(MySerializationTest.java:37)
```

Chcielibyśmy stworzyć taki system, który sam tworzyłby cały kod, czyli pewnego rodzaju automat do generowania źródeł klas, który dosłownie „pisałby”, to czego nam po prostu się nie chce. Ciała metod implementowane byłyby na podstawie pól klas i ich typów. Cały proces

> *Castor jest na tyle elastyczny, że pozwala programiście wpłynąć na proces generacji kodu*

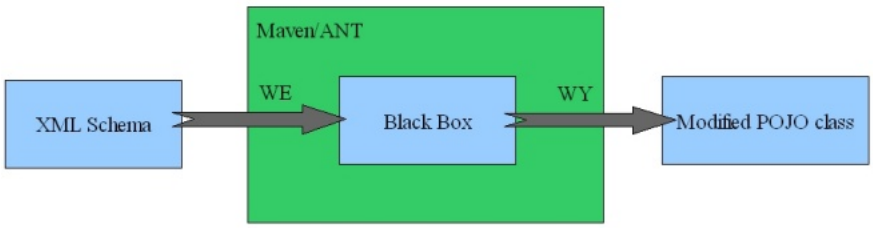
tworzenia odpalany byłby np. za pomocą *Maven*'a lub *ANT*'a. Aby było ciekawiej (trudniej), nasz system będzie musiał spełniać pewne założenia projektowe:

1. Elastyczność systemu ma być zapewniona poprzez integrację z zewnętrznymi interfejsami w postaci plików Schema. Integracja ta ma sprowadzać się stworzenia modelu danych (zmodyfikowanych* klas *POJO*) na podstawie plików *.xsd.
2. Generacja kodu wynikowego ma być w pełni zautomatyzowana, użytkownik systemu powinien jedynie zadbać o dostarczenie odpowiednich plików Schema.
3. Kod wynikowy klas musi się dać skompilować i uruchomić w środowisku J2ME.
4. Z uwagi na tak restrykcyjne środowisko, kod wynikowy powinien mieć jak najmniejszy rozmiar.

* zmodyfikowanych – zawierających gotowe implementacje metod `read()/write()`

Próba rozwiązania

Co chcemy zrobić?



Jest masa narzędzi/framework'ów w sieci, które potrafią stworzyć nam model danych, a co za tym idzie gotowy kod klasy, na podstawie plików Schema (*XML, vocabulary grammar check*). Wygenerowany kod najczęściej przyjmuje formę klasy JavaBean **[13]**, czyli musi posiadać bezargumentowy konstruktor o dostępie publicznym, pola klasy (widoczność: private) dostępne są poprzez publiczne metody (accessor/mutator methods), klasa musi być serializowalna.

Do najbardziej popularnych framewok'ów można by zaliczyć JAXB **[14]**, Apache XMLBeans **[15],** Castor Project **[16]** czy też (na razie w fazie eksperymentalnej) ASF **[17]** (*Adaptive Serializer framework*, SOA-J **[18]** – polecam :p) oraz mniej znane takie jak: Enhydra Zeus **[19],** Arborealis, from Beautiful Code BV **[20]**, JBind **[21]**, Quick **[22]**.

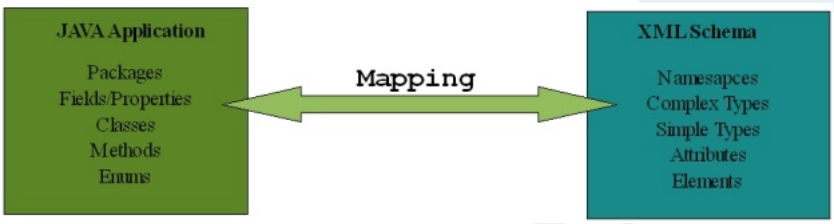
W naszym projekcie użyjemy jednej z bibliotek Castora w wersji 1.2 (czyli nasz Black Box z rysunku nr 1). Biblioteka odpowiedzialna będzie za generowanie kodu źródłowego naszych klas. Klasy tworzone będą na podstawie plików Schema (*W3C XML Schema 1.0 Second Edition, Recommendation*), a biblioteka/tool, który się tym zajmie to `Castor XML Code Generator`.

Castor jest na tyle elastyczny, że pozwala programiście wpłynąć na proces generacji kodu. Proces ten może być kontrolowany poprzez zewnętrzne pliki (np. binding.xml). Plik ten definiuje relację (mapping) pomiędzy Java a XML i jest używany, jeżeli domyślne powiązanie nie spełni w jakiś sposób naszych oczekiwań. Co to oznacza? Domyślny mechanizm nie radzi sobie w przypadku gdy `XML` *Schema* dopuszcza takie same nazwy dla tagu `<element>` i `<complexType>`. Generator kodu wyprodukuje wtedy dwie klasy z taką samą nazwą (*Fully Qualified Class Name*), jednak później to ta ostatnia nadpisze tą pierwszą. Innym przykładem zastosowania jest zmiana domyślnego powiązania pomiędzy typami danych lub próba wprowadzenia ich własnych reguł walidacyjnych i co za tym idzie chęć poinformowania o tym generator kodu (`SourceGenerator`). Według mnie zastosowanie adnotacji (JAXB) w naszym przypadku nie jest zbyt dobrym rozwiązaniem. Za każdym razem, gdy będziemy chcieli zmienić mapping, trzeba będzie zmienić kod źródłowy, przekompilować klasy, a potem zrobić deploy na serwer, jest to dość czasochłonne. Innym minusem jest fakt, że niektórych typów mappingów po prostu nie da się zrobić stosując same adnotacje (np. *multivariate type mappings*). W tym wypadku najlepiej radzi sobie ASF, gdzie każdy mapping danego typu może posiadać wiele strategii jego użycia. Odpo-

> *jedna i ta sama klasa Javy może zostać zmapowana do dwóch różnych plików schema*

wiednia strategia (`<startegy>` element) wybierana jest w zależności od kontekstu, czyli nasze narzędzie samo dostosowuje się do naszych potrzeb. Mając też na uwadze SoC (*Separation of Concerns, SOA – Using Java Web Services, Mark D. Hansen*, chapter 11.3) dobrym nawykiem w trakcie projektowania systemu powinna być separacja warstwy mappingu (*mapping layer*) od implementacji (*implementation layer*). Właśnie ta separacja wpływa na to iż w trakcie działania naszej aplikacji mapping może ulec zmianie, mało tego, mamy też możliwość obsługi wielu mappingów (w JAXB każda klasa ma tylko jeden „zestaw" adnotacji).
Mapping można przedstawić jako parę <J, X>., gdzie J jest to dowolny element Javy, a X dowolny komponent schemy.

Przykład:
`foo:AddressType i foo.Address`



Mapping jest przeprowadzany przez serializer, który zamienia poszczególne instancje klas Javy na instancje XML, które z kolei odpowiadają odpowiednim plikom schema (Java → XML). Funkcja odwrotna wykonywana jest przez deserializer (XML → Java). Mapping jest relacją, ale niekoniecznie funkcją, innymi słowy jedna i ta sama klasa Javy może zostać zmapowana do dwóch różnych plików schema. Mapping zachodzi w przypadku gdy mamy do czynienia zarówno z istniejąca już klasą Java oraz istniejącym plikiem Schema (rys. 2).

Z książki: „SOA Using Java Web Services", Mark D. Hansen, chapter 5.1:
„...type mapping you work with are between existing Java and existing XML schema definitions. At runtime, you will not be working with any machine-generated artifacts."

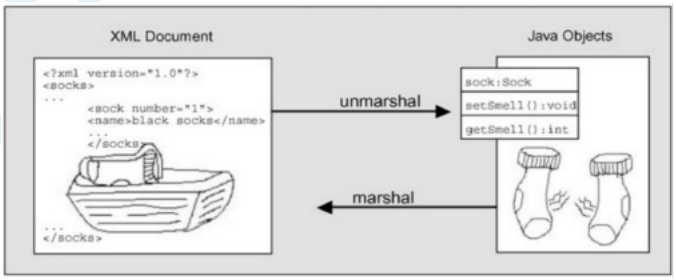
Mapping może być uważany jako swoistego rodzaju most między światem Javy a światem XML'a. Można by pokusić się o stwierdzenie:

„...an XML document is an *instance* of an XML Schema and an Object is an *instance* of a Class...".

Przy pomocy drugiej biblioteki Castor'a o nazwie `Castor XML Data Binding` odbywa się marshaling/unmarshaling. Co to jest `Marshal`?

"A military officer who marshals things, arranges things in methodical order..." :)

...czyli konwersja strumienia (sekwencji bajtów) na obiekt (unmarshall) i obiektu w strumień bajtów (marshall) ale i też forma prezentacji obiektu w pamięci. Przedstawię to na przykładzie, rysunek (rys. 3) pochodzi z artykułu *Sam'a Brodkin'a, JavaWorld.com*, 12/28/01 **[23]**.



Po co nam te operacje? Chyba najlepiej będzie, jak przytoczę tu cytat z tego samego artykułu :)
„The ability to work with unmarshaled XML documents would be great because I could use and maintain regular Java objects much more easily and naturally than I could with a bunch of XML parsing code."

> *Binding pozwala nam w sposób programowy dostać się do zawartości dokumentu XML*

Obie te operacje (marshal/unmarshal) mogą być przeprowadzone poprzez użycie takich klas jak: org.exolab.castor.xml.Marshaller/Unmarshaller. Do dyspozycji mamy też klasę: org.exolab.castor.xml.XMLContext, która odpowiedzialna jest za konfigurację całego środowiska.

Castor może pracować w trzech trybach:
- działanie na podstawie introspekcji (używany jest mechanizm refleksji),
- na podstawie deskryptorów (2 odmiany: runtime i compile time) klas,
- przy użyciu zewnętrznych plików, które określają „mapping" pomiędzy XML'em a klasą JAVA.

Binding pozwala nam w sposób programowy dostać się do zawartości dokumentu XML (przechowywanej w pamięci) bez przechodzenia przez cała strukturę dokumentu (*DOM*, *JDOM*), dlatego też jest to dość wydajne rozwiązanie. Castor narzuca tu pewien wyższy poziom abstrakcji, pozwala na dostęp do danych zawartych w dokumencie przy pomocy modelu obiektowego. Narzędzia, które wykorzystują SAX API lub DOM API bardziej skupiają się nad strukturą dokumentu niż nad danymi, które są w nim zawarte. Ponadto dane traktowane są jako string'i i muszą być cast'owane na odpowiedni typ.

W naszym systemie na razie nie będziemy używać tej biblioteki, więc nie będę się dalej rozpisywał. Zostawię to tylko jako informację. Więcej (odnośnie API) można znaleźć na stronie Castora:
http://www.castor.org/reference/html-single/index.html#XML%20data%20binding

Wracając do biblioteki generującej kod źródłowy, należy wspomnieć, że od wersji `1.0.2` obsługiwane jest generowanie kodu zgodnego z Java 5.0, parametryzacja kolekcji i czy też możliwość używania złożonych enumeratorów. Jednak żadna z tych rzeczy nam się nie przyda, ponieważ nasz kod wynikowy musi poprawnie skompilować się jedynie pod Java 1.1.

Proces tworzenia klas można odpalić na trzy sposoby:
1. Z linii komend:
2. Task w ANT
3. Plugin w Maven

Wszystko, co będzie nam potrzebne, znajduje się w artefakcie `castor-codegen-1.2.jar`. Biblioteka konfigurowana jest przy pomocy pliku `castorbuilder.properties`. Plik ten umieszczony jest w katalogu: `org/exolab/castor/builder` (jest to domyślny plik konfiguracyjny, Castor użyje go, kiedy nie znajdzie na *classpath* żadnego innego. Najważniejszą rzeczą, która zmienimy w tym pliku, będzie właściwość:

```
Org.exolab.castor.builder.jclassPrinterTypes=\
org.exolab.castor.builder.printing.WriterJClassPrinter,\
org.exolab.castor.builder.printing.TemplateJClassPrinter
```

Wartość ta określa, jaka klasa odpowiedzialna będzie za generowanie kodu wynikowego. Domyślnie jest to:
`org.exolab.castor.builder.printing.WriterJClassPrinter`,
czyli prosta klasa, która posłuży nam później za szablon dla naszego projektu :).

W pliku tym możemy znaleźć komentarz:

```
# It can be changed programmatically
# by calling Sourcegenerator.setJClassPrinter(fullyQualifiedClassName)
```

Jednak w samej klasie SourceGenerator nie udało mi się tej metody znaleźć ;/
http://www.castor.org/1.2/javadoc/org/exolab/castor/builder/SourceGenerator.html

Plik ten daje nam też możliwość podania nazwy nadklasy, która będzie rozszerzana przez naszą klasę (`org.exolab.castor.builder.superclass`), wymuszenia generowania klas opakowujących zamiast prymitywów (`org.exolab.castor.builder.primitivetowrapper`), czy też tworzenia kodu zgodnego wyłącznie z Java 1.4 (`org.exolab.castor.builder.javaVersion`).

Podczas generowania kodu implicite korzystamy z klas Apache Velocity **[24]**. Klasa `TemplateJClassPrinter` odpowiedzialna jest

> *Castor narzuca tu pewien wyższy poziom abstrakcji, pozwala na dostęp do danych zawartych w dokumencie XML przy pomocy modelu obiektowego.*

za inicjalizację silnika Velocity. Między innymi ładowany jest plik: `/org/exolab/castor/builder/printing/templates/library.vm`, który np. zawiera makra do tworzenia JavaDocs czy sygnatur metod. Drugi plik: `/org/exolab/castor/builder/printing/templates/main.vm` jest głównym szablonem używanym przy generacji kodu.

Jeżeli nie dodamy odpowiednich bibliotek (`velocity-1.6.1.jar`) do naszego classpath to dostaniemy taki błąd:

```
java.lang.NoClassDefFoundError:
org/apache/velocity/context/Context
```

To jest mój pierwszy artykuł (oby nie ostatni :p) więc z góry przepraszam za wszelkie niedopowiedzenia, literówki i błędy. W następnej części dowiemy się jak praktycznie wykorzystać tę wiedzę, stworzymy dwa projekty. Pierwszy będzie odpowiedzialny za logikę dodawania do generowanych klas metod `read()` i `write()` (faza generate-sources), drugi będzie uruchamiał cały mechanizm, korzystając z *Maven*'a i *ANT*'a. Mam nadzieję, że komuś się to przyda :)

Linki

1. *http://java.sun.com/j2se/1.3/docs/guide/serialization/spec/serialTOC.doc.html*
2. *http://en.wikipedia.org/wiki/Marker_interface_pattern*
3. *http://www.java.com/en/download/faq/what_kvm.xml*
4. *http://developers.sun.com/mobility/configurations/questions/vmdiff/*
5. *http://java.sun.com/products/cldc/wp/ProjectMontyWhitePaper.pdf*
6. *http://www.ibm.com/developerworks/websphere/zones/wireless/weme_eval_runtimes.html?S_CMP=rnav*
7. *http://na.blackberry.com/eng/developers/started*
8. *http://www.doja-developer.net.*
9. *http://j2me.synclastic.com/2006/07/25/revisiting-j2me-object-serialization/*
10. *http://www.w3.org/2006/02/Sierra10022006.pdf*
11. *http://www.j2mepolish.org*
12. *http://www.garret.ru/serme.html*
13. *http://en.wikipedia.org/wiki/JavaBeans*
14. *https://jaxb.dev.java.net/*
15. *http://xmlbeans.apache.org/*
16. *http://www.castor.org/reference/html-single/index.html#xml.code.generator*
17. *http://soa-j.org*
18. *http://www.youtube.com/watch?v=-R-vA_Tc2fM*
19. *http://zeus.enhydra.org/*
20. *http://www.beautifulcode.nl/arborealis/*
21. *http://sourceforge.net/projects/jbind/*
22. *http://sourceforge.net/projects/jxquick/*
23. *http://www.javaworld.com/javaworld/jw-12-2001/jw-1228-jaxb.html?page=1*
24. *http://www.apache.net.pl/velocity/engine/1.6.1/velocity-1.6.1.zip*

Wzorce projektowe: Temporal Object

Michał Bartyzel

Gdy zaczynałem poznawać wzorce projektowe punktem wyjścia dla mnie były książki w stylu GoF, blogi, fora, itd. Znajdowałem tam przede wszystkim diagramy UML, oraz banalne przykłady kodu w stylu: fabryka pizzy, szablon algorytmu, budowniczy okienka. Mój kłopot polegał na tym, że chociaż rozumiałem o czym się do mnie pisze, to nie wiedziałem jak zastosować te koncepcje w moim kodzie. Moje projekty związane były np. ze sklepem internetowym lub systemem ankietowym i nijak to się miało do pizzy, algorytmów czy okienek. Brakowało mi przede wszystkim sensownej implementacji wzorca oraz konkretnych wskazówek jak i gdzie go użyć.

Po jakimś czasie zacząłem analizować źródła programów OpenSource takie jak Spring i apache-commons. Sęk w tym, że aby zrozumieć dobrze te projekty trzeba było wzorce projektowe wcześniej znać, a ja dopiero chciałem się ich nauczyć. Sporo mnie kosztowało rozgryzanie tego tematu.

W tej serii artykułów odpowiem choć na część w/w pytań. Będę: omawiał różne wzorce, podawał przykładowe implementacje i podpowiadał gdzie można ich użyć. Jeśli chodzi o sam sposób użycia czyli wprowadzanie wzorca do projektu i wykrywanie potencjalnych miejsc jego zastosowania w projekcie, to ten temat zostanie poruszony w wątku o refaktoringu (w najbliższej przyszłości).

Obiekty z historią

Wyobraźmy sobie, że należy stworzyć model obiektowy, który posłuży do zrealizowania funkcjonalności sklepu (taaa...wiem, że przykład mocno wyświechtany, lecz jakże użyteczny i skoro skojarzenia takie jak *Hello world!*, *foo bar* i *Team-Member* mocno zapadły w umysły rzesz programistów, więc i ja nie będę odstawał i posłużę się dobrze znanym przykładem rzeczonego sklepu).

Centralną klasą modelu będzie obiekt *Order*. Zakładając, że klient użytkujący sklep może zapisać stan swojego zamówienia, a następnie do niego wrócić, okazuje się, że warto śledzić historię jego...hmmm...niezdecydowania? W przypadku sklepu może być to pomocne np. podczas analizy aktywności klienta, na podstawie której można będzie mu w przyszłości zaproponować nowe produkty i usługi.

Postawiony problem można uogólnić następująco: stan danego obiektu może zmieniać się w czasie i należy zapewnić możliwość śledzenia historii zmian.

W tym miejscu zrób krótką przerwę, weź kartkę i długopis oraz zaproponuj przykładowe rozwiązanie omówionej kwestii.

Już masz? Świetnie, zatem porównaj je z dalszą częścią artykułu.

Temporal Object

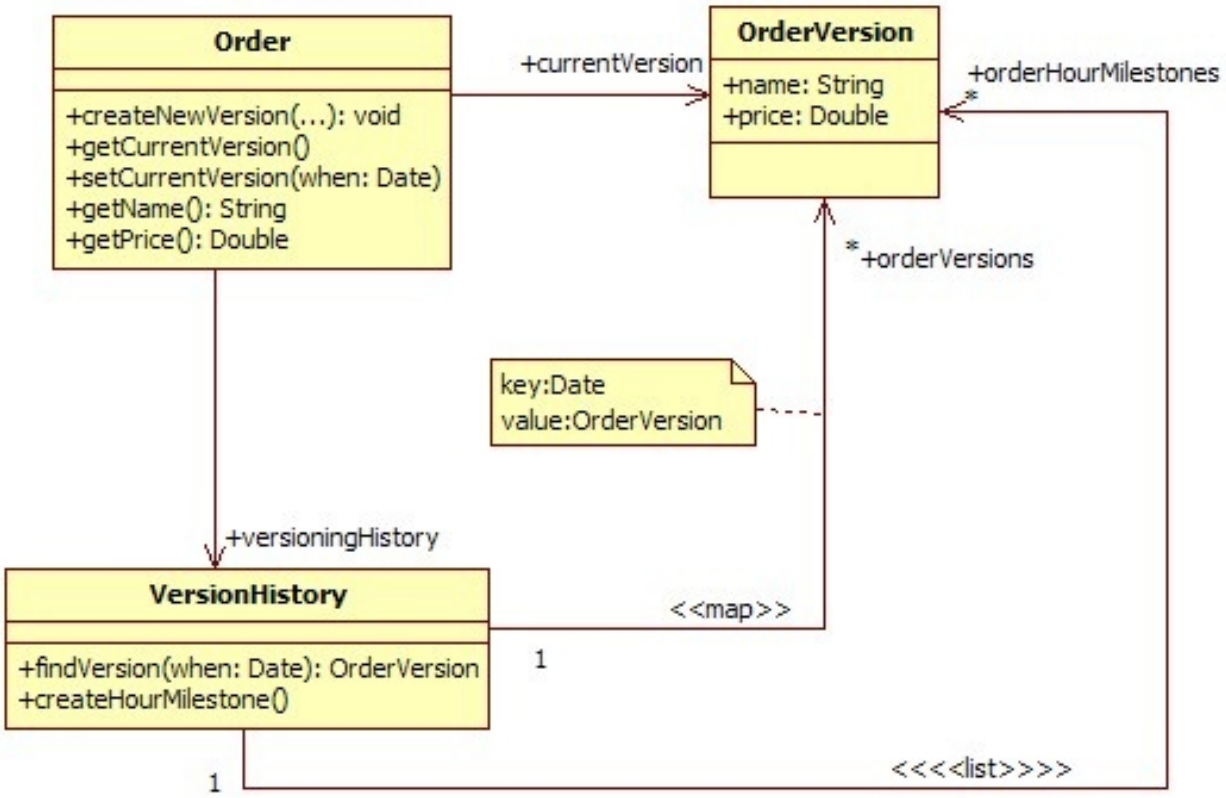
Twoje rozwiązanie jest z pewnością wystarczające, lecz posłuchaj o innym, które jest na tyle często eksploatowane przez programistów, zostało określone mianem wzorca projektowego.

Intencją wzorca *Temporal Object* jest śledzenie zmian w obiekcie i udostępnianie ich na życzenie. Wzorzec ten jest również znany pod nazwami *History on Self* oraz *Version History*.

W przykładzie mamy do czynienia obiektem reprezentującym zamówienie. Sformułujmy wymagania co do funkcjonalności:

- usługa pracuje z obiektem zamówienia Order
- zamówienie można dowolnie zmieniać
- historia zmiana ma być śledzona i udostępniana na życzenie
- dla celów raportowych, oprócz bieżących, należy zapamiętywać godzinowe milestones

Spójrzmy na projekt rozwiązania. Głównym konceptem jest wprowadzenie obiektu OrderVersion, który śledzi zmiany w zamówieni, tzn. dla każdej zmiany tworzony jest nowy obiekt OrderVersion. Sam obiekt klasy Order, z które będą korzystały usługi jest niejako proxy bieżącej wersji zamówienia. Dodatkowo wprowadzona została klasa VersionHistory, której od-



powiedzialnością jest zarządzanie historią zamówienia.

Całe piękno tego rozwiązania polega na tym, że usłudze udostępniony będzie obiekt *Order*, który powinien proksować API *OrderVersion* tyle, że pracuje zawsze na wersji bieżącej. Reszta przetwarzania jest ukryta przed klientem.

Zgodnie z założeniem tej serii artykułów przedstawiam również implementację wzorca.

```
public class Order implements Serializable {
	private Long id;
	private VersionHistory versioningHistory = new VersionHistory();
	private OrderVersion currentVersion;
	public void createNewVersion( String productId, String productName, Double price ) {
		OrderVersion orderVersion = new OrderVersion();
		orderVersion.setCustomId( productId );
		orderVersion.setName( productName );
		orderVersion.setPrice( price );
		versioningHistory.addOrderVersion( orderVersion.getDate(), orderVersion );
		currentVersion = orderVersion;
	}
	public void setCurrentVersionTo( Date when ) {
		currentVersion = versioningHistory.findVersion( when );
	}
	protected OrderVersion getCurrentVersion() {
		return currentVersion;
	}
	public String getCustomId() {
		return getCurrentVersion().getCustomId();
	}
	public void setCustomId( String customId ) {
		getCurrentVersion().setCustomId( customId );
	}
	//delegacje reszty getterów i setterów
}
```

