Numer 01/2011(9)

Java eXPress

CZASOPISMO DLA DEWELOPERÓW JAVA

Developed by: DEVELOPERS WORLD

Programowanie dla iOS

Transakcje w systemach Java Enterprise, cz.III - Kolejki

Testowanie z użyciem Mockito

Tworzenie komponentów i separacja odpowiedzialności w aplikacjach Flex

Zarządzanie sobą w czasie – pułapki

Dokumentowanie kodu źródłowego java

Designed by:

jakubsosinski.awsome.pl

Patroni:

Adobe

e-point

Lider biznesowych zastosowań technologii Java

Conference.clone();

Gdy pojawiał się pierwszy numer JAVA exPress była jedna, jedyna słuszna konferencja javowa w Polsce. Mowa oczywiście o JDD. Kolejny rok to pojawienie się GeeCONa, którego miałem zaszczyt i przyjemność współorganizować. Dodatkowo rozwinęła skrzydła Javarsovia, która w 2010 roku cieszyła się naliczniejszą grupą uczestników. Javarsovii udało się zorganizować konferencję, która jest zupełnie darmowa. Tak, tak. Niesamowite, ale prawdziwe. Oczywiście kosztem braku prelegentów spoza Polski, ale czy kraj pochodzenia ma znaczenie?

GeeCON poszedł inna drogą. Od początku miał stać się międzynarodową konferencją ze znanymi w świecie Javy prelegentami. 2009 rok był świetny. Wiele dobrych słów płynęło z ust uczestników. Niemniej jednak rok 2010, w przeciwieństwie do Javarsovii, był nieco przespany. Jakość konferencji, wykładów oraz prelegentów wcale nie wzrosła. Wręcz przeciwnie.

A na co możemy liczyć w 2011 roku? Oczywiście stabilne JDD. Bez fajerwerków, ale każdy wie czego można się spodziewać. Dobry catering, ładne hostessy i przeplatanka znakomitych prelegentów zagranicznych ze słabymi wykładami sponsorskimi. Niestety podczas JDD raczej nie ma możliwości wyboru prezentacji i jesteśmy często zmuszeni do odsiedzenia „wykładu".

Javarsovia, jeśli starczy sił organizatorom, to znowu świetna atmosfera i najlepsi polscy prelegenci. Do tego ponoć niezapomniany klimat, którego sam nie mogłem jeszcze doświadczyć. Ale póki co cisza w sieci, a czas ucieka. Specjalna zagrywka organizatorów? Oby.

GeeCON jak zwykle super sale kinowe, wygodne fotele i wyśmienite ekrany. Organizatorzy zakończyli CFP i zaczynają ujawniać coraz więcej nazwisk. I tutaj znowu przeplatanka dobrych prelegentów z nieznanymi nazwiskami i mało ciekawymi prezentacjami. Dobrze, że w przeciwieństwie do JDD mamy wybór. Mam także nadzieję, że kolejni ujawnieni prelegenci podniosą znacznie poprzeczkę. Przecież wśrod 100 zgłoszeń znajdzie się sporo bardzo dobrych wykładów. A dodatkowo przecież organizatorzy zaprosili swoje gwiazdy. Poczekajmy do 1 lutego.

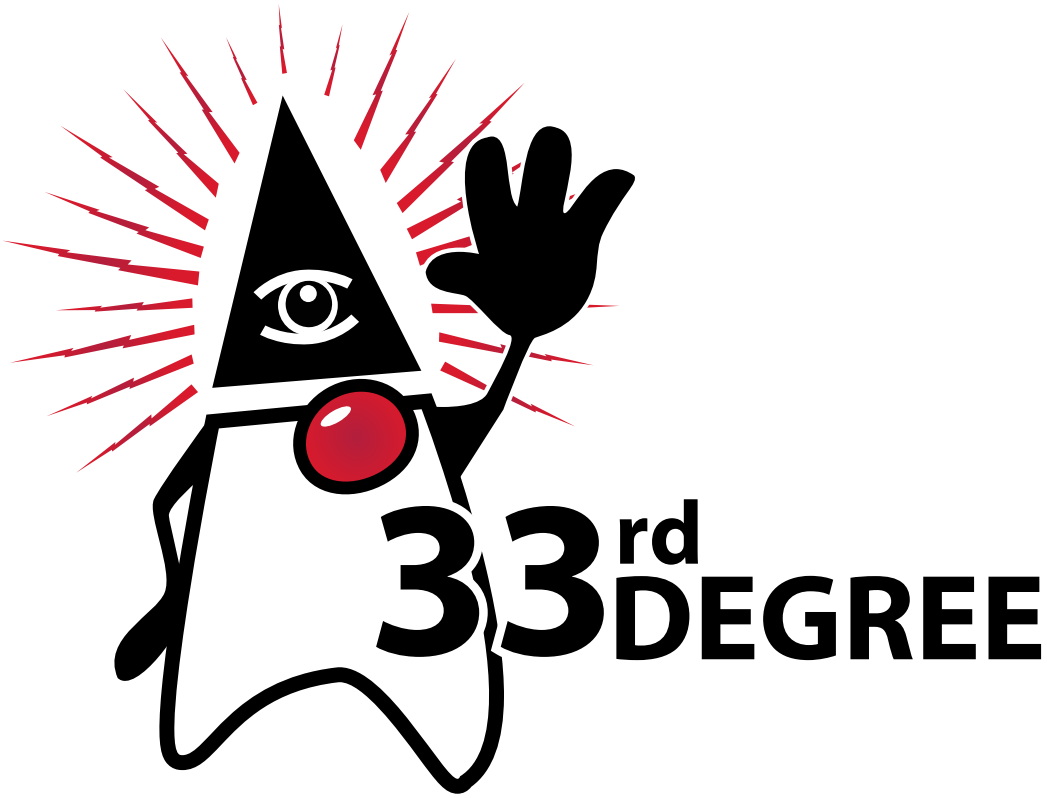
Czy coś jeszcze. No tak. W 2011 roku będzie konferencyjny kolejny klon. O nieco innym charakterze. Bez wygodnych foteli i ekranów kinowych. Za to pełne nastawienie na wzorowe prezentacje i wybornych prelegentów. Coż wiele mówić. Sprawdzie sami na http://33degree.org.

Do zobaczenia na konferencjach,
Grzegorz Duda

Rozkład jazdy

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

Where Java Masters meet



Kraków, 6-8 kwietnia 2011

http://33degree.org

Dokumentowanie kodu źródłowego Java

Łukasz Lechert

Każdy projekt złożonego systemu informatycznego obarczony jest ryzykiem. Ryzyka te dotyczą aspektów związanych z zarządzania wiedzą oraz jej transferem. Można wyobrazić sobie sytuację w której tylko jeden z programistów od początku pracujący w projekcie posiada szczegółową wiedzę na temat działania kluczowych modułów systemu. Pewnego dnia zachęcony intratną propozycją pracy odchodzi do konkurencji. Niektórzy stwierdzą, że nic się nie stało - damy sobie radę. Przewidujący i doświadczeni programiści dostaną gęsiej skórki. Brak świadomości konieczności posiadania wiedzy dostępnej dla całego zespołu prowadzi zazwyczaj do powstawania antywzorców.

Prowadząc projekt na bazie metodyk lekkich, staramy się produkować jak najmniej dokumentów (Kod ponad dokumentacją). W większości przedsięwzięciach minimum jakie musiał obligatoryjnie zagwarantować programista to stworzenie dokumentacji kodu źródłowego, zgodnie z zasadami ustalonymi w projekcie. Nie zawsze jednak się o tym pamięta. Pospieszne, automatyczne generowanie dokumentacji na bazie kodu źródłowego w wielu wypadkach spełnia tylko funkcje wywiązania się z umownych ustaleń z kontrahentem. Dostarczenie „dokumentacji" tylko w formie wydruku zbioru wyprodukowanych pakietów i klas oraz relacji między nimi nie spełnia podstawowych funkcji jakie ona ma pełnić.

Cechy i funkcje dokumentacji

Każdy z nas posiada cele zawodowe, prywatne, które stara się osiągnąć w założonej perspektywie czasowej. Założenia oraz cele charakteryzują się cechami, które te elementy powinny koniecznie posiadać. Każdy cel powinien zostać sformułowany według koncepcji SMART.

Pierwszą cechą jest prostota. Definiując cele interpretacja nie powinna powodować kłopotów w jego zrozumieniu. Sformułowanie musi być jednoznaczne i nie pozostawiające miejsca na luźną interpretację. Cele powinny być małe i nie posiadać większego stopnia skomplikowania.

Drugą cechą jest mierzalność. Element należy sformułować w taki sposób, by można było liczbowo wyrazić stopień lub przynajmniej umożliwić jednoznaczną weryfikację jego realizacji.

Każde zamierzenie oraz działania z założenia powinny być realistyczne i osiągalne. Cel zbyt ambitny lub utopijny podkopuje wiarę w jego osiągnięcie oraz osłabia motywację do jego realizacji.

Ze względu szybkość współczesnego życia cele powinny być istotne. Róbmy tylko to, co jest istotne w projekcie, a zadania o mniejszym priorytecie lub nieważne można przesunąć na dalszy plan.

Idąc dalej oraz biorąc pod uwagę ograniczenie 3 P (wymagań, budżetu i czasu), cel musi być określony w czasie. Podsumowując, powinien mieć dokładnie określony horyzont czasowy w jakim zamierzamy go osiągnąć.

Bazując na opisywanych cechach, można dokonać próby zdefiniowania podstawowych cech, jakimi będzie się charakteryzować dokumentacja kodu źródłowego. Przytoczony przykład jest tylko propozycją i w miarę powstania potrzeby może być rozwijany według uznania Czytelnika.

Cechy dokumentacji kodu źródłowego

- jednoznaczność i spójność (prostota – Kto jest odbiorcą?),
- kompletność (mierzalność – Ile wytworzymy dokumentacji?),
- poprawność (istotność – Jakie informację przekażemy?),
- modyfikowalność (horyzont czasowy - Jak

> Brak świadomości konieczności posiadania wiedzy dostępnej dla całego zespołu prowadzi zazwyczaj do powstawania antywzorców.

długo będziemy ją tworzyć i pielęgnować?).

Kto jest odbiorcą? Odpowiedź na to pytanie w wielu zaspołach jest bardzo prosta. Tylko programiści. Kod źródłowy tworzą programiści dla programistów. Zatem dokumentacja będzie wykorzystywana tylko do rozwoju kodu. Ale czy na pewno są to informacje zarezerwowane tylko dla osób bezpośrednio zaangażowanych w tworzenie kodu?

Patrząc szerzej, z dokumentacji kodu źródłowego mogą korzystać testerzy, architekci lub przedstawiciele zleceniodawcy, weryfikujący jej poprawność podczas odbioru produktu. Dokumentacja powinna być poprawna językowo, jednoznaczna, prosta i spójna. Ma umożliwić klientowi rozwój jego systemu nawet w sytuacji braku wsparcia ze strony pierwotnego zleceniobiorcy. Ile jej wytworzymy? Raczej jak najmniej. „Po co opisywać coś co widać w kodzie". Rzeczywiście, analizując kod źródłowy, można pozyskać informacje jak działa dana metoda oraz jaki algorytm implementuje. Rodzi się pytanie.

Po co analizować kod jak można przeczytać dwa lub trzy zdania, że metoda wykorzystuje algorytm Forda-Bellmana obliczający najkrótszą odległość od wybranego wierzchołka grafu do pozostałych? Jest to informacja bardziej „strawna" niż składnia języka programowania.

Podczas opracowywania dokumentacji należy również zachować umiar. Dokumentowanie prostych „seterów" i „geterów" nie jest dobrym pomysłem oraz może jedynie pełnić funkcję rozrywkową i rozbawiać zespół. Jedną z najistotniejszych cech jest modyfikowalność.

Po pierwsze warto zadbać, żeby dokumentację można było dalej pielęgnować. Żargon i niedokładność byłych kolegów z zespołu nie musi być powszechnie i zawsze rozumiany.

Po drugie wskazane jest żeby trochę pomyśleć o przyszłości i odpowiedzieć sobie na pytanie. Jak będzie pracować się z systemem i rozwijać kod za kilka lat, jeżeli klient zleci jego modyfikację oraz nie będzie informacji jak działają rozwiązania i będą występować braki w dokumentacji kodu źródłowego? W najgorszym wypadku pozostaje debugger, poklepanie po ramieniu oraz zlecenie nowego zadania najbardziej doświadczonym kolegom.

Analogicznie jak dokumentacja dla administratora systemu, końcowego użytkownika, dobra dokumentacja kodu źródłowego pełni określone funkcje w procesie produkcji oprogramowania.

Funkcje dokumentacji

- informacyjna (pomaga szybko wejść w wybrany temat),
- archiwalna (przechowuje informacje o ewolucji systemu),
- standaryzująca (pozwala wypracować własne praktyki odnoszące się do kodu źródłowego).

Nie istnieją globalne standardy i niezmienne reguły tworzenia dokumentacji kodu źródłowego. Chcąc poprawić jakość oprogramowania można pokusić się o opracowanie ogólnych zaleceń dotyczących dokumentowania kodu przez programistów.

> Dokumentowanie poszczególnych metod powinno być obowiązkowe.

Zalecenia

Narzędzie javadoc oraz komentarze zawierające dokumentację stosuje wielu programistów java. Istotnym celem jest stworzenie dla projektu wytycznych określających jej poprawne tworzenie na potrzeby firmy. Wyzwanie nie jest trywialne z uwagi na fakt potrzeby znalezienia punktu równowagi pomiędzy użytecznością, prostotą oraz kompletnością. Przyglądając się projektom realizowanym w języku java, niezbędne jest dokumentowanie klas, interfejsów oraz typów wyliczanych. Przytoczone elementy powinny zostać udokumentowane obligatoryjnie.

Oprócz opisu odpowiedzialnośći klasy, dokumentacja powinna zawierać informacje o autorze, wersji, informacji o wersji systemu, kiedy klasa się pojawiła oraz wiązania do systemu zarządzania kodem źródłowym. W miarę potrzeby można zamieścić linki do innych fragmentów dokumentacji powiązanych z klasą. Przykład minimalnej dokumentacji dla klas zawiera fragment z listingu 1.

```
/**
 * <Opis odpowiedzialności klasy>
 *
 * @author  <Inicjały developera>
 * @since 1.0
 * @see link
*/
```

Kolejnym elementem są metody klas. Dokumentowanie poszczególnych metod powinno być obowiązkowe. Jako wyjątek można przyjąć pominięcie opisów prostych "geterów" i "seterów". W przypadku tak prostych metod uzasadnione jest dokumentowanie "geterów" i "seterów" w przypadku jeśli dotyczą one atrybutów, których znaczenie nie jest intuicyjne. W takiej sytuacji eleganckim rozwiązaniem jest dodanie odnośnika do odpowiedniego atrybutu, gdzie znajduje się szerszy opis. Przykład zastosowania tagu @see prezentuje listing 2.

```
/**
 * @see Person#phoneNumber
 */
public String getPhoneNumber() {
  return phoneNumber;
}
```

Dokumentacja dla pozostałych metod musi zawierać tagi @param, @return i @throws, jeśli generuje wyjątki. Szczególną uwagę oraz obowiązkową dokumentację wymagają metody, które realizują skomplikowane algorytmy oraz interpretacja ich działania nie jest prosta. W takich przypadkach wymagana jest dodatkowa definicja wzorów z których korzysta metoda. Przykład takiej metody prezentuje listing 3.

```
/**
 * Oblicza marze wedlug wzoru ...
 *
 * @param sp cena zakupu
 * @param cp cena sprzedazy
 * @param tax podatek
 *
 * @return marza
 */
public  double  getMargin(double  cp,
double sp, double tax) {
  double margin = 0.0;

  if (vkp != 0) {
    double helper = cp / sp;

    // Margin = 1 - Cost price / Sell-
ing price * (1 + (tax / 100))
    margin = 1 - helper * (1 + (tax /
100.0));
  }

  // Calculation ...

  return margin;
}
```

W przypadku atrybutów wielokrotnie pomija się dołączanie dokumentacji w kodzie źródłowym. Atrybuty są elementami, które zazwyczaj opisują się same, jednak w przypadku specjalistycznych pojęć np. z zakresu rachunkowości lub finansów przedsiębiorstw warto opisać co oznaczają. Podobnie jak w przypadku "geterów" i "seterów", zaleca się dodawanie linków do atrybutów realizujących relację pomiędzy obiektami. Przykładem jest listing 4.

> W sieci dostępnych jest wiele projektów, które umożliwiają kontrolę dokumentacji

```
public class Exam {

    /**
     * Pytania dla egzaminu
     *
     * @see Questions
     */
    Questions questions;

    public Questions getQuestions() {
     return questions;
    }
}
```

Pomocnymi narzędziami mogą się również okazać znaczniki HTML, które można zamieszczać wewnątrz komentarzy. Umożliwiają one formatowanie tekstu, wprowadzanie akapitów, oznaczanie tekstu jako kod źródłowy lub pogrubianie ważnych elementów. Stosowanie tych udogodnień może podnieść jakość i czytelność dokumentacji. Gdy dokumentacja kodu źródłowego jest gotowa do wprowadzenia do repozytorium, można dodatkowo sprawdzić jej poprawność oraz zgodność z globalnymi wytycznymi projektowymi. Na rynku dostępne są narzędzia, wiele z nich to narzędzia darmowe – open source, które umożliwiają sprawne przeprowadzenie kontroli poprawności dokumentacji w kodzie źródłowym java.

Kontrola poprawności dokumentacji kodu źródłowego java

W sieci dostępnych jest wiele projektów, które umożliwiają kontrolę dokumentacji w kodzie źródłowym projektu java. Każdy programista może skorzystać z wtyczki dla środowiska IDE Eclipse. Projekt Eclipse-CS jest darmowym rozwiązaniem, oferowanym na licencji LGPL. Rozwiązanie integruje analizator kodu źródłowego Checkstyle z Eclipsem 3.4 oraz gwarantuje weryfikację i kontrolę dokumentacji po stronie klienta.

Instalacja wtyczki odbywa się w sposób standardowy. Z poziomu menu *Help* wybierając funkcję *Software Updates*, w prosty sposób można zainstalować wtyczkę, służącą do kontroli dokumentacji kodu źródłowego java. Po instalacji należy wskazać projekt do kontroli oraz prawym klawiszem myszy wybrać funkcję Properties → Checkstyle, a następnie włączyć kontrolę za pomocą przełącznika *Checkstyle active for this project*. Projekt zostanie zbudowany ponownie, a w edytorze kodu źródłowego wskazane zostaną istniejące błędy w dokumentacji, które nie są zgodne ze standardowym szablonem Sun Checks ustawionym standardowo po instalacji.

Wtyczka Eclipse-CS udostępnia również tworzenie własnych szablonów, dostosowanych do potrzeb danego projektu. Chcąc stworzyć własny szablon należy przejść do zakładki *Local Check Configurations* oraz następnie za pomocą przycisku *New* stworzyć nową konfigurację szablonu. Tak stworzony element należy dodatkowo skonfigurować, klikając przycisk *Configure*. Z listy dostępnych elementów należy wybrać *Javadoc Comments* oraz następnie element *Method Javadoc* w przypadku wprowadzania reguł dla metod klas.

Dobrym ćwiczeniem jest wymuszenie konieczności wprowadzania w komentarzach dokumentacji tagów @return, @param oraz @throws. W tym przypadku w oknie *Edit module configuration* pozostawiamy odznaczone opcje allowMissingJavaDoc, allowMissingParamTag, allowMissingThrowsTag oraz allowMissingReturnTag. Posiadając tak zdefiniowaną konfigurację jeżeli programista np. zapomni opisać wartości zwracane przez metodę, Eclipse powiadomi go o tym fakcie wyświetlając lupę oraz załączając dodatkowo informację o brakującym elemencie w dokumentacji. Opisywaną sytuację przedstawia prezentowany rysunek 1.

Eclipse-CS bazuje na narzędziu Checkstyle. Narzędzie Checkstyle łatwo integruje się z Jakarta Ant, co stwarza możliwości opracowania mechanizmów kontroli dokumentacji i kodu źródłowego java po stronie serwera budującego kod. Integracja pozwala kontrolować kod globalnie, na poziomie repozytorium zarządzania wersjami rozwijanego systemu oraz raportować znalezione usterki i błędy.

> Wzrost jakości opracowywanej dokumentacji, implikuje łatwiejszą, późniejszą pielęgnację i rozbudowę systemów.

```
/**
 * Simple method
 *
 * @param x collection of numbers
 */
public int foo(int... x) {
    int result = 0;

    for(int z : x) result = result + z;

    return result;
}
}
```

Rysunek 1. Brak taga @return w dokumentacji – informacje wyświetlane przez Eclipse-CS

Podsumowanie

Podsumujmy zalety, wady i nakłady pracy, które wiążą się z wdrożeniem własnych rozwiązań kontroli poprawności dokumentacji w kodzie źródłowym. Niewątpliwie zespół zainteresowany tematem będzie musiał poświęcić czas na przemyślenia i opracowanie optymalnego, globalnego rozwiązania, dostosowanego do własnych potrzeb. Dodatkowo jeśli w projekcie korzysta się z narzędzi weryfikujących kod źródłowy systemu, należy również uwzględnić dostosowanie reguł do opracowanych standardów. Proces raportowania i późniejszych korekt wymaga wskazania odpowiednich priorytetów do popełnionych błędów. Wprowadzenie wytycznych w jednej organizacji, może na początku ograniczać swobodę członków zespołów, rodzić przyzwyczajenia, które nie będą miały zastosowania w projektach innych firm. Od programistów wymagana jest konsekwencja w działaniu i samodyscyplina w celu utrwalenia proponowanego standardu. Niewątpliwą zaletą stosowania własnego rozwiązania w tym obszarze jest wzrost jakości opracowywanej dokumentacji, co implikuje łatwiejszą, późniejszą pielęgnację i rozbudowę systemów.

W sieci

- Javadoc: http://java.sun.com/j2se/javadoc/
- Eclipse-CS: http://eclipse-cs.sourceforge.net/
- Checkstyle: http://checkstyle.sourceforge.net/

Informacje o autorze

Autor jest absolwentem specjalizacji Inżynieria Oprogramowania Politechniki Wrocławskiej oraz Studium Podyplomowego Zarządzanie Projektem na Poznańskim Uniwersytecie Ekonomicznym. Interesuje się systemami wspierającymi procesy logistyczne, zagadnieniami zarządzania projektem oraz oprogramowaniem o otwartym kodzie.

Kontakt: lukasz.lechert@gmail.com

Programowanie Dla iOS

Sebastian Pietrowski

W moim artykule przedstawię, dlaczego warto zainteresować się platformą iOS i w jaki sposób napisać prostą aplikację na urządzenia Apple: iPad, iPhone oraz iPod Touch (w przyszłości mogą to być wszystkie iPod-y). W tym celu przedstawię również podstawy języka Objectiv-C.

Czy warto programować na iOS?

Rozpocznę od podania kilku informacji, które powinny pobudzić naszą wyobraźnię. Co 3 sekundy gdzieś na świecie sprzedawany jest iPad. 15000 aplikacji jest przesyłanych każdego tygodnia. Dotychczas dokonano 5 bilionów pobrań aplikacji. Apple zapłacił 1 bilion dolarów dla developerów, sprzedano 100 milionów urządzeń, które pracują pod kontrolą iOS, funkcjonuje 150 milionów kont z kartą kredytową pozwalającą na szybki zakup Twojej aplikacji. Dane te pochodzą z keynote, który odbył się na WWDC i oczywiście są przestarzałe. . .

Sama platforma dostarcza nam sporo funkcjonalności zarówno sprzętowej: mocny procesor, wydajny i o dużej rozdzielczości wyświetlacz, żyroskop, kamera, jak również odpowiednie API do przetwarzania dzwięku, wideo, lokalizacji, żyroskopu itp.

Rozwój systemu iOS jest bardzo dynamiczny. W czerwcu ogłoszono zmianę nazwy iPhone OS na iOS i nadanie numeru 4, gdzie równocześnie wprowadzono wiele zmian, a od ogłoszenia wersji 4 powstały dwa uaktualnienia wersje 4.1 oraz 4.2.

Werjsa 4.1 przyniosła najciekawszą nową funkcjonalność, którą jest Game Center. Game Center pozwala bardzo prosto i szybko stworzyć grę pozwalającą w łatwy sposób grać z drugą osobą. Otrzymujemy przy tym szereg standardowych funkcjonalności jak obsługa połączenia, rankingi, znajomi.