```
public class VersionHistory implements Serializable {
	private Map<Date, OrderVersion> orderVersions
		= new HashMap<Date, OrderVersion>();
	private List<OrderVersion> orderHourMilestones = new ArrayList<OrderVersion>();
	public OrderVersion findVersion( Date date ) {
		return orderVersions.get( date );
	}
	public void addOrderVersion( Date date, OrderVersion version ) {
		orderVersions.put( date , version );
	}
	public void createHourMilestone() {
		//...
	}
}
public class OrderVersion implements Serializable {
	private Long id;
	private String customId;
	private String name;
	private Double price;
	private Date date;
	public OrderVersion() {
		this.date = Utils.getNow();
	}
}
	//gettery i settery
```

A co z persystencją?

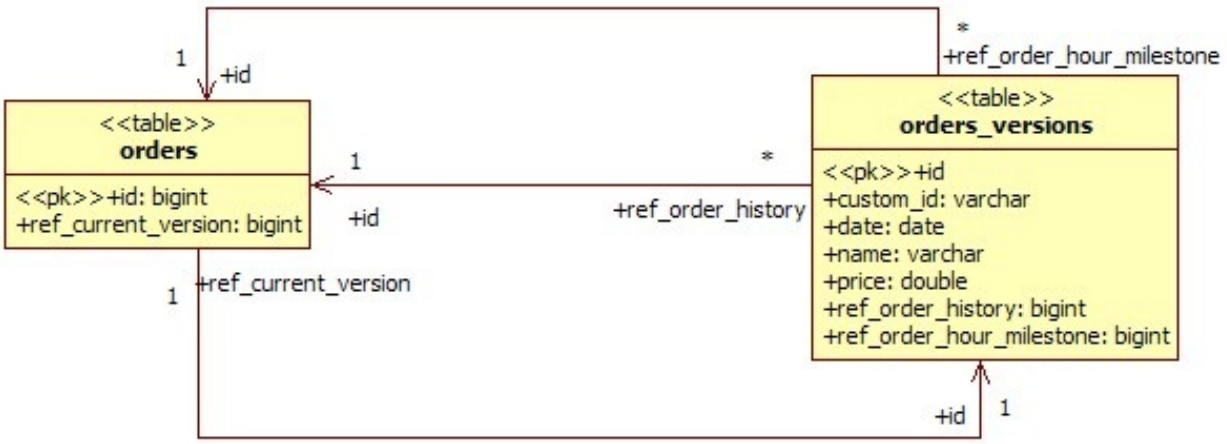
Kolejny problem, o który można potknąć się podczas nauki wzorców to kwestia związana z trwałym przechowywaniem danych. O ile w języku obiektowym można napisać niemal wszystko, również w bazie danych można stworzyć dowolnie złożone rozwiązanie to jednak sklejenie tego razem czasem nastręcza kłopotów. Dlatego, aby opis wzorca był kompletny zajmijmy się teraz trwałym przechowywaniem danych w relacyjnej bazie danych.

Domyślnie, używając dostarczycieli persystencji dla JPA, przyjmowana jest zasada, że jeden obiekt jest mapowany do jednej tabeli w bazie danych. Moim zdaniem przyjęcie takiej arbitralnej zasady prowadzi do bałaganu w bazie danych oraz do jej „niewyważenia". Niewyważenie rozumiem jako sytuację, gdzie poszczególne tabele przechowują nieproporcjonalnie dużą ilość danych, np. jedna tabela ma 2 kolumny oraz 10 wierszy, natomiast inna 20 kolumn i 10000 wierszy. Taka sytuacja w moim mniemaniu daje przesłanki do zastanowienia się, czy ta mała tabela jest potrzebna. Być może można znajdujące się w niej dane umieścić jako dodatkową kolumnę i innej tabeli i w ten sposób uprość zapytania SQL pracujące na bazie. Zaznaczam, że to moje prywatne zdanie.

Wykorzystując mapowania JPA umieścimy strukturę obiektową w dwóch tabelach: *orders* – przechowującej zamówienia oraz *orders_versions* – przechowującą wersje poszczególnych zamówień.

Schemat bazy danych będzie wyglądał jak na poniższym rysunku.

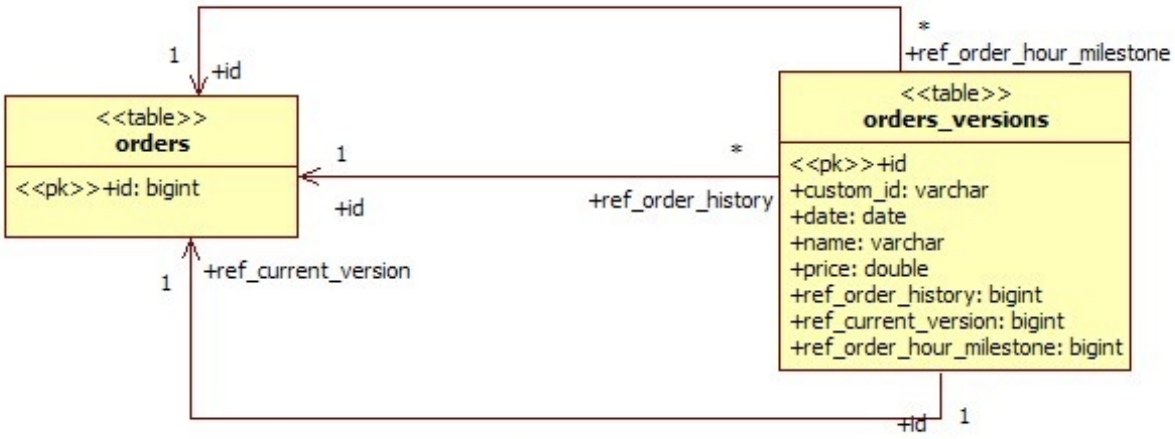
Wiersze z *orders* identyfikują poszczególne zamówienia oraz wskazują na jego bieżącą wersję. Natomiast wiersze z *orders_versions* przechowują dane na temat danej wersji. Dodatkowo każda wersja wskazuje na zamówienie do którego należy oraz, jeśli jest godzinowym milestonem, to wskazuje na właściciela. Na poziomie obiektowym pomiędzy obiektami *Order* oraz *VersionHistory* występuje relacja 1:1, zatem wiersze z *orders* identyfikują również obiekt *VersionHistory*. Z tego względu *orders_versions* posiada dodatkowe wskazanie na *orders* w postaci klucza *ref_order_hour_milestone*, określające, że dana wersja należy do historii wersji danego zamówienia.




```
@Entity @Table( name = "orders" )
@NamedQuery( name="Order.findAll", query="from Order" )
public class Order implements Serializable {
	@Id
	@GeneratedValue( strategy=GenerationType.AUTO )
	private Long id;

	@Embedded
	private VersionHistory versioningHistory = new VersionHistory();

	@OneToOne
	@JoinColumn(name="ref_current_version")
	private OrderVersion currentVersion;
}

@Embeddable
public class VersionHistory implements Serializable {

	@OneToMany(cascade=CascadeType.ALL)
	@MapKey( name="date" )

@JoinColumn( name="ref_order_history" )
	private Map<Date, OrderVersion> orderVersions
		= new HashMap<Date, OrderVersion>();

	@OneToMany
	@JoinColumn( name="ref_order_hour_milestone" )
	private List<OrderVersion> orderHourMilestones = new ArrayList<OrderVersion>();
}

@Entity @Table( name = "orders_versions" )
public class OrderVersion implements Serializable {
	@Id
	@GeneratedValue( strategy=GenerationType.AUTO )
	private Long id;
	@Column( name="custom_id" )
	private String customId;
	private String name;
	private Double price;
	private Date date;
}
```

Odpowiednie mapowania JPA wyglądają jak powyżej.

Jak można zauważyć pomiędzy tabelami *orders* a *orders_versions* występuje powiązanie dwukierunkowe.



Na wstępie tego rozdziału wspominałem o dbaniu o optymalność zapytań. Przeprowadziłem test i zapis jednego zamówienia z trzema wersjami powoduje wykonanie na bazie następujących zapytań SQL:

```
Hibernate: insert into orders (ref_current_version) values (?)
Hibernate: insert into orders_versions (custom_id, date, name, price) values (?, ?, ?, ?)
Hibernate: insert into orders_versions (custom_id, date, name, price) values (?, ?, ?, ?)
Hibernate: insert into orders_versions (custom_id, date, name, price) values (?, ?, ?, ?)
Hibernate: update orders set ref_current_version=? where id=?
Hibernate: update orders_versions set ref_order_hour_milestone=? where id=?
Hibernate: update orders_versions set ref_order_history=? where id=?
Hibernate: update orders_versions set ref_order_history=? where id=?
Hibernate: update orders_versions set ref_order_history=? where id=?
```

Zmieńmy jedna nieco schemat bazy danych przenosząc powiązanie zamówienia z jego bieżącą wersją do tabeli orders_versions. Rysunek powyżej.

Zmiana w mapowaniach jest bardzo niewielka.