Zostało również wydane kolejne wydanie 4.2 i po raz kolejny dostarczono nam developerom kilku ciekawostek, najważniejsza to wprowadzenie pełnej kompatybilności iPhone/iTouch oraz iPad, co udostępnia funkcje iOS 4.0 oraz iOS 4.1 na iPad. AirPlay oraz AirPrint pozwalają na łatwe połączenie naszych urządzeń mobilnych z drukarką oraz z AppleTV. Zapowiedzi iPad 2.0 sugerują, że bedziemy mogli stworzyć wygodne przenośne urządzenia do prezentacji i telekonferencji.

Tak więc otrzymujemy bardzo ciekawą marketingowo platformę. Apple stale inwestuje w rozwój tej platformy coraz większe pieniądzę, więc czego możemy chcieć więcej?

Zanim na dobre się rozgrzejesz, przyjrzymy się drugiej stronie medalu.

Po pierwsze niezbędna będzie inwestycja w komputer z jabłuszkiem na wieku, najtańsza możliwość w Polsce to Mac Mini ze zniżką studencką jedyne 3300PLN plus monitor (to już kwestia gustu), to umożliwia nam naukę, programowanie i testowanie aplikacji na symulatorze. Oczywiście do tego dochodzi przestawienie się z naszego ulubionego IDE na Xcode.

W momencie gdy zechcemy zaaplikować naszą aplikację do Apple AppStore, miejsca, gdzie można zakupić/pobrać aplikację, musimy dokonać aktywacji konta deweloperskiego. Rejestracja kosztuje 99$ na rok. Rejestracja w Polsce jest utrudniona, gdyż należy przesłać faxem formularz zgłoszeniowy z numerem karty kredytowej do Appla. Po upływie kilkunastu dni możemy wysłać naszą aplikację do weryfikacji. Podobnie wygląda sprawa testowania aplikacji na prawdziwym urządzeniu, wymagana jest wcześniejsza rejestracja z opłatą.

Czy warto? - trudno odpowiedzieć na to pytanie, dla mnie było to zdrowe oderwanie od standardowych CRUD-ów webowych, nad którymi najczęściej pracujemy w Enterprise Java J.

Podstawy Objectiv-C

Objective-C jest niewielkim choć z pewnością ciekawym rozszerzeniem ANSI C. Wzoro-

> pod wieloma względami Objective-C jest bardziej zaawansowany niż Java

wany na języku Smalltalk posiada zaawansowane możliwości paradygmatu obiektowego, dzięki czemu otrzymujemy język C, który jest w pełni obiektowy. Jako ciekawostkę podam, że czyste C może również być wykorzystane w ramach projektu w Objective-C i niektóre ze starszych standardowych bibliotek Appla wciąż posiadają typowe dla C API.

Jeżeli w tym momencie zaczynacie myśleć, co takiego musi mieć w sobie iPhone by zmusić mnie do programowania w C (przecież ostatnio robiliśmy to na studiach bo trzeba było oddać program na zaliczenie), spieszę z wyjaśnieniem, że pod wieloma względami Objective-C jest bardziej zaawansowany niż Java.

Tak więc po kolej, jak możesz się spodziewać po obiektowym języku, wszystko kręci się wokół obiektów. Tutaj różnica pomiędzy Java a Objetive-C jest, można by powiedzieć, tylko w składni. O ile instrukcje sterujące mają również C-like składnię, o tyle wywołanie metod zdecydowanie się różni. Wywołanie metod jest zapożyczone z języka Smalltalk, czyli nie wywołujemy metody, a wysyłamy komunikat do odbiorcy. Pozostałe koncepcje, jak single inharitance, interfejsy, polymorfizm możemy uznać za identyczne.

```
[receiver message]; //simple example
[myRectangle setWidth:20.0]; // message with parameter
[myRectangle setOriginX: 30.0 y: 50.0]; message with multiple parameters
```

Kilka ciekawostek, których w Javie nie znajdziecie

Dynamiczne typowanie, Objective-C umożliwia zarówno statyczne jak i dynamiczne typowanie. Statyczne typowanie uzyskujemy analogicznie do języka Java, natomiast do dynamicznego typowania używany jest specjalny typ id, który nie przechowuje informacji o typie. Pozwala to na wykorzytanie potencjału typowania dynamicznego, a tam gdzie istnieje potrzeba, możemy wykorzystać typowanie statyczne.

```
Rectangle *thisObject;
id anObject;
```

NullPointerException nie istnieje, wysłanie komunikatu do obiektu nil zwraca nil, a jeśli wynikiem jest liczba, zwraca 0.

Properties, być może pojawią się w siódmej edycji JAVA, w Objective-C są już obecne. Służy do tego para słów kluczowych @property/@synthesize, które w zależności od parametrów przekazanych do @property utworzą metody dostępowe. Dodatkowo @property pozwala na sterowanie sposobu przypisania właściwości, synchornizacji operacji oraz trybu dostępu (tylko odczyt, odczyt/zapis). Miłym dodatkiem jest umożliwienie używania notacji „kropkowej" do dostępu do właściwości. Objective-C z automatu przekształci je na wywołanie metod „setter/getter".

```
//header
@property (nonatomic, copy) NSString *value;
//body
@synthesize value;
//usage
myInstance.value = @"test";
printf("myInstance value: %d", myInstance.value);
//message
[myInstance setValue:@"test"];
printf("myInstance value: %d", [myInstance value]);
```

Odpowiednikiem interfejsów w języku Java są protokoły w jęzuku Objective-C. Ciekawą możliwością w protokołach jest wyznaczenie sekcji metod opcjonalnych, których klasy implementujące ten protokół nie muszą implementować. Używana jest do tego dyrektywa @optional. Każda klasa ma metodę BOOL conformsToProtocol: Protocol* aProtocol, która odpowiada YES w przypadku, gdy obiekt implementuje dany protokół.

W przypadku opcjonalnych metod możemy użyć selektorów by sprawdzić czy dany obiekt odbiera dany komunikat. Można porównać selektory do klasy Method z reflection API. Wykorzystujemy do tego komunikatu BOOL responseToSelector: (SEL) aSellector. Myślę,

> Całość instalacji zajmuje około 10GB i niestety trwa tyle, co podróż z Radomska do Częstochowy pociągiem pospiesznym.

że kawałek kodu wyjaśni sprawę:

```
//protokół
@protocol MyProtocol
- (void)requiredMethod;
@optional
- (void)anOptionalMethod;
- (void)anotherOptionalMethod;
@required
- (void)anotherRequiredMethod;
@end
//hipotetyczne użycie
if ( ! [receiver conformsToProtocol:@
protocol(MyProtocol)]  ) {
    // tutaj moge używać wszystkich
metod wymaganych, użycie opcjonalnej
może spowodować błąd
 if ( [anObject respondsToSelector:@
selector(anOptionalMethod)] ) {
  // tutaj mogę użyć metody opcjonal-
nej anOptionalMethod
  }
}
```

Kolejnym krokiem są Kategorie, pozwalają one dodawać metody do istniejącej klasy bez potrzeby posiadania kodu źródłowego. Często również mogą być stosowane zamiast dziedziczenia. Wielokrotnie chciałem rozszerzyć klasę String (nie tylko ja, dowodem może być wielość klas nazwanych StringUtils), w Objective-C jest to możliwe dzięki kategoriom.

Ostatnia rzecz to oczywiście oczekiwane od lat „domknięcia". W Objective-C od wersji iOS 4.0 możemy używać „bloków," które są odpowiednikiem „domknięć". Są one najczęściej wykorzystywane do przekazania kodu, który jest później wykonywany w innym kontekście bądź jako alternatywa dla metod typu „callback".

Czego w Objective-C nie znajdziecie.

O ile Objective-C ma 'odśmiecarkę', to w wersji na iOS nie można jej użyć. Częściowo wspomaga nas mechanizm zwany „Reference counting". Mechanizm ten polega na pamiętaniu ilości wskazań na obiekt a w momencie gdy, licznik ten dojdzie do 0, obiekt zostaje zwolniony. Niestety do nas programistów należy dbanie by wartość tego licznika była odpowiednia.

Kolejną bolączką jest brak przestrzeni nazw (pakiety), co jest to przyczyną faktu, że w Objective-C większość z bibliotek poprzedza 2-3 literowe skróty i tak mamy: NSString, OCUnit, itp. Powstała strona, gdzie zarejestrowane są wszystkie prefixy http://www.cocoadev.com/index.pl?ChooseYourOwnPrefix.

Oczywiście nie są to wszystkie różnice pomiędzy językami, ale mam nadzieję, iż pokazałem po pierwsze - że Objective-C nie jest językiem lat 80-tych i można w nim programować, po drugie - że nam Javowcom nauka Objective-C powinna przyjść łatwo, a większym problemem jest tutaj zmiana IDE, niż zmiana języka.

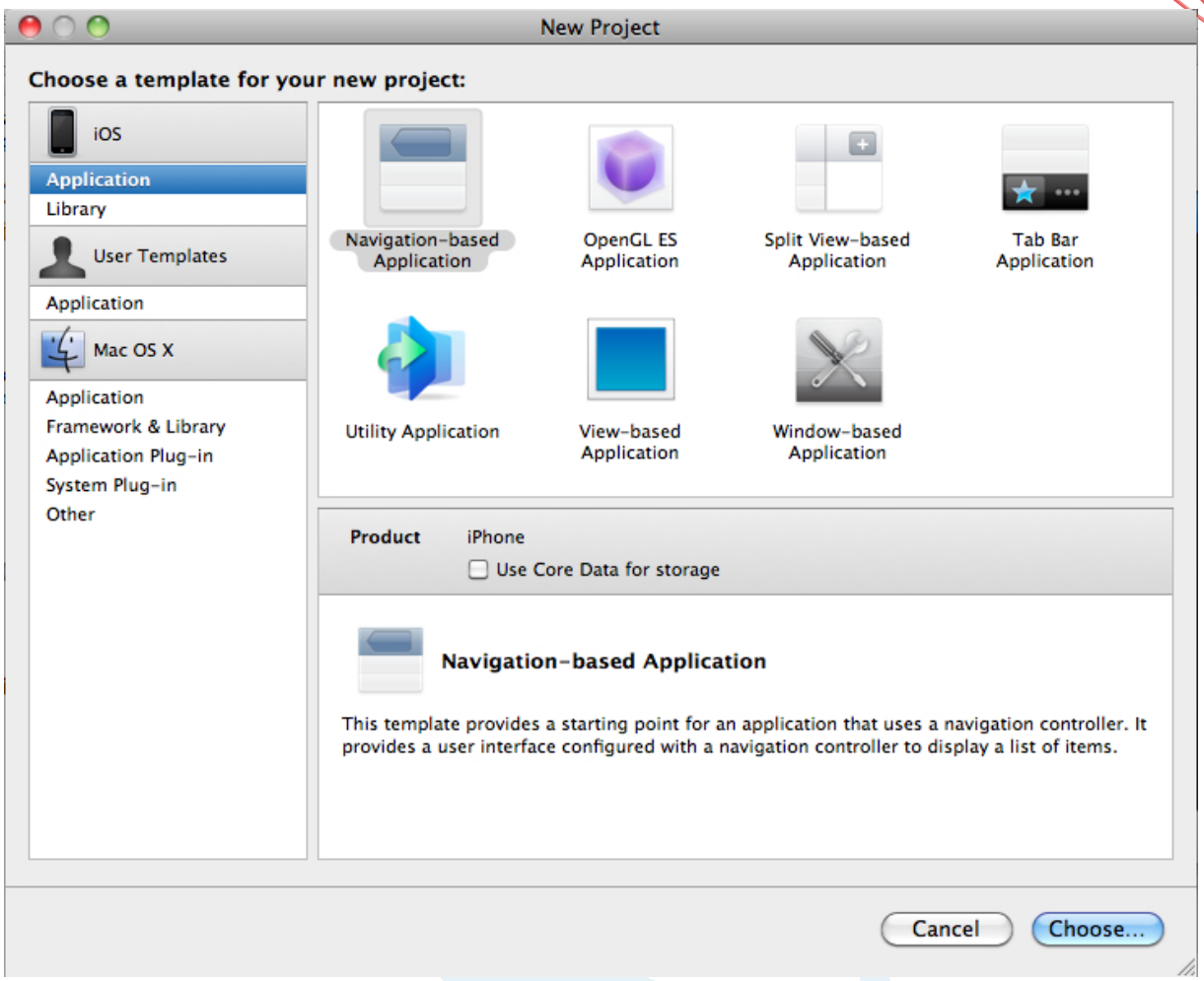
Prosta aplikacja

Postanowiłem, że skupię się na standardowym „Hallo world!", z dużym naciskiem na pokazaniu narzędzi wykorzystywanych w trakcie rozwoju aplikacji, tak by każdy z czytelników mógł razem ze mną utworzyć pierwszą aplikację.