```
        //...
public class Order implements Serializable {
        //...
        @OneToOne(mappedBy="parentOrder")
        private OrderVersion currentVersion;
        //...
        public void createNewVersion( String productId, String productName,
                Double price ) {
                //..
                orderVersion.setParentOrder( this );
        }
}

//..
public class OrderVersion implements Serializable {
        //..
        @OneToOne
        @JoinColumn(name="ref_order")
        private Order parentOrder;
        //..
}
```

Michał Bartyzel
Konsultant, trener, współwłaściciel firmy szkoleniowo-doradczej BNS IT. W pracy zawodowej zajmuje się doskonaleniem programistów i zespołów programistycznych, wdrażaniem metodyk pracy oraz rozwijaniem kompetencji pracowników branży IT. Jest współzałożycielem łódzkiego oddziału Java User Group.

> *Używając narzędzi O\RM do trwałego zapisu danych, nie dajmy się zwieść iluzji, że programista może zapomnieć o bazie danych*

Zestaw zapytań tym razem wygenerowany przez Hibernate wygląda następująco:

```
Hibernate: insert into orders values ( )
Hibernate: insert into orders_versions (cu-
stom_id, date, name, ref_order, price) valu-
es (?, ?, ?, ?, ?)
Hibernate: insert into orders_versions (cu-
stom_id, date, name, ref_order, price) valu-
es (?, ?, ?, ?, ?)
Hibernate: insert into orders_versions (cu-
stom_id, date, name, ref_order, price) valu-
es (?, ?, ?, ?, ?)
Hibernate: update orders_versions set ref_or-
der_hour_milestone=? where id=?
Hibernate: update orders_versions set ref_or-
der_history=? where id=?
Hibernate: update orders_versions set ref_or-
der_history=? where id=?
Hibernate: update orders_versions set ref_or-
der_history=? where id=?
```

Zatem mamy o jedno zapytanie mniej. Czy to dużo? Trudno powiedzieć, aczkolwiek na każde tysiąc zapisów zamówienia do bazy danych mamy tysiąc zapytań mniej...

Podsumowując

Wzorca *Temporal Object* można użyć jeśli występuje potrzeba śledzenia i zapamiętywania zmian w modelu obiektowym. Używając narzędzi O\RM do trwałego zapisu danych, nie dajmy się zwieść iluzji, że programista może zapomnieć o bazie danych. Jeśli mamy na uwadze wydajność należy o tym pamiętać, zwłaszcza wtedy, gdy większość narzędzi ukrywa przed programistą złożoność swoich działań.

Programowanie bez słowa redeploy...

Marcin Gadamer

Codzienna praca developera aplikacji webowych wygląda mniej więcej tak: pisanie kodu, kompilacja, deployowanie na serwerze, testowanie, pisanie kodu, kompilacja, deployowanie, testy i tak w kółko.

Podczas gdy pisanie kodu sprawia (a przynajmniej powinno sprawiać) przyjemność, kompilacja trwa zaledwie do kilku minut tak proces wgrywania aplikacji na serwer może trwać w nieskończoność. Czy zatem nie można by pominąć tej fazy lub maksymalnie ją przyspieszyć? Okazuje się, że jest taka możliwość, a z pomocą przychodzi **JavaRebel**.

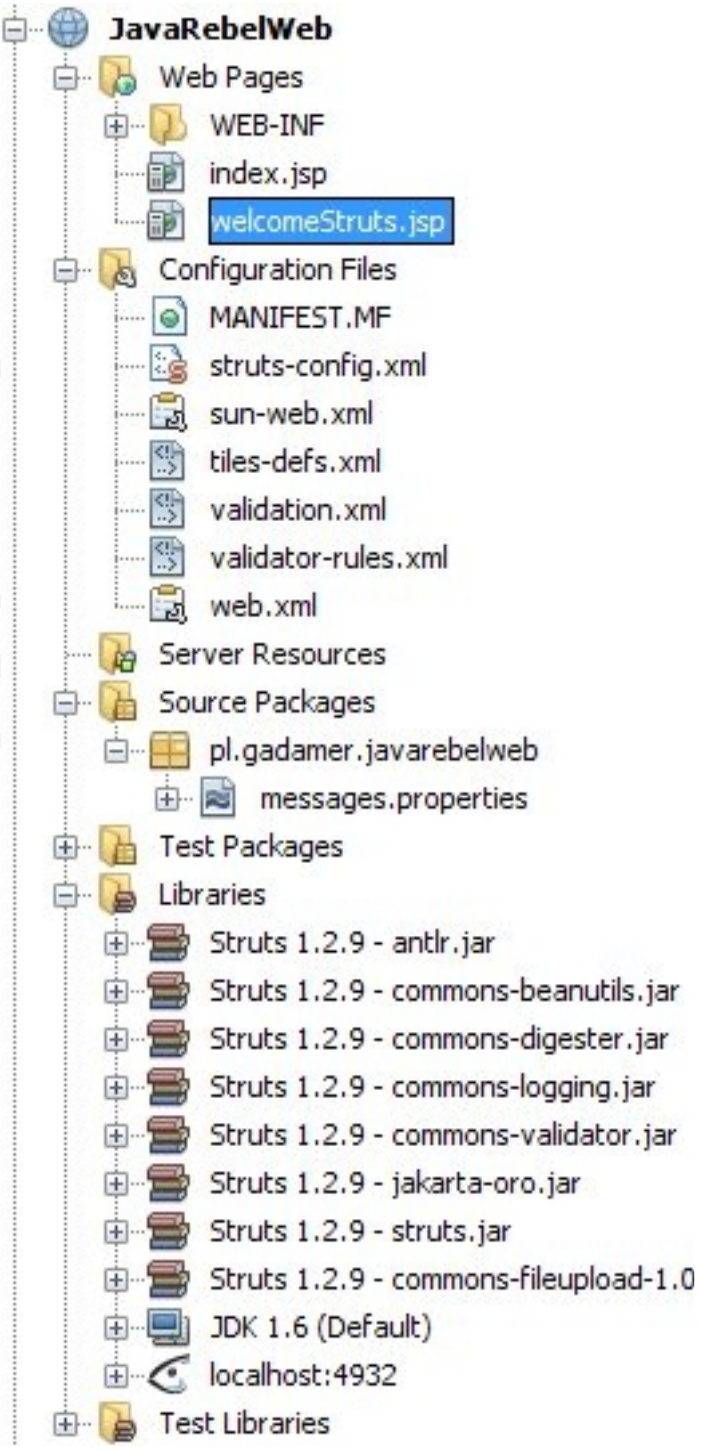
JavaRebel

JavaRebel to plugin dla wirtualnej maszyny Javy (JVM), który umożliwia podmianę w locie zmienionych plików .class aplikacji na serwerze. Nie musimy więc sami wysyłać na serwer całej aplikacji, która niejednokrotnie zajmuje kilka Mb, lecz JavaRebel wysyła automatycznie zmienione przez Nas pojedyncze pliki.

Licencja

Jak zatem zdobyć takie narzędzie? Najprościej będzie pobrać je ze strony producenta *http://www.zeroturnaround.com/javarebel/* Niestety za posiadanie tego plugina trzeba zapłacić. Za roczną licencję przyjdzie nam wydać 99$. Istnieje również możliwość darmowego przetestowania JavaRebel.

Mam dobrą wiadomość dla posiadaczy brązowego pasa na JavaBlackBelt (*http://javablackbelt.com/*). Każdy posiadacz tego pasa ma możliwość otrzymania licencji za darmo. Dodam jeszcze, że licencje rozdawane są również podczas spotkań JUGów – tak zdobyłem swoją.



Rys 1. Struktura katalogów

Przykładowa aplikacja

Pora na zaprezentowanie JavaRebel „w akcji”. Do tego celu stworzę przykładową aplika-

O autorze

Studiuję na AGH Automatykę i Robotykę, a ponadto pracuję jako programista Java. Do niedawna byłem programistą JSE, a od niedawna poznaję JEE.

Zainteresowania: fotografia (cały rok), żeglarstwo (w lecie) oraz narty (w zimie).

Oprócz tego uczę się j. obcych, czytam książki, jem, śpię i próbuję działać tak, żeby doba miała przynajmniej 30h - wtedy mogę zrobić jeszcze więcej.

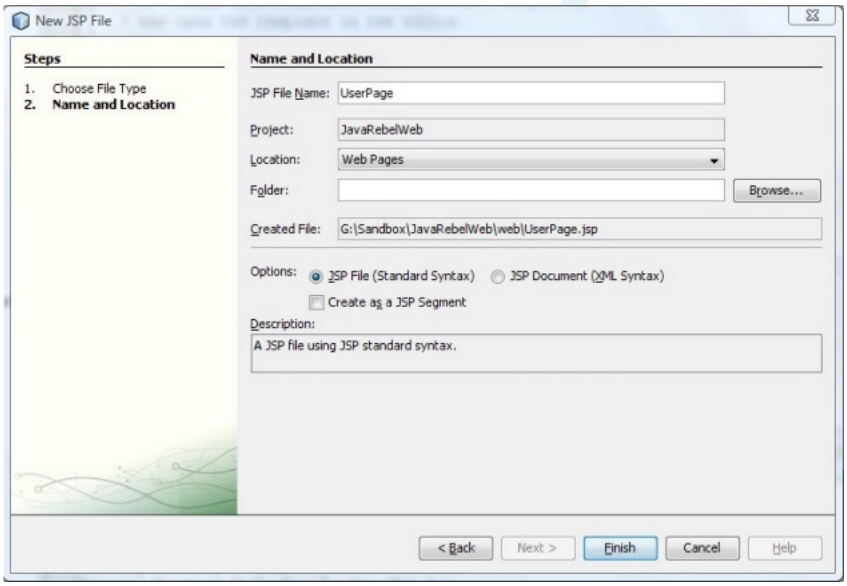
> *JavaRebel automatycznie wysyła do JVM zmienione przez nas pojedyncze pliki.*

cję Web z wykorzystaniem frameworka Struts1, a jako serwer posłuży mi GlassFish v2.

Pierwszą czynnością jest utworzenie projektu. W NetBeans 6.5 wybieram `File->New Project ...` i z listy wybieram `Java Web->Web Application`. Projekt nazwałem *JavaRebelWeb*. Podczas kolejnych kroków wybieram framework *Struts1.2.9* oraz zaznaczamy opcję *„Add Struts TLDs”* i czekam kilka chwil na utworzenie projektu. Powinna zostać stworzona struktura podobna jak na rysunku 1.

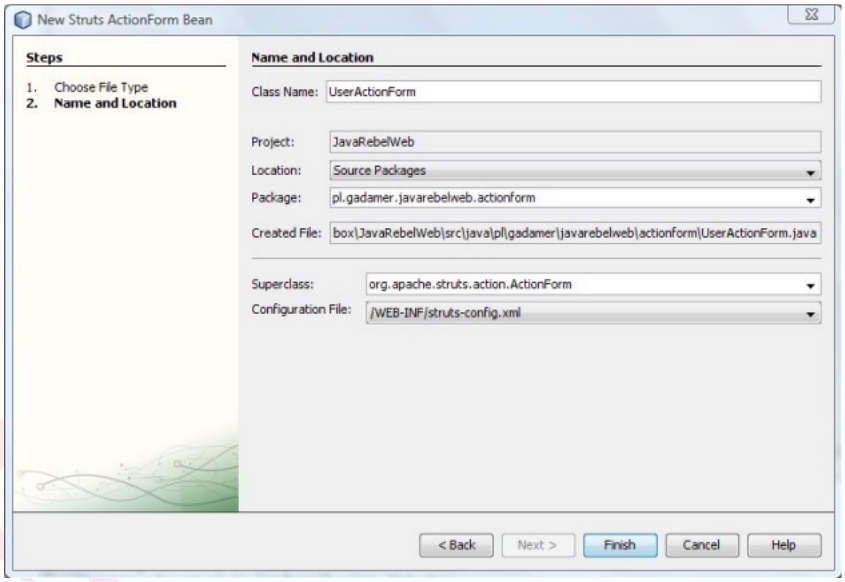
Na początek proponuję stworzyć formularz, który będzie zawierał imię oraz numer identyfikacyjny. Następnie na jego podstawie zaprezentuje działanie JavaRebel.

Zacznijmy więc od utworzenia strony JSP z formularzem. W tym celu naciskamy `Ctrl+N`, a następnie z listy wybieramy `Web->JSP`. W kolejnym kroku wypełniamy pola jak zostało to pokazane na rysunku 2



Rys 2. Tworzenie strony JSP do obsługi klienta – UserPage.jsp

W wygenerowanej stronie dodajemy formularz z dwoma polami oraz przyciskiem do zatwierdzenia. Źródło strony powinno wyglądać podobnie jak w Listingu nr 1.



Rys 3. Konfiguracja UserActionForm.java

Znacznik `<html:errors/>` umieściłem już teraz, choć będziemy go potrzebować dopiero później – przy walidacji pól.