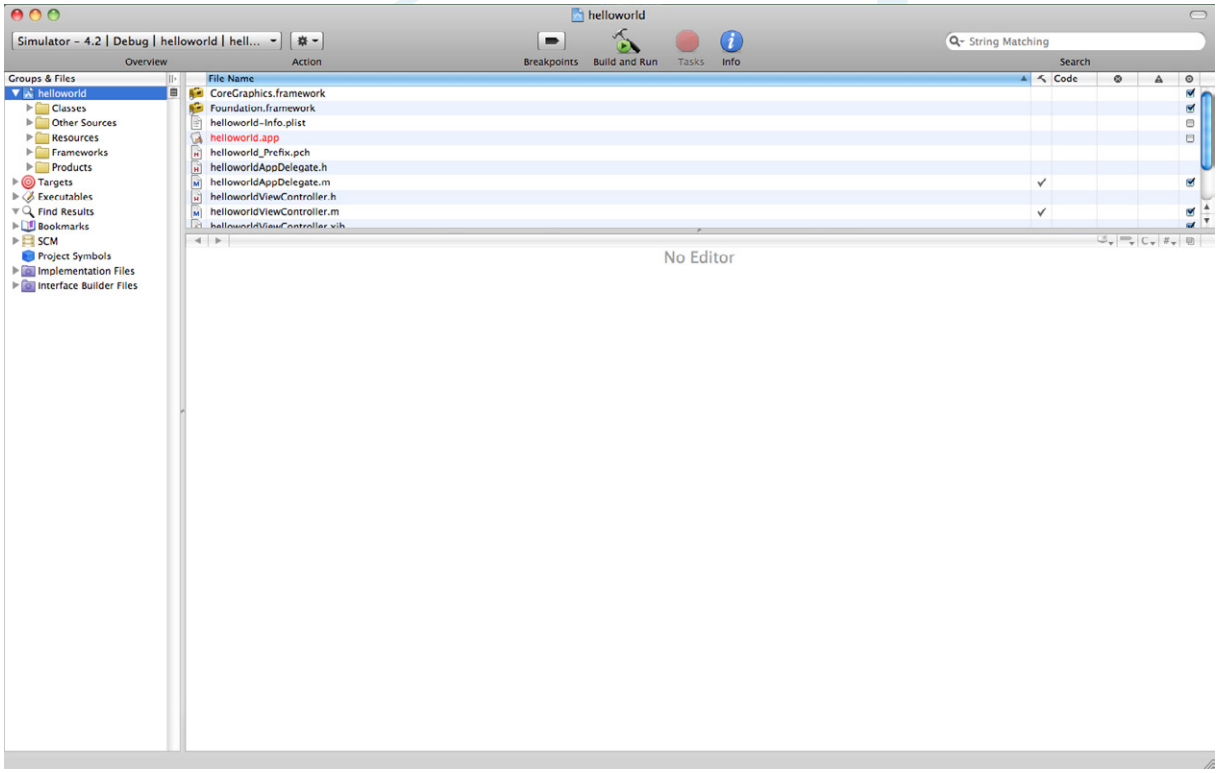
Naszą przygodę rozpoczynamy od instalacji XCode, każdy zarejestrowany użytkownik strony developers.apple.com może pobrać najnowsze wydanie XCode. Niestety z każdym uaktualnieniem musimy pobrać całą paczkę o rozmiarze około 4GB. Całość instalacji zajmuje około 10GB i niestety trwa tyle, co podróż z Radomska do Częstochowy pociągiem pospiesznym. Po zakończeniu instalacji otrzymujemy pełne środowisko do pisania i testowania aplikacji.

Po uruchomieniu środowiska XCode otrzymujemy okno dialogowe, w którym możemy wybrać jeden z projektów, nad którym ostatnio pracowaliśmy, badź „Create a new Xcode project" by utworzyć nowy projekt. Po wybraniu opcji utworzenia nowego projektu otrzymujemy dialog pokazany na rysunku 1.

Na dialogu tym po lewej stronie znajdują się typy dostępnych projektów, w części głównej



Rys 1. Widok dialogu nowego projektu



Rys 2. Xcode w pełnej okazałości

> W fazie beta znajduje się Xcode4, w którym mam wrażenie, Apple zapożyczył wiele funkcjonalności z IntelliJ wprowadzając wiele udogodnień.

znajdują się możliwe rodzaje projektów w ramach wybranego typu. Ze względów szkoleniowych wybierzemy „View-Based Application" i krok po kroku rozwiniemy naszą aplikację, w taki sposób by pokazać proces budowy applikacji na urządzenia pracujące pod kontrolą iOS. Wybieramy więc „View-Based Application" i naciskamy przycisk „Choose", wybieramy miejsce, gdzie umieścimy projekt i nadajemy mu nazwę, zatwierdzamy i już po chwili możemy cieszyć się widokiem podobnym do tego na rysunku 2.,

Teraz po krótce opiszę Xcoda: na górze ekranu znajduje się rozwijane menu w którym określamy parametry uruchomiania aplikacji, zobacz rysunek 3. Na środku w kolejności:

- przycisk breakpoints do przełączenia pomiędzy trybem uruchomienia i śledzenia aplikacji
- przycisk build and run/debug do uruchomienia aplikacji.
- przycisk tasks do aktualnie działających w tle zadań.
- przycisk info do kontekstowego dialogu inspektora.

Po lewej stronie znajduje się drzewo obiektów w naszym projekcie, drzewo to jest drzewem logicznym. Główną część ekranu zajmuje lista plików z edytorem. Zmiana edytowanego pliku następuje po wybraniu pliku z listy, podwójne naciśnięcie powoduje otworzenie dodatkowego okna z zawartością pliku. Wartościowe skróty klawiszowe:

- Option+Command+Up – przełącza pomiędzy plikiem nagłówka (.h) a plikiem implementacji (.m).
- Command +Shift+E - zwiększa edytor (tak, że zajmuje również część listy plików)
- Command +Option+Shift+E - zwiększa edytor (tak, że zajmuje również część listy plików i drzewa obiektów)

Ostatni skrót jest niezwykle trudny do wykonania dlatego warto poświęcić chwilę czasu by zmienić je na własne. Pomocne w pierwszych krokach z XCodem jest zakupienie screencast-u „Becoming Productive in Xcode" http://pragprog.com/screencasts/v-mcxcode/becoming-productive-in-xcode. Na stronach Apple znajdziesz wiele przykładów i informacji o Xcode. Jednym z nich jest „A Tour of Xcode". W fazie beta znajduje się Xcode4, w którym mam wrażenie, Apple zapożyczył wiele funkcjonalności z IntelliJ wprowadzając wiele udogodnień.

Rys 3. Menu uruchomieniowe

Czas na testowe uruchomienie aplikacji, w tym celu w rozwijanym menu rysunek 3 wybieramy odpowiednio opcje: Simulator, Debug, iPhone by uruchomić na symulatorze iPhone-a lub Simulator, Debug, iPad by uruchomić na symulatorze iPad-a. Naciskamy przycisk build and run i po chwili uruchamia się nowy program iOS Symulator z naszą aplikacją, która chwilowo jest po prostu szarym prostokątem.

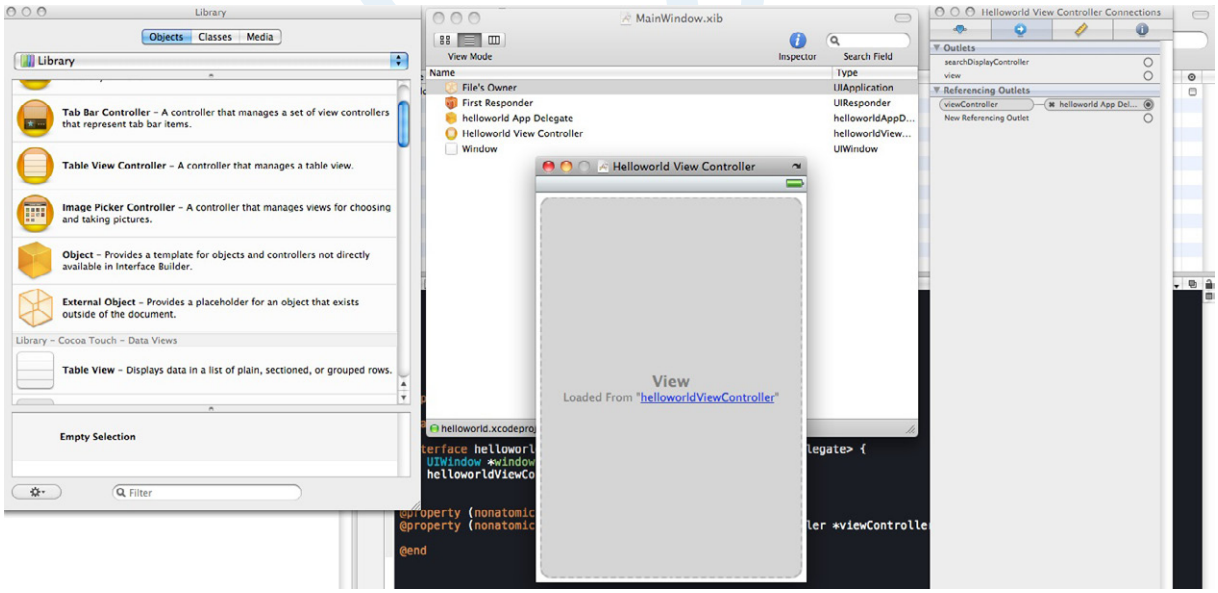
Pora zakodować coś łatwego i pokazać kolejne narzędzia, które są używane do budowy appliacji iOS. Zaraz po wygenerowaniu aplikacji mamy plik main.m, od którego rozpoczyna

> Niestety InterfaceBuilder składa się z wielu okien, co powoduje na początku pewien chaos, zwłaszcza że niektóre okna mogą być zamknięte

się wykonanie applikacji. Plik ten w większości aplikacji nie jest zmieniany, gdyż jedyne co się w nim dzieje, to utworzenie obiektu UIApplicationMain. Obiekt ten wczytuje plik MainWindow.xib (o ile nie skonfigurujemy inaczej), plik ten to zapisane informacje o obiektach wraz z ich grafem połączeń, można przyjąć tu analogie do pliku XML z konfiguracją Spring Framework. Sam plik zawiera XML, ale niestety nie jest zbyt czytelny dla człowieka, do jego edycji używamy programu Interface Builder.