```
<body>
<h1><bean:message key="user.page.title" /></h1>
<html:form action="/UserForm.do" >
<bean:message key="user.page.name" />
<html:text  property="name"/>
<br />
<bean:message key="user.page.number" />
<html:text  property="number" />
<br />
<html:errors/>
<br />
<html:submit>Zaakceptuj! </html:submit>
</html:form>
</body>
```

Listing 1. Kod UserPage.jsp

Stwórzmy teraz klasę, która będzie odzwierciedlała pola we wspomnianym formularzu. W tym celu naciskamy `Ctrl+N`, a następnie z listy wybieramy `Struts->Struts ActionForm Bean`. W kolejnym kroku wypełniamy pola jak zostało to pokazane na rysunku 3.

> *aby serwer wykorzystywał JavaRebel wystarczy dodać dwie linijki*

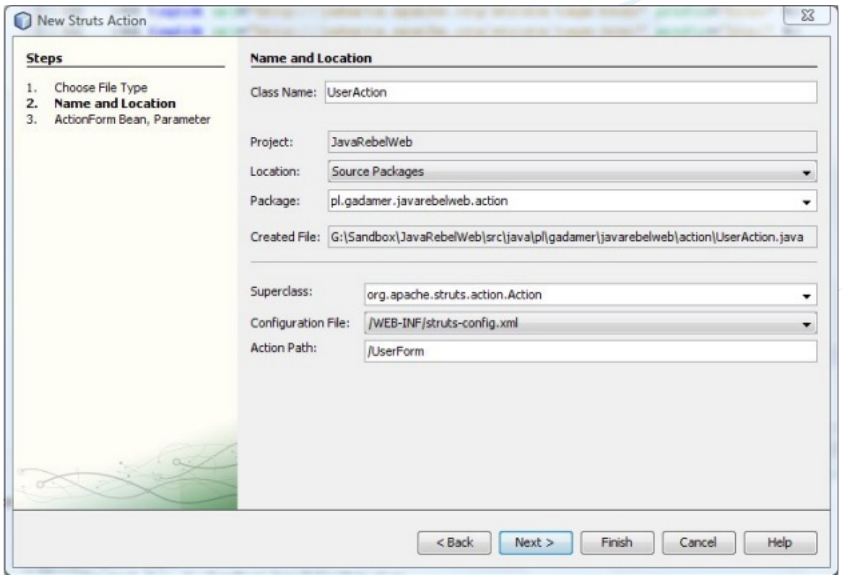
Jak możemy zauważyć w klasie UserActionForm.java zostały utworzone dwa pole prywatne (listing nr 2). Jedno to imię, a drugie to numer – mamy więc to czego potrzebowaliśmy.

```
private String name;
private int number;
```

Listing 2. Pola w UserActionForm.java

Pora na utworzenie akcji, która zostanie wykonana po naciśnięciu przycisku Zaakceptuj! W tym celu naciskamy ponownie Ctrl+N, a następnie z listy wybieramy Struts->Struts Action. W kolejnym kroku wypełniamy pola jak zostało to pokazane na rysunku 4 oraz na rysunku 5.

Ostatnią czynność jaką wykonamy to zmodyfikowanie pliku *web.xml* tak, aby naszą stroną startową była wcześniej utworzona strona *UserPage.jsp* (Listing 3)

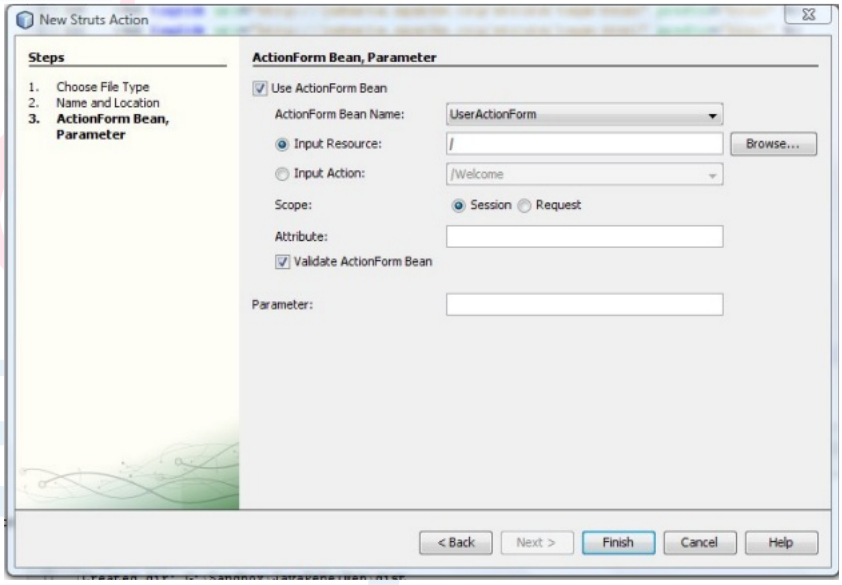


Rys 4 Konfiguracja UserAction.java

Tak stworzyliśmy bardzo prostą aplikację webową. Pora na sprawdzenie czy wszystko działa jak powinno. Naciskamy `Shift + F11` i czekamy aż aplikacja się zbuduje. Po zobaczeniu komunikatu `BUILD SUCCESSFUL` możemy