Kliknijmy dwukrotnie na plik MainWindow.xib, otworzy nam się Interface Builder, a całość wygląda mniej więcej jak rysunek 4. Niestety InterfaceBuilder składa się z wielu okien, co powoduje na początku pewien chaos, zwłaszcza że niektóre okna mogą być zamknięte, dlatego najlepiej nauczyć się skrótów klawiatury:

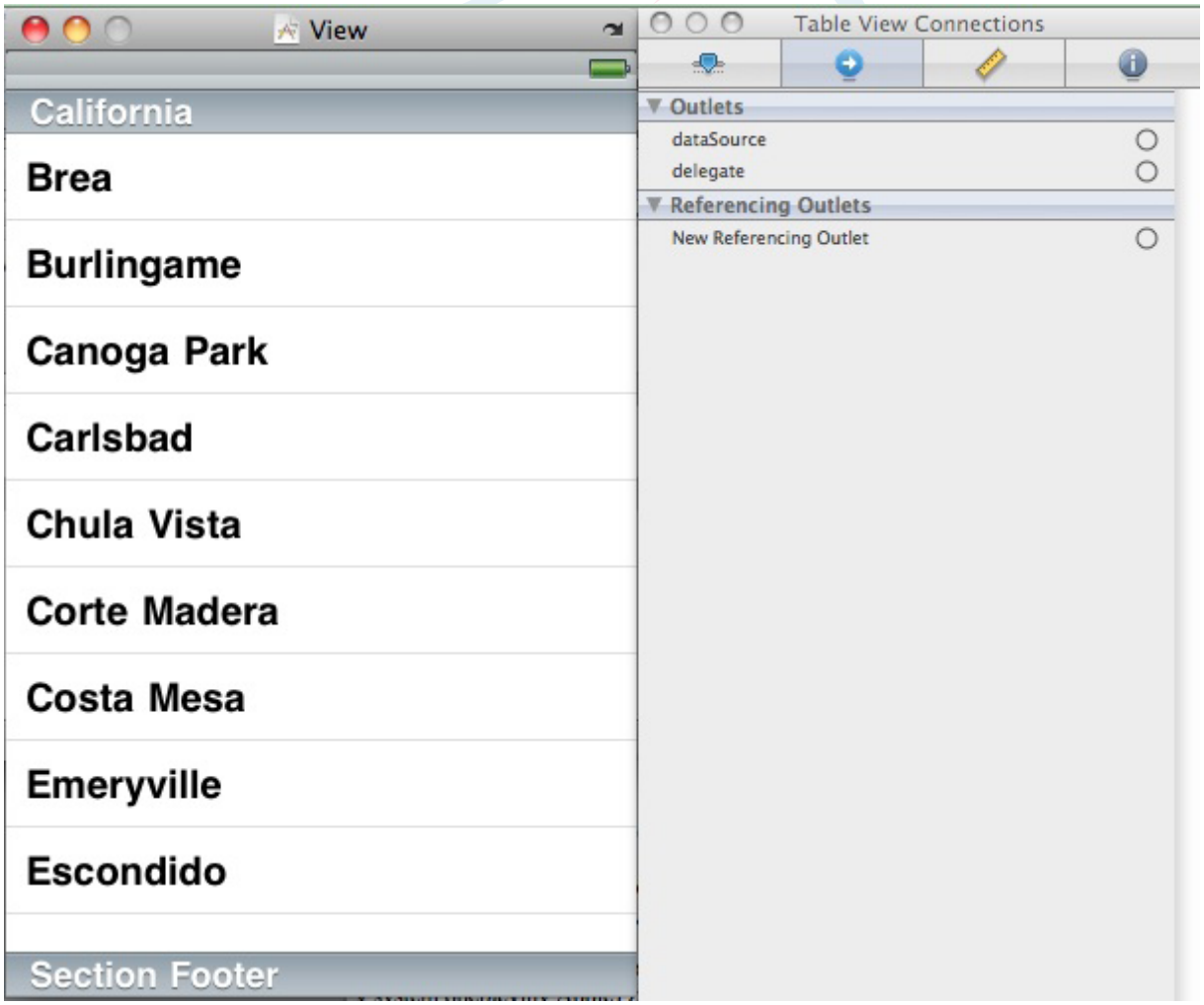
- Shift+Command+L biblioteka obiektów i graficznych elementów (rysunek 4 okno najbardziej po lewej)
- Shift+Command+I – inspektor (rysunek 4 okno najbardziej po prawej). Inspektor ten działa w czterech trybach, zakładki na górze przełączane skrótami
- Command+1 – inspektor atrybutów (edycja właściwości wybranego obiektu – analogiczne do edycji właściwości JavaBeans)
- Command+2 – inspektor połączeń pomiędzy obiektami (tutaj „wstrzykujemy” zależności obiektów)
- Command+3 – inspektor rozmiaru (definiujemy rozmiar obiektów oraz w jaki sposób obiekt ma reagować na zmianę orientacji ekranu)
- Command+4 – inspektor obiektu (definiujemy typ obiektu)
- Alt+Command+Down – miejsce pracy (rysunek 4 okno środkowe dolne okno), działa tylko na obiektach które posiadają widok.
- Domyślnie otwierane okno (rysunek 4 okno środkowe na górze), zamknięcie tego okna kończy pracę z dokumentem. W przypadku gdy pracujemy na wielu dokumentach NIB, możemy się przełączać pomiędzy nimi poprzez wybór Windows->nazwa pliku nib.



Rys 4. Okna programu InterfaceBuilder

> W każdym pliku NIB znajduje się obiekt „File's Owner", który jest głównym obiektem.

W każdym pliku NIB (tak nazywane są pliki z rozszerzeniem xib, skrót NIB najprawdobniej wziął się od nextStep Interface Builder, gdzie NextStep to pierwszy system operacyjny Apple) znajduje się obiekt „File's Owner", który jest głównym obiektem. W przypadku MainWindow.xib nasz „File's Owner" to UIApplication, który odpowiada za uruchomienie całej aplikacji, w zakładce inspektora połączeń, widzimy połączenie pomiędzy „delegate" a obiektem „helloworld App Delegate". Jest to standardowy wzorzec wykorzystywany praktycznie z każdym widokiem, który całą logikę przekazuje do swojego delegata, w tym przypadku klasa UIApplication posiada delegata UIApplicationDelegate. Protokół ten posiada opcjonalne metody, które wywoływane są w przypadku zdarzeń systemowych, na przykład applicationDidEnterBackground wywoływana jest w momencie gdy nasza aplikacja przechodzi w stan pracy w tle. W naszej aplikacji obiekt „helloworld App Delegate" to klasa typu helloworldAppDelegate implementuje ona protokół UIApplicationDelegate oraz definiuje dwa pola window oraz viewController, które następnie zdefiniowane są jako @ property, wraz z dodatkową flagą IBOutlet, dzięki temu InterfaceBuilder pokazuje te pola jako możliwe do połączenia. Połączenia te są widoczne w inspektorze połączeń po wybraniu obiektu „helloworld App Delegate", i tak pole window wskazuje na obiekt Window, a pole viewController wskazuje na obiekt Helloworld View Controller. Oglądając atrybu-



Rys 5. TableView z pustymi zależnościami

> Przygodę z iPhone uważam za ciekawą i wyzywającą.

ty „Helloworld View Controller” zobaczymy że „Nib Name” ustawiony ma wartość helloworldViewController, oznacza to, że kontroler wczyta plik helloworldViewController.xib i na jego podstawie utworzy widok.

Otwórzmy plik helloworldViewController.xib i z biblioteki obiektów przeciągnijmy obiekt TableView na nasz widok – patrz rysunek 5. W inspektorze połączeń widać, że tabela potrzebuje dwóch kolaboratorów dataSource typu UITableViewDataSource oraz delegate typu UITableViewDelegate. Protokół UITableViewDataSource posiada dwie wymagane metody oraz kilka opcjonalnych:

- numberOfRowsInSection – zwraca ilość kolumn dla przekazanej tabeli i sekcji.
- cellForRowAtIndexPath – zwraca obiekt UITableViewCell dla podanego indexu.
- Pozostałe metody są opcjonalne:
  - numberOfRowsInSection – zwraca liczbę sekcji w tabeli,
  - titleForHeaderInSection – zwraca nagłówek sekcji,
  - titleForFooterInSection – zwraca stopkę sekcji,
  - canEditRowAtIndexPath – stwierdza, czy można edytować komórkę,
  - commitEditingStyle – zatwierdza zmiany w edytowanej komórce,
  - canMoveRowAtIndexPath – stwierdza, czy można przesuwać komórki w tabeli,
  - moveRowAtIndexPath – zatwierdza zmiany przy przesunięciu komórek w tabeli,
  - sectionIndexTitlesForTableView – zwraca indeks wyświetlany po lewej stronie,
  - sectionForSectionIndexTitle – zwraca numer sekcji dla indexu.

Drugi potrzebny obiekt to delegat UITableViewDelegate. Wszystkie metody delegata są opcjonalne. Pozwalają one na modyfikacje wyglądu, informują o wyborze kolumn, edycji, zmianie kolejności, co pozwala delegatowi zarządzać danymi. Ponieważ zazwyczaj delegat jak i dataSource działają na tych samych strukturach danych, łączy się je w jedną klasę. Kod źródłowy dostępny jest na stronach java-express.pl.

Oczywiście to nie wszystkie możliwe do użytkowania narzędzia. Pozostałe narzędzia to: debuger, organizer, konsola, instruments, analizator wycieków - są one jednak poza zakresem tego artykułu.

Kolejny proces nie opisany w tym artykule to rejestracja aplikacji w AppStore oraz zarządzanie kluczami. W przypadku zainteresowania chętnie opiszę bardziej zaawansowane funkcjonalności wraz z pozostałymi narzędziami.

Podsumowanie

Przygodę z iPhone uważam za ciekawą i wyzywającą. Urządzenie dostarczyło mi wiele satysfakcji, dla Apple w międzyczasie stało się głównym dochodem, co widać po inwestycjach czynionych w kierunku rozwoju platformy AppStore oraz systemu iOS. System OSX ma pojawić się na wirtualnej maszynie VMWare, co może tylko zwiększyć jego popularność. Czy Android lub Blackberry nadgoni stracony czas? Tego nie wiem, wiem jedynie, że programowanie iPhone po przystosowaniu się do nowego środowiska jest przyjemne, a przy odrobinie szczęścia (czytaj: dobrego pomysłu), może uczynić Cię bogatym!

O autorze

Sebastian Pietrowski prowadzi bloga pietrowski.info. Pasjonat szerokopojętego rzemiosła programistycznego. Kontakt z autorem sebastian.pietrowski [goryl] gmail.com

Transakcje w systemach Java EE: Korzystanie z systemów kolejkowania w serwerach aplikacji

Jarosław Błąd

Systemy kolejkowania w środowisku serwera aplikacji

Możliwość korzystania z systemów kolejkowania w aplikacjach Java Enterprise zdefiniowana jest przez specyfikację Java Message Service (JMS) oraz przez specyfikację samej platformy Java Enterprise. Specyfikacje te definiują również zachowania transakcyjne systemów kolejkowania z punktu widzenia programisty aplikacji.

Wsparcie dla systemów kolejkowania zostało wprowadzone od specyfikacji J2EE 1.3, w której serwer aplikacji musiał wspierać JMS w wersji 1.0. Od specyfikacji J2EE 1.4 serwer aplikacji musi wspierać JMS w wersji 1.1. Z punktu widzenia JMS API od 2002 roku nic w tej materii się nie zmieniło. W praktyce oznacza to, że każdy współczesny serwer aplikacji zawiera własną implementację systemu kolejkowania oraz umożliwia podłączenie się do systemów kolejkowania oferowanych przez zewnętrznych dostawców.

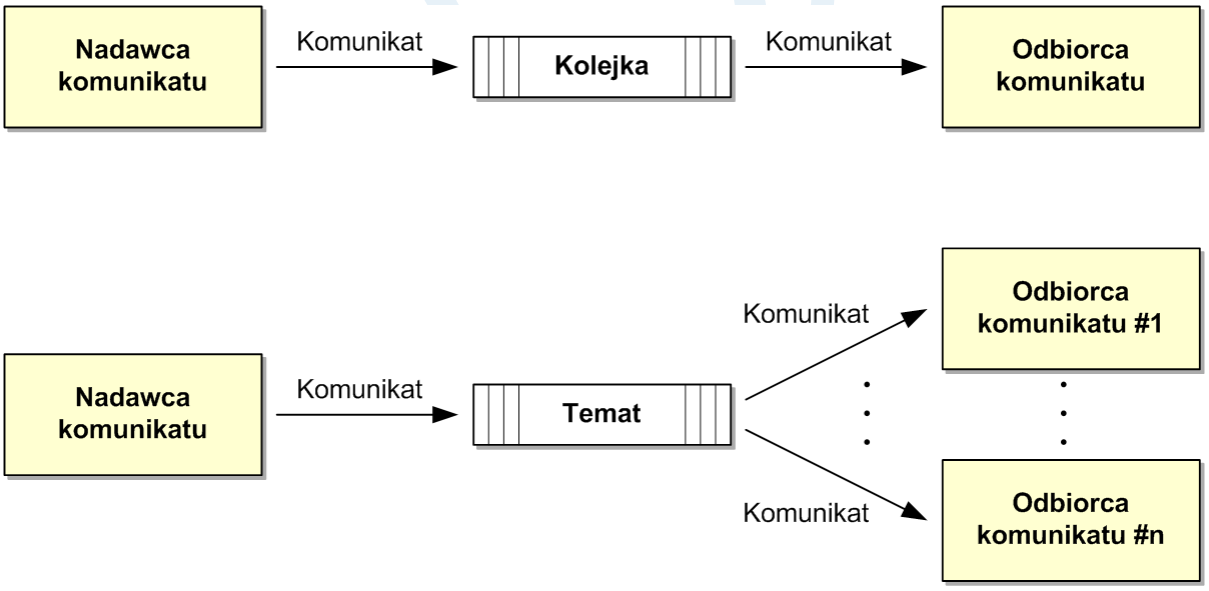
Specyfikacja JMS definiuje dwa sposoby asynchronicznego komunikowania się aplikacji z wykorzystaniem systemów kolejkowania. Są to modele:

- Point to point, w którym mamy jednego nadawcę i jednego odbiorcę, a komunikaty są przesyłane za pomocą kolejki (Queue).
- Publish subscribe, w którym mamy jednego nadawcę i dowolną liczbę odbiorców, a komunikaty są przesyłane (publikowane) za pomocą tzw. tematu (Topic).

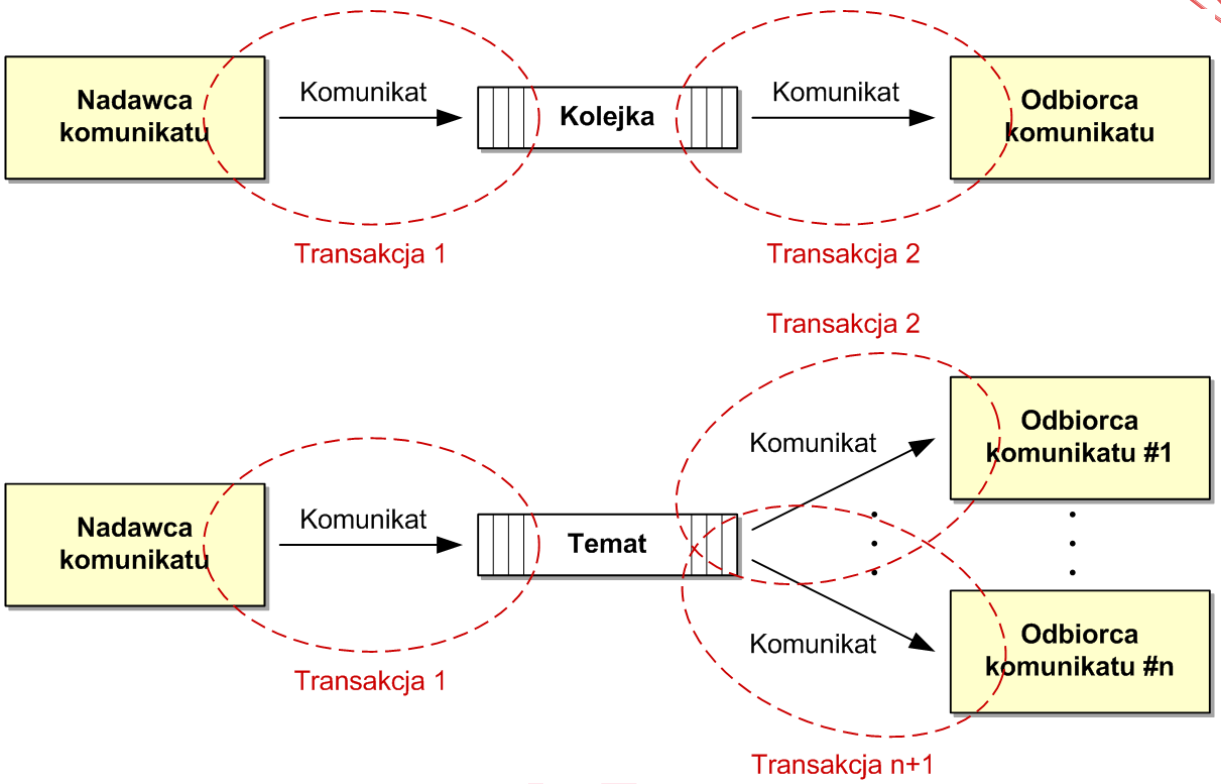
Oba modele przedstawia rysunek 1.

W przypadku korzystania z systemów kolejkowania kluczowe jest zrozumienie, że wysłanie komunikatu a następnie jego odebranie odbywa się w całkowicie odrębnych i niezależnych od siebie transakcjach. W przypadku kolejki (Queue) mamy dwie transakcje – jedną obsługującą wysłanie komunikatu i drugą obsługującą odebranie komunikatu. W przypadku tematu (Topic) mamy jedną transakcję obsługującą wysłanie komunikatu i tyle transakcji obsługujących odebranie komunikatu ile jest zarejestrowanych jego odbiorców. Zagadnienie to ilustruje rysunek 2.

Nie możemy zatem oczekiwać, że uda nam się wysłać komunikat i odebrać odpowiedź w jednej transakcji – jest to częsty błąd pro-



Rys 1. Sposoby asynchronicznego komunikowania się aplikacji z wykorzystaniem systemów kolejkowania



Rys 2. Wysłanie i odebranie komunikatu odbywa się w całkowicie odrębnych i niezależnych od siebie transakcjach

gramistów stawiających pierwsze kroki w systemach kolejkowania. Próba wysłania komunikatu a następnie odebrania komunikatu odpowiedzi w jednej transakcji spowoduje zawieszenie aplikacji. Jest to spowodowane tym, że komunikat nie został w rzeczywistości wysyłany, bowiem transakcja nie została jeszcze zatwierdzona. Nie możliwe jest więc odebranie odpowiedzi na coś, co jeszcze nie zostało wysłane.

Podobnie jak w przypadku baz danych serwer aplikacji tworzy odpowiednie środowisko dla aplikacji, które chcą korzystać z systemów kolejkowania. Na środowisko to składają się następujące elementy:

- Deskryptor aplikacji, w którym twórca aplikacji definiuje logiczne nazwy zasobów JMS (kolejki, tematy), z których będzie korzystał. Dla aplikacji internetowych zasoby te są definiowane za pomocą elementów <resource-ref> w pliku web.xml.
- Definicji fizycznych zasobów JMS, w szczególności:
- Fabryk połączeń do kolejek i tematów (ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory).
- Kolejek (Queue).
- Tematów (Topic).
- Odwzorowania zdefiniowanych w aplikacjach nazw zasobów logicznych na zasoby fizyczne zdefiniowane w konfiguracji serwera. Odwzorowanie to jest realizowane za pomocą deskryptorów specyficznych dla danego serwera aplikacji i jest ustanawiane w momencie instalacji aplikacji na serwerze.
- Dostawcy JMS (JMS Provider), który umożliwia współpracę serwera aplikacji z konkretnym systemem kolejkowania. Jest to odpowiednik sterownika JDBC.
- Menedżera transakcji, który w imieniu aplikacji potrafi zarządzać transakcjami rozproszonymi, w których mogą może uczestniczyć operacje na systemie kolejkowania, o ile dostawca JMS wspiera transakcje rozproszone.

W tym artykule koncentruję się wyłącznie na aspektach transakcyjnego korzystania z systemów kolejkowania. Całkowicie pomijam dość złożoną kwestię konfigurowania dostępu do tych systemów z poziomu serwera aplikacji. Na potrzeby przykładów zakładam, że odpo-

> Wysłanie komunikatu a następnie jego odebranie odbywa się w całkowicie odrębnych i niezależnych od siebie transakcjach.

wiednie zasoby systemów kolejkowania są już poprawnie skonfigurowane i dostępne pod właściwymi kluczami w drzewie JNDI.

Podobnie jak w przypadku korzystania z baz danych w systemach kolejkowania możemy mieć do czynienia z dwoma modelami transakcji – lokalnym i globalnym (rozproszonym). Przyjrzyjmy się dokładniej obu modelom.

Transakcje lokalne

W przypadku transakcji lokalnych możemy operować wyłącznie na zasobach systemu kolejkowania (kolejki i tematy). Sterowanie transakcją odbywa się wtedy za pomocą mechanizmów, które są dostępne w JMS API. W przypadku transakcji lokalnych serwer aplikacji jest odpowiedzialny jedynie za udostępnienie odpowiednich zasobów (ConnectionFactory, Queue, Topic).

Zarządzanie transakcjami jak i wszystkie operacje wysyłania i odbierania komuników za pomocą JMS API odbywają się w ramach sesji (interfejs javax.jms.Session). Sesja JMS definiuje następujące metody pozwalające na zarządzanie lokalną transakcją:

- void commit() - zatwierdza transakcję, co powoduje usunięcie z kolejek pobranych komunikatów oraz dodanie do kolejek wysłanych komunikatów.
- void rollback() - wycofuje transakcję, co powoduje, że pobrane komunikaty nie są usuwane z kolejek, komunikaty wysłane nie są przekazywane do kolejek.
- boolean getTransacted() - zwraca informację, czy sesja pracuje w trybie transakcyjnym.

Jak można zauważyć, nie mamy tutaj metody, która sygnalizowałaby rozpoczęcie transakcji. O tym czy dana sesja jest transakcyjna decyduje sposób jej utworzenie w ramach połączenia z systemem kolejkowania (interfejs javax.jms.Connection). Interfejs ten udostępnia metodę createSession(boolean transacted, int acknowledgeMode) pozwalającą w zależności od parametru transacted stworzyć sesję transakcyjną bądź nietransakcyjną.

W związku z tym, że w JMS API występuje bardziej rozbudowana hierarchia sesji i połączeń, metod tworzenia sesji jest więcej. Pełny obraz sytuacji przedstawia tabela 1 (wszystkie interfejsy są zdefiniowane w pakiecie javax.jms):

Hierarchia ta ma również znaczenie w kontekście zarządzania transakcjami. Jeżeli używamy sesji QueueSession to nie możemy w jej ramach w sposób transakcyjny operować na tematach (Topic). Analogicznie, jeśli używamy TopicSession to nie możemy w jej ramach w sposób transakcyjny operować na kolejkach (Queue). Aby móc w sposób transakcyjny operować zarówno na kolejkach jak i tematach musimy korzystać z bardziej ogólnego zestawu (ConnectionFactory, Connection, Session). Możliwość tą wprowadzono w specyfikacji JMS 1.1.

Warto pamiętać, że drugi parametr metod pozwalających na tworzenie sesji, który determinuje sposób potwierdzania odbioru wiadomości, w przypadku tworzenia sesji transakcyjnych nie ma znaczenia. Zwyczajowo ustawia się go na zero, ale ustawienie go na dowolną wartość i tak nie zmienia zachowania systemu. Jeśli sprawdzimy wartość tego parametru za pomocą metody Session#getAcknowledgeMode() to w przypadku sesji transakcyjnej zawsze otrzymamy wartość Session#SESSION_TRANSACTED.

Fabryka połączeń	Połączenie	Metoda tworzenia sesji	Sesja
`ConnectionFactory`	`Connection`	`createSession`	`Session`
`QueueConnectionFactory`	`QueueConnection`	`createQueueSession`	`QueueSession`
`TopicConnectionFactory`	`TopicConnection`	`createTopicSession`	`TopicSession`

Tabela 1. Metody tworzenia sesji.

> W ramach jednej sesji
> możemy przeprowadzić wiele transakcji lokalnych.

Wykonywanie transakcji lokalnych ma zatem następujący przebieg:

- Pobranie z JNDI odpowiedniej fabryki połączeń (ConnectionFactory).
- Pobranie z JNDI zasobów na których będziemy operować (Queue, Topic).
- Utworzenie połączenia (Connection).
- Utworzenie transakcyjnej sesji JMS (Connection#createSession(true, 0)).
- Wykonanie w ramach sesji transakcyjnej operacji na pobranych zasobach (odbieranie, wysyłanie komunikatów).
- Zatwierdzenie lub wycofanie transakcji (Session#commit(), Session#rollback())
- Zamknięcie sesji i połączeń.

W ramach jednej sesji możemy przeprowadzić wiele transakcji lokalnych. Każde zatwierdzenie lub wycofanie transakcji powoduje, że niejawnie otwierana jest nowa transakcja.

Jeżeli sesja zostanie utworzona jako nietransakcyjna wszystkie komunikaty będą pobierane i wysyłane na bieżąco. Przypomina to pracę z bazą danych w trybie automatycznego zatwierdzania transakcji (java.sql.Connection#setAutoCommit(true)).

Zobaczmy na przykładach zastosowanie tych mechanizmów.