```
<welcome-file-list>
<welcome-file>UserPage.jsp</welcome-file>
</welcome-file-list>
```

Listing 3. Zmiana w web.xml



Rys 5. Konfiguracja UserAction.java – kolejny krok

sprawdzić, że pliki .class znajdują się w katalogu *JavaRebelWeb\build\web\WEB-INF\classes*. Teraz pozostaje Nam przystąpić do konfiguracji serwera , tak aby potrafił się komunikować z JavaRebel.

Aby serwer wykorzystywał wspomniany plugin wystarczy dodać dwie linijki. Dla serwera GlassFish należy je umieścić w pliku *domain.xml* znajdującym się w katalogu *config* naszej domeny. Owe najważniejsze linijki zostały pokazane w listingu nr 4.

```
<jvm-options>-noverify
-javaagent:C:\Programowanie\Java\JavaRebel\javarebel-1.2.2\javarebel.jar
</jvm-options>
<jvm-options>-Drebel.dirs=G:\Sandbox\JavaRebelWeb\build\web\WEB-INF\classes\
</jvm-options>
```

Listing 4. Konfiguracja JavaRebel

> *W tle omawiany plugin podmienił klasę*

```
#############################################################

ZeroTurnaround JavaRebel 1.2.2 (200812021546)
(c) Copyright Webmedia, Ltd, 2007, 2008. All rights reserved.

This product is licensed to Marcin Gadamer (Personal)

#############################################################

JavaRebel: Directory 'G:\Sandbox\JavaRebelWeb\build\web\WEB-INF\classes' will be monitored for class changes.
```

Listing 5. Konsola z JavaRebel

Pierwsza z nich wskazuje na położenie pliku JAR *javarebel.jar,* natomiast druga informuje jaki katalog ma zostać „śledzony" przez JavaRebel w poszukiwaniu zmienionych klas.

Po zapisaniu zmian uruchamiamy serwer. Jeśli wszystko zostało poprawnie skonfigurowane na ekranie pojawi się komunikat podobny jak w listingu nr 5.

Nie pozostaje nic innego jak standardowo zdeployować utworzoną wcześniej aplikację na serwerze.

Zabawę czas zacząć...

W ulubionej przeglądarce podajemy adres *http://localhost:8080/JavaRebelWeb/,* i po naciśnięciu `ENTER` powinna pokazać się strona *UserPage.jsp*, którą wcześniej przygotowaliśmy, tak ja na rysunku nr 6

Rys 6. Strona główna aplikacji

Zabawę zacznijmy od utworzenia strony JSP, która się wyświetli po podaniu poprawnych danych – nazwijmy ją GoodUser.jsp. Nie będę zamieszczał tutaj szczegółów tworzenia strony. Jedynie na co warto zwrócić uwagę, to na wykonanie w pliku struts-config.xml odpowiedniej modyfikacji jak zostało to przedstawione w listingu nr 6.

```
<action input="/" name="UserActionForm"
path="/UserForm" scope="session"
type="pl.gadamer.javarebelweb.action.UserAction">
<forward name="success" path="/GoodUser.jsp" />
</action>
```

Listing 6. Zmiana w struts-config.xml

Ostatnią rzeczą jest redeploy aplikacji. Dlaczego redeploy? Ponieważ stworzyliśmy nową stronę JSP, a nie była to zmiana w klasie *.java

Gdzie stwierdzamy czy dane podane są poprawnie, czy błędnie? Odpowiedzialna jest za to metoda `validate` w `UserActionForm.java`.

NetBeans wygenerował nam metodę w której sprawdzamy czy pole *name* jest wypełnione. Jeśli nie jest wypełnione to komunikat błędu pojawi się na stronie *UserPage.jsp* w miejscu wcześniejszego dodania znacznika `<html:errors/>`

Spróbujmy zmodyfikować metodę tak aby pole *name* musiało zawierać literę „*a*". Przy-

> „ *została dodana teraz nowa metoda, a nie jak poprzednio zmodyfikowana jedynie istniejąca* ”

kładowy kod jest zamieszczony w listingu nr 7. W pliku z wiadomościami pamiętajmy o dodaniu odpowiedniego klucza.

```
public ActionErrors validate(
    ActionMapping mapping,
    HttpServletRequest request) {
  ActionErrors errors =
    new ActionErrors();
  if (!getName().contains("a")) {
    errors.add("name",
      new ActionMessage(
        "error.name.wrong"));
  }
  return errors;
}
```

Listing 7. Imię zawiera literę "a"

Budujemy aplikację i odświeżamy stronę z formularzem. Na pierwszy rzut oka nic się nie zmieniło. Spójrzmy zatem do konsoli. Tam możemy znaleźć wpis jak w listingu nr 8

```
JavaRebel: Reloading class:
'pl.gadamer.javarebelwar.actionform.UserActionForm'
```

Listing 8. Działanie JavaRebel

Czyli jednak coś się zmieniło! W tle omawiany plugin podmienił klasę UserActionForm.class

Rys 7. Walidacja formularza - dane nieprawidłowe

Sprawdźmy czy, aby na pewno. W polu imię wpisujemy „*Robert*” i naciskamy *Zaakceptuj!* Pojawił się napis *Podano błędne imię!* jak na rysunku nr 7, czyli walidacja zadziałała. Wpiszmy zatem imię z literką „a” - proponuję Marcin Identyfikator jak poprzednio wpisujemy dowolny. Po naciśnięciu przycisku *Zaakceptuj!* zostajemy przeniesieni na stronę `GoodUser.jsp`, więc dane są prawidłowe.

Dodajmy dodatkowo jeszcze walidację dla numeru. Dla przykładu niech pole liczba odpowiada liczbie słów w imieniu. Przykładowy kod został zamieszczony w listing nr 9.

```
public ActionErrors validate(ActionMapping
mapping, HttpServletRequest request) {
  ActionErrors errors = new ActionErrors();
  if (!getName().contains("a")) {
    errors.add("name", new
      ActionMessage("error.name.wrong"));
  }
  errors.add(validateNumber());
  return errors;
}