Wysyłanie wielu wiadomości w jednej transakcji lokalnej

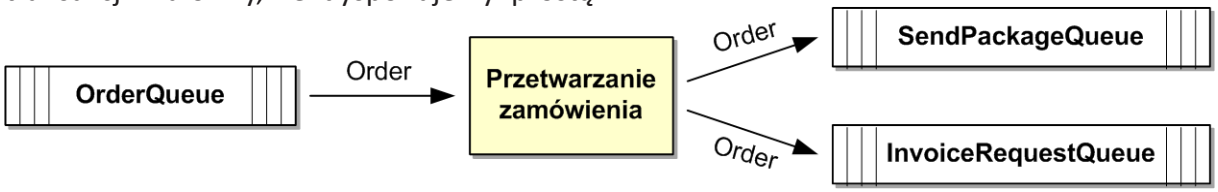
Na początek prosty przykład pokazujący jak można wysłać kilka komunikatów w jednej transakcji. Załóżmy, że dysponujemy prostą klasą reprezentującą zamówienie w sklepie internetowym (Order) i chcemy przekazać do kolejki dwa zamówienia w jednej transakcji. Załóżmy również, że serwer aplikacji jest tak skonfigurowany, że w drzewie JNDI pod kluczem jms/QCF znajduję się fabryka połączeń do kolejek, a pod kluczem jms/OrderQueue kolejka, do której można przekazywać zamówienia.

Jeśli zamiast do kolejki chcielibyśmy w jednej transakcji wysłać wiele komunikatów do tematu (Topic) to przykład byłby analogiczny. Należałoby jedynie zastąpić klasy QueueConnectionFactory, QueueConnection, QueueSender ich odpowiednikami związanymi z obsługą tematów, tj. TopicConnectionFactory, TopicConnection, TopicPublisher. Oczywiście w drzewie JNDI powinny znajdować się obiekty reprezentujące temat a nie kolejkę (patrz Listing 1)

Odbieranie i wysłanie wiadomości w jednej transakcji lokalnej

Rozważmy bardziej złożony przypadek transakcji lokalnej obejmującej zarówno odbieranie jak i wysyłanie komunikatów. Załóżmy, że chcemy napisać kawałek kodu przetwarzającego zamówienia. Zamówienia spływają do kolejki OrderQueue, następnie są przetwarzane przez nasz program i trafiają do dwóch kolejek – SendPackageQueue (skąd trafią do systemu zajmującego się wysyłką paczki z magazynu) oraz InvoiceRequestQueue (skąd trafią do systemu finansowego, który wystawi fakturę za zamówienie). Sytuację tę ilustruje rysunek 3.

Kod który realizuje funkcjonalność opisanego przypadku może wyglądać jak na Listingu 2.



Rys. 3. Odbieranie i wysłanie wiadomości w jednej transakcji lokalnej

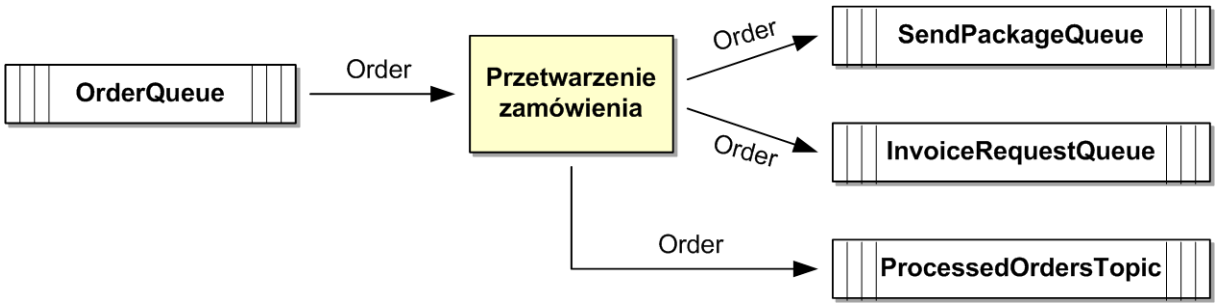
```
...
try {
  Context ctx = new InitialContext();

  QueueConnectionFactory qcf =
    (QueueConnectionFactory) ctx.lookup("jms/QCF");
  QueueConnection connection = qcf.createQueueConnection();
  Queue orderQueue = (Queue) ctx.lookup("jms/OrderQueue");
  QueueSession session = null;
  QueueSender orderQueueSender = null;

  connection.start();
  boolean transacted = true;
  session = connection.createQueueSession(transacted, 0);
  orderQueueSender = session.createSender(orderQueue);

  Order firstOrder = getFirstOrder();
  Order secondOrder = getSecondOrder();

  ObjectMessage firstOrderMessage = session.createObjectMessage(firstOrder);
  ObjectMessage secondOrderMessage =
    session.createObjectMessage(secondOrder);

  orderQueueSender.send(firstOrderMessage);
  orderQueueSender.send(secondOrderMessage);

  session.commit();

  orderQueueSender.close();
  session.close();
  connection.close();
} catch (Exception e) {
  throw new RuntimeException(e);
}
...
```

Listing 1. Wysyłanie wielu wiadomości w jednej transakcji lokalnej

W przykładzie tym zastosowano blokującą operację odbierania komunikatu. Oznacza to, że program będzie czekał na wywołaniu metody orderQueueReceiver.receive(), aż w kolejce pojawi się jakieś zamówienie. Zgodnie ze specyfikacją J2EE można to odbieranie zamienić na nieblokujące w przypadku, gdy pracujemy z poziomu kontenera klienta (Application Client Container). Zastosowanie odbierania blokującego jest jednak bardziej ogólne.

Spróbujemy jeszcze nieco skomplikować powyższy przykład dokładając do kodu obsługi zamówienia wysłanie komunikatu do tematu ProcessedOrdersTopic. Ilustruje to rysunek 4.

Niestety okazuje się, że proste zmodyfikowanie poprzedniego przykładu nie jest możliwe. Wynika to z tego, że jest zastosowany interfejs QueueSession, w ramach którego nie możemy tworzyć obiektów odbierających czy wysyłających komunikatów związanych z tematem (Topic). Należy zastosować nowy mechanizm wysyłania i odbierania komunikatów wprowa-



Rys. 4. obsługa zamówienia z wysłaniem komunikatu do tematu ProcessedOrdersTopi

```
...
try {
  Context ctx = new InitialContext();

  QueueConnectionFactory qcf =
    (QueueConnectionFactory) ctx.lookup("jms/QCF");
  QueueConnection connection = qcf.createQueueConnection();
  Queue orderQueue = (Queue) ctx.lookup("jms/OrderQueue");
  Queue sendPackageQueue = (Queue) ctx.lookup("jms/SendPackageQueue");
  Queue invoiceRequestQueue =
    (Queue) ctx.lookup("jms/InvoiceRequestQueue");
  QueueSession session = null;
  QueueReceiver orderQueueReceiver = null;
  QueueSender sendPackageQueueSender = null;
  QueueSender invoiceRequestQueueSender = null;

  boolean transacted = true;
  session = connection.createQueueSession(transacted, 0);
  orderQueueReceiver = session.createReceiver(orderQueue);
  sendPackageQueueSender = session.createSender(sendPackageQueue);
  invoiceRequestQueueSender = session.createSender(invoiceRequestQueue);

  connection.start();

  // Odebranie zamówienia z koleki
  ObjectMessage orderMessage = (ObjectMessage)orderQueueReceiver.receive();
  Order order = (Order)orderMessage.getObject();

  // Przetworzenie zamówienia...

  // Wysłanie nowych komunikatów
  ObjectMessage packageMessage = session.createObjectMessage(order);
  ObjectMessage invoiceMessage = session.createObjectMessage(order);

  sendPackageQueueSender.send(packageMessage);
  invoiceRequestQueueSender.send(invoiceMessage);

  session.commit();

  orderQueueReceiver.close();
  sendPackageQueueSender.close();
  invoiceRequestQueueSender.close();
  session.close();
  connection.close();
} catch (Exception e) {
  throw new RuntimeException(e);
}
...
```

Listing 2. Odbieranie i wysłanie wiadomości w jednej transakcji lokalnej

dzony w specyfikacji JMS 1.1. Zatem kod, który w pełni realizuje nowe wymaganie może wyglądać jak na Listingu 3.

Jak można zauważyć sposób odbierania i wysyłania komunikatów różni się znacząco od zastosowanego w poprzednim przykładzie. Ten dualizm sposobów odbierania i wysyłania komunikatów ma dość przykre konsekwencje praktyczne. Zazwyczaj rozbudowa istniejącego kodu wymaga dość znacznych przeróbek polegających na wymianie kodu specyficznego dla kolejek czy tematów na kod bardziej ogólny potrafiący obsłużyć jednocześnie kolejki i tematy.

Ograniczenia lokalnych transakcji JMS

Podstawowym i jednocześnie najważniejszym ograniczeniem lokalnych transakcji JMS jest brak możliwości uczestniczenia w takiej transakcji innych zasobów, np. baz danych. Jest to o tyle dotkliwe, że w praktyce większość aplikacji wykorzystuje bazy danych a systemy kolej-

```
...
try {
  Context ctx = new InitialContext();

  ConnectionFactory cf = (ConnectionFactory) ctx.lookup("jms/QF");;
  Connection connection = cf.createConnection();;
  Queue orderQueue = (Queue) ctx.lookup("jms/OrderQueue");
  Queue sendPackageQueue = (Queue) ctx.lookup("jms/SendPackageQueue");
  Queue invoiceRequestQueue = (Queue) ctx.lookup("jms/InvoiceRequestQueue");
  Topic processedOrdersTopic =
    (Topic)ctx.lookup("jms/ProcessedOrdersTopic");
  Session session = null;
  MessageConsumer orderMessageConsumer = null;
  MessageProducer sendPackageMessageProducer = null;
  MessageProducer invoiceRequestMessageProducer = null;
  MessageProducer processedOrdersMessageProducer = null;

  boolean transacted = true;
  session = connection.createSession(transacted, 0);
  orderMessageConsumer = session.createConsumer(orderQueue);
  sendPackageMessageProducer = session.createProducer(sendPackageQueue);
  invoiceRequestMessageProducer =
    session.createProducer(invoiceRequestQueue);
  processedOrdersMessageProducer =
    session.createProducer(processedOrdersTopic);

  connection.start();

  // Odebranie zamówienia z koleki
  ObjectMessage orderMessage =
    (ObjectMessage)orderMessageConsumer.receive();
  Order order = (Order)orderMessage.getObject();

  // Przetworzenie zamówienia...

  // Wysłanie nowych komunikatów
  ObjectMessage packageMessage = session.createObjectMessage(order);
  ObjectMessage invoiceMessage = session.createObjectMessage(order);
  ObjectMessage processedOrderMessage = session.createObjectMessage(order);

  sendPackageMessageProducer.send(packageMessage);
  invoiceRequestMessageProducer.send(invoiceMessage);
  processedOrdersMessageProducer.send(processedOrderMessage);

  session.commit();

  orderMessageConsumer.close();
  sendPackageMessageProducer.close();
  invoiceRequestMessageProducer.close();
  processedOrdersMessageProducer.close();
  session.close();
  connection.close();
} catch (Exception e) {
  throw new RuntimeException(e);
}
...
```

Listing 3. Nowy mechanizm wysyłania i odbierania komunikatów.

kowania stanowią jedynie uzupełnienie związane zazwyczaj z integracją systemów. Teoretycznie możliwe jest wykonanie aplikacji bazującej wyłącznie na przetwarzaniu i przesyłaniu komunikatów. Jednakże w przypadku współczesnych aplikacji biznesowych byłoby to zadanie czysto akademickie, mające niewiele wspólnego w praktyką.

Kolejnym ograniczeniem transakcji lokalnych jest problem związany z odbieraniem komunikatów. W architekturze Java Enterprise w sen-

> W tym przypadku zarządzanie transakcjami odbywa się za pomocą mechanizmów dostarczanych przez serwer aplikacji

sowny sposób można odbieranie komunikatów zrealizować jedynie w kontenerze klienta (Application Client Container), który w praktyce rzadko jest wykorzystywany. Oczywiście standard Java Enterprise przewidział na potrzeby odbierania komunikatów specjalny typ komponentów EJB – Message Driven Bean, ale jest to już związane z zastosowaniem transakcji globalnych (rozproszonych).