private ActionErrors validateNumber() {
  ActionErrors errors = new ActionErrors();
  int nameLength = getName().length();
  if (getNumber() != nameLength) {
    errors.add("number", new
      ActionMessage("error.number.wrong"));
  }
  return errors;
}
```

Listing 9. Walidacja numeru

Podobnie jak poprzednio jedynie budujemy aplikację, a następnie odświeżamy stronę główną. W konsoli znów pojawia się informacja o podmianie klasy UserActionForm.class przez JavaRebel. Zauważmy, że została dodana teraz nowa metoda, a nie jak poprzednio zmodyfikowana jedynie istniejąca. (Zakładam, że do poprawnego działania dysponujemy już kompletnym plikiem z wiadomościami i nie musimy w nim dokonywać zmian).

Jak już wspomniałem wcześniej JavaRebel doskonale działa jednak tylko dla zmian w plikach *.class. Podczas zmian w pliku *.jsp lub pliku *.properties musimy wykonać ponowny de-

> *JavaRebel doskonale się sprawdzi, gdy dokonujemy zmian w istniejącym już kodzie*

ploy aplikacji.

Wracając do aplikacji zachowanie jest takie jak chcieliśmy:
- Dla niepoprawnego pola *name* oraz niepoprawnego pola *number* otrzymujemy dwa komunikaty o błędzie,
- Dla jednego błędnego pola otrzymujemy tylko jeden komunikat dla tego pola,
- Gdy we wpisanym imieniu występuje litera „a" oraz gdy została podana odpowiednia liczba następuje przejście do drugiej strony

Możliwości

JavaRebel oferuje znacznie więcej możliwości. Możemy za jej pomocą dodawać i usuwać metody, dodawać i usuwać konstruktory, dokonywać zmian w adnotacjach oraz (od wersji 2.0) dodawać nowe klasy bez konieczności ponownego deployu.

Jest również wsparcie dla frameworków takich jak Spring, Struts2 oraz Tapestry4.

Lista serwerów na jakich możemy korzystać z JavaRebel również jest spora. Obejmuje ona m.in. serwery:
- IBM WebSphere BEA,
- Weblogic 8.x, 9.x, 10.x,
- GlassFish v2,
- Oracle OC4J 9.x, 10.x,
- Tomcat 4.x, 5.x, 6.x,
- JBoss 3.x, 4.x (Java 5 lub 6),
- Jetty 5.x, 6.x (Java 5 lub 6),
- Equinox OSGi (including Eclipse plugins)

Podsumowanie

Uważam, że JavaRebel doskonale się sprawdzi, gdy w aplikacji dokonujemy zmian w istniejącym już kodzie, ponieważ jeśli przychodzi Nam często dodawać nowe pliki to i tak wymaga to ponownego deployu. Doceniony zostanie również w aplikacjach, których deploy trwa kilka, a może nawet kilkanaście minut.

Mam nadzieję, że na tych kilku stronach udało mi się choć w skrócie zaprezentować działanie pluginu JavaRebel. Niewątpliwie jest on godny zainteresowania i przetestowania czy opłaca się wydać blisko 100$ na licencję. Zaoszczędzony tak czas można wykorzystać np. na przeczytanie kolejnego artykułu w JAVA exPress...

Express killers, cz. II

Damian Szczepanik

W drugim odcinku przyjrzyjmy się dwóm przykładom, które kryją pułapki związane z upraszczaniem kodu, skracaniem zapisu oraz chęcią pochopnego przyspieszenia procesu implementacji (i późniejszemu nadrabianiu podczas testowania).

Upraszczanie kodu niesie zwykle jeden cel: jest krótszy w zapisie. Proces ten ma jednak tę wadę, że kod wynikowy jest trudniejszy w analizie – szczególnie przez osoby trzecie. Nasuwa się zatem pytanie, jaki jest cel tak zwanej „optymalizacji" skoro w obu przypadkach bytecode jest identyczny?

Przyjrzyjmy się następującemu przykładowi, w którym programista wykorzystał funkcję log(), aby nie pisać tego samego kodu dwa razy. Sam pomysł godny polecenia, ale implementacja ma jedno ale:

```
public class Instance
{
  private Instance[] inst = { new Instance() {},
    new Instance() {} };

  public static void main(String[] args)
  {
    System.out.println(new Instance());
  }

  private String log()
  {
    StringBuffer sb = new StringBuffer();
    for (Instance i : inst)
      sb.append(i);
    return (sb.toString());
  }

  public String toString()
  {
    return "["+log()+"]";
  }
}
```

Jakie? Co zostanie wydrukowane na wyjście, biorąc pod uwagę fakt, że tablica inst zawiera dwa elementy?

Przyjrzyjmy się kolejnemu kawałkowi kodu. Tym razem programista zastosował operator warunkowy, by wartość jego mogła być obliczona na etapie kompilacji. Czy poniższa klasa się skompiluje, czy tylko skomplikuje? Spójrzmy:

```
public class Echo
{
  static class MyThrowable extends Throwable
  {
    public int echo()
    {
      return 2;
    }
  }


  static class MyException extends Exception
  {
    public int echo()
    {
      return 1;
    }
  }

  public static void main(String[] args)
  {
    System.out.println(
      (true ? new MyThrowable() :
        new MyException()).echo());
  }
}
```

Metoda main() próbuje stworzyć dwa obiekty i wywołać na jednym z nich metodę echo(). Powinien zostać zwrócony wynik w postaci liczby 2. Okazuje się jednak, że pomimo iż obie klasy implementują metodę echo(), kompilator protestuje i nie potrafi skompilować kodu właśnie z powodu tej metody. Gdzie jest źródło problemu?

Odpowiedzi na stronie 15

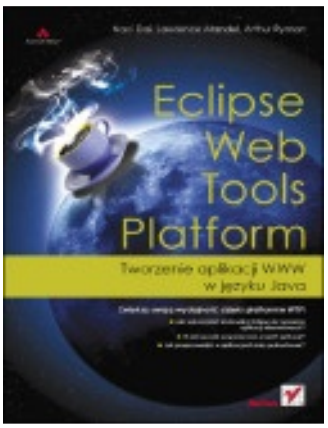
Recenzja: Eclipse Web Tools Platform

Marcin Dąbrowski

Witam.
To jak narazie moja pierwsza recenzja jaką kiedykolwiek napisałem, więc prosze mi wybaczyć pewne niekonsekwencje wynikające z małego doświadczenia. Naturalnie jestem otwarty na wszelkie sugestie i uwagi odnośnie stylu oraz "jakości" wyprodukowanego przezemnie materiału. Zaczynajmy, więc...

W dzisiejszym odcinku postanowiłem opisać książkę na którą natrafiłem parę dni temu, książkę oczekiwaną przez społeczenstwo "fanów" IDE Eclipse od bardzo, bardzo dawna. Książka o której dzisiaj napisze to Eclipse Web Tools Platform.

EWTP autostwa Naci Dai, Lewerence Mandel oraz Arthura Rymana to dość ciekawa pozycja. Zwracając uwagę jedynie na tytuł, czytelnik mógłby sądzić iż ma przed sobą pozycję, która w dokładny sposób opisuje wszelkie "zakamarki" oraz niuanse dotyczące obsługi Elipsa wraz z dołączonym do niego web tool platform. Okazuje się jednak że nic bardziej mylnego, mimo iż książka nie należy do specjalnie cienkich (731 stron) to nie zawiera praktycznie żadnych informacji na temat skrótów klawiczowych czy inszych "udoskonaleń ułatwiających życie". Cytując autorów "Z zasady żadnego z elementów WTP nie omawiamy całkowicie wyczerpująco - tylko w takim zakresie, który jest istotny z praktycznego punktu widzenia". Innymi słowy z książki możemy się dowiedzieć głównie esencji interesującego nas tematu a nie podstaw, które łatwo możemy wyszukać w plikach pomocy, ale czy to źle?



Pozycja składa się z czterech części (w sumie z szesnastu rozdziałów), które w zasadzie można bez większego problemu przeczytać oddzielnie.

Część pierwsza o bardzo "energicznej" nazwie "Zaczynamy" wprowadza nas najprościej w świecie w podstawy samego Eclipse WTP. Znajdziemy tu informacje: Co to jest? Gdzie to jest? oraz Jak się to sciąga i konfuguruje? Dodatkowo w rodziale "Elementarz" mamy ukazany przykład "stawiania" swojego pierwszego servisu. Nie jest to naturalnie nic wyczerpującego, ale z pewnością pozwoli osobie, która nigdy nie miała z czymś takim do czynienia na stworzenie swojego pierwszego "maleństwa". Dodam tylko iż rozdział składa się z czterego oddzielnych podejść, które również możemy traktować indywidalnie.

W drugiej części "Tworzenie aplikacji WWW w Javie" autorzy na chwilę odchodzą od tematyki samego Elipsa i skupiają się bardziej na samej technologii. Znajdziemy tutaj informacje dotyczące organizacji projektu oraz jego struktury, wyjaśnienie poszczególnych warstw tj. prezentacji, logik biznesowych czy trwałości. W tej części książki możemy także dowiedzieć się co kryje się za tajemniczymi skrótami jak SOAP (mydełko), WSDL czy REST. Co więcej zostanie nam nawet przestawiony JUnit oraz jego koledzy, czyli poznamy jak i po co wykonywać testy.

Trzecia cześć została w całości poświęcona środowiskowi WTP. Dowiemy się tu jak dodawać nowe servery, jak stworzyć wtyczkę rozszerzenia WSDL czy DocBook.

W czwartej i zarazem ostatniej części natkniemy się na informacje temat innych narzędzi postałych na szkielecie Eclipsa. Ja osobiście ostatnio część książki potraktowałem jako ciekawostę. Choć nie mogę wykluczyć iż w przyszłości nie przyjdzie mi korzystać z wymienionych tam aplikacji.

Podsumowując, książka nie jest zbiorem skrótów klawiczowych czy opisem perspektyw i widoków. To dość niecodzienna pozycja starająca się dotknąć jak najwięcej i uświadomość czytelnikowi jak olbrzymi świat kryje się za "Eclipse WTP". Jeżeli chciałbyś rozpocząć projektowanie servisow webowych za pomocą tego IDE to naprawdę gorąco polecam. Z pewnością jest warta swej ceny.