Zatem, o ile omawiane w poprzedniej części artykułu, lokalne transakcje JDBC mają duże znaczenie praktyczne, to lokalne transakcje JMS w zasadzie są ograniczone do wąskiej grupy programów standalone, których zadaniem jest przetwarzanie lub rutowanie komunikatów.

Transakcje globalne (rozproszone)

W przypadku transakcji globalnych operacje na zasobach systemu kolejkowania (Queue, Topic) mogą być częścią większej transakcji obejmującej również operacje na innych zasobach (np. bazach danych). W tym przypadku zarządzanie transakcjami odbywa się za pomocą mechanizmów dostarczanych przez serwer aplikacji, a dokładniej przez menedżera transakcji rozproszonych, dostępnego dla programisty aplikacji przez obiekt UserTransaction. Odbywa się to dokładnie w taki sam sposób jak przedstawiłem w poprzednich częściach artykułu.

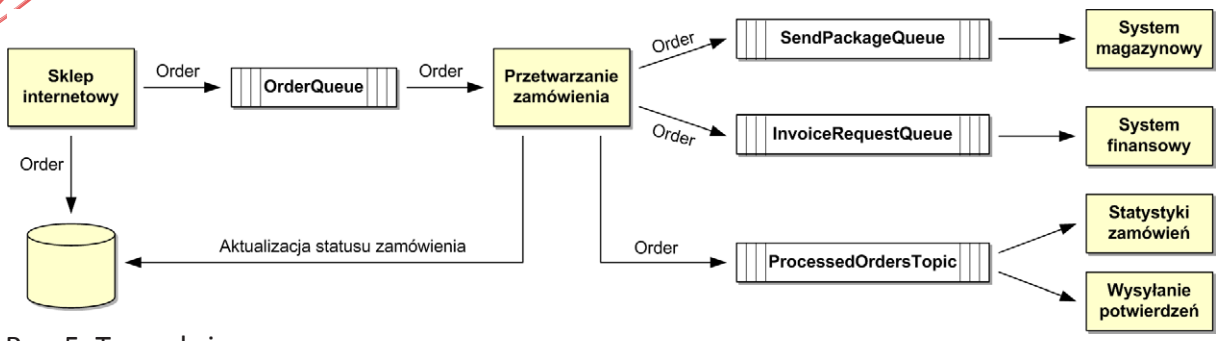
Aby systemy kolejkowania mogły uczestniczyć w transakcji rozproszonej ich dostawca musi zapewnić pracę w ramach dwufazowego protokołu zatwierdzania transakcji. Dostawca takiego systemu musi również przygotować bibliotekę (JMS Provider), która zapewni poprawną współpracę z serwerem aplikacji w ramach interfejsów XA (XAConnectionFactory, XAConnection, XASession itp.). Mechanizm współpracy z serwerem jest analogiczny do tego jaki omawiałem przy okazji transakcji rozproszonych w bazach danych.

Aby pokazać zagadnienie transakcji globalnych w JMS rozbudujmy nasz przykład z systemem przetwarzania zamówień. Składał się on będzie teraz z następujących elementów:

- Aplikacji internetowej zbierającej zamówienia od klientów i posiadającej własną bazę danych zamówień.
- Elementu przetwarzającego zamówienia, który będziemy realizować jako komponent Message Driven Bean (MDB).
- Systemu magazynowego, z którego są wysyłane paczki z zamówieniami.
- Systemu finansowego, w którym są wystawiane faktury.
- Systemu aktualizującego statystyki zamówień.
- Systemu wysyłającego potwierdzenia obsłużenia złożonego zamówienia.
- Kolejki przechowującej zamówienia – OrderQueue.
- Kolejki przechowującej żądania wysłania paczki z magazynu – SendPackageQueue.
- Kolejki przechowującej żądania wystawiania faktury za zamówienie – InvoiceRequestQueue.
- Tematu przechowującego informacje o przetworzonych zamówieniach – ProcessedOrdersTopic.

Zadaniem aplikacji internetowej, realizowanym w jednej transakcji globalnej, będzie:

- Zachowanie zamówienia w bazie danych.
- Przekazanie zamówienia do kolejki OrderQueue.
- Zadaniem elementu przetwarzającego zamówienie, również w jednej transakcji globalnej, będzie:
- Odebranie zamówienia z kolejki OrderQueue.



Rys. 5. Transakcje rozproszone

- Zlecenie wysłania paczki do systemu magazynowego przez wysłanie komunikatu do kolejki SendPackageQueue.
- Zlecenie wystawienia faktury do systemu finansowego przez wysłanie komunikatu do kolejki InvoiceRequestQueue.
- Aktualizacja statusu zamówienia w bazie danych aplikacji internetowej.
- Powiadomienie innych systemów o przetworzeniu zamówienia przez wysłanie komunikatu do tematu ProcessedOrdersTopic.

Całość zagadnienia ilustruje rysunek 5.

Operacje na bazie danych i wysyłanie komunikatów do kolejki w jednej transakcji globalnej

Rozpocznijmy od kodu realizującego interesujący nas fragment funkcjonalności aplikacji internetowej. Korzystamy w nim z omawianego w poprzednich częściach artykułu interfejsu UserTransaction, żeby zarządzać zakresem naszej transakcji (ut.begin(), ut.commit()). Wewnątrz transakcji wykonujemy kolejno operacje na bazie danych i kolejce zamówień. Kod realizujący omawianą funkcjonalność może wyglądać jak na Listingu 4.

Zwróćmy uwagę, że nie zakładamy transakcyjnej sesji JMS oraz nie wykonujemy na niej operacji commit(). Wynika to z tego, że w trakcie trwania transakcji globalnej nie jest dozwolone wywoływanie metod commit()/rollback() na sesji JMS. Jest to sytuacja analogiczna do tej, z którą mieliśmy do czynienia przy transakcjach globalnych w bazie danych. Tam również na obiekcie połączenia do bazy danych nie mogliśmy wykonywać operacji commit()/rollback().

Odbieranie i wysyłanie komunikatów oraz operacje na bazie danych w jednej transakcji globalnej

Przyjrzyjmy się teraz implementacji kluczowego elementu naszego systemu, czyli komponentu którego zadaniem jest przetworzenie zamówienia według opisanych wcześniej założeń. Specyfikacja J2EE przewidziała do tego celu specjalny typ komponentu EJB – Message Driven Bean (MDB). Jest to bardzo prosty komponent, posiadający jedną istotną metodę onMessage(), która jest wywoływana w momencie, gdy w kolejce lub temacie znajdzie się komunikat gotowy do pobrania. Wymaga to oczywiście spięcia komponentów MDB odpowiedzialnych za przetwarzanie właściwych typów komunikatów z odpowiednimi zasobami na serwerze – odpowiednio kolejkami lub tematami. Odbywa się podczas deploymentu aplikacji na serwerze i jest realizowane za pomocą deskryptorów aplikacji specyficznych dla danego serwera. Więcej informacji na temat komponentów MDB można znaleźć w specyfikacji EJB.

Zobaczmy zatem jak może wyglądać implementacja naszego elementu przetwarzającego zamówienia w postaci komponentu MDB (Listing 5).

Do właściwego działania komponent MDB wymaga jeszcze odpowiedniego opisu, który instruuje serwer jak komponent powinien się zachowywać oraz jakich elementów wymaga do swojego działania. We wcześniejszych wersjach specyfikacji J2EE opis taki był dostarczany w postaci pliku ejb-jar.xml, w nowszych zamiast niego można stosować adnotacje – szczegóły w specyfikacji EJB. Na Listingu 6 znajduje się przykład deskryptora do komponentu z Listingu 5.

Kilka słów wyjaśnienia wymaga zachowanie transakcyjne komponentu MDB. W kompo-

```
...
try {
  Context ctx = new InitialContext();
  UserTransaction ut =
    (UserTransaction)ctx.lookup("java:comp/UserTransaction");;

  ut.begin();

  Order order = getOrder();

  // wstaw zamówienie do bazy danych
  DataSource ds =
    (DataSource)ctx.lookup("java:comp/env/jdbc/TransactionDemoDS");
  Connection conn = ds.getConnection();
  insertOrder(order, conn);
  conn.close();

  // wstaw zamówienie do kolejki
  QueueConnectionFactory qcf =
    (QueueConnectionFactory) ctx.lookup("jms/QCF");;
  QueueConnection connection = qcf.createQueueConnection();;
  Queue orderQueue = (Queue) ctx.lookup("jms/OrderQueue");
  QueueSession session = null;
  QueueSender queueSender = null;

  connection.start();
  session = connection.createQueueSession(false, 0);
  queueSender = session.createSender(orderQueue);

  ObjectMessage orderMessage = session.createObjectMessage(order);

  queueSender.send(orderMessage);

  queueSender.close();
  session.close();
  connection.close();

  ut.commit();
} catch (Exception e) {
  throw new RuntimeException(e);
}
...
```

Listing 4. Operacje na bazie danych i wysyłanie komunikatów do kolejki w jednej transakcji globalnej

```
public class ProcessOrderMDB implements MessageDrivenBean, MessageListener {
  MessageDrivenContext mdbctx = null;

  public ProcessOrderMDB() {}

  public void setMessageDrivenContext(MessageDrivenContext mdbctx)
      throws EJBException{
    this.mdbctx  = mdbctx;
  }

  public void ejbRemove() throws EJBException {}
  public void ejbCreate() throws EJBException {}
  public void onMessage(Message msg) {
    ObjectMessage objectMessage = (ObjectMessage) msg;
    try {
      Order order = (Order)objectMessage.getObject();

      // przetworzenie zamówienia...

      Context ctx = new InitialContext();
```

```
        // przekazanie zamówienia do kolejek
        QueueConnectionFactory qcf = (QueueConnectionFactory)
          ctx.lookup("jms/QCF");
        QueueConnection queueConnection = qcf.createQueueConnection();
        Queue sendPackageQueue = (Queue) ctx.lookup("jms/SendPackageQueue");
        Queue invoiceRequestQueue = (Queue)
          ctx.lookup("jms/InvoiceRequestQueue");
        QueueSession session = null;
        QueueSender sendPackageQueueSender = null;
        QueueSender invoiceRequestQueueSender = null;

        queueConnection.start();

        session = queueConnection.createQueueSession(false, 0);
        sendPackageQueueSender = session.createSender(sendPackageQueue);
        invoiceRequestQueueSender =
          session.createSender(invoiceRequestQueue);

        ObjectMessage packageMessage = session.createObjectMessage(order);
        ObjectMessage invoiceMessage = session.createObjectMessage(order);

        sendPackageQueueSender.send(packageMessage);
        invoiceRequestQueueSender.send(invoiceMessage);

        sendPackageQueueSender.close();
        invoiceRequestQueueSender.close();
        session.close();
        queueConnection.close();

        // aktualizacja statusu zamówienia
        DataSource ds=(DataSource)
          ctx.lookup("java:comp/env/jdbc/TransactionDemoDS");
        Connection conn = ds.getConnection();
        updateOrderStatus(order, conn);
        conn.close();

        // przekazanie zamówienia do tematu
        TopicConnectionFactory tcf = (TopicConnectionFactory)
          ctx.lookup("jms/TCF");
        TopicConnection topicConnection = tcf.createTopicConnection();
        Topic processedOrdersTopic =
          (Topic)ctx.lookup("jms/processedOrdersTopic");
        TopicPublisher processedOrdersTopicPublisher = null;

        topicConnection.start();
        TopicSession topicSession =
          topicConnection.createTopicSession(false, 0);
        processedOrdersTopicPublisher =
          topicSession.createPublisher(processedOrdersTopic);
        ObjectMessage processedOrderMessage =
          topicSession.createObjectMessage(order);
        processedOrdersTopicPublisher.publish(processedOrderMessage);

        topicSession.close();
        topicConnection.close();
      } catch (Exception e) {
        throw new RuntimeException(e.getMessage(), e);
      }
    }
}
```

Listing 5. Implementacja elementu przetwarzającego zamówienia w postaci komponentu MDB

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBe-
ans 2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">

<ejb-jar id="ejb-jar_1">

  <description><![CDATA[JMS Transcation Demo]]></description>
  <display-name>jms-transaction-demo</display-name>

  <enterprise-beans>
    <message-driven id="MessageDriven_ProcessOrder">
      <description><![CDATA[Process order bean]]></description>

      <ejb-name>ProcessOrderMDB</ejb-name>
      <ejb-class>pl.epoint.transactiondemo.ejb.ProcessOrderMDB</ejb-class>

      <transaction-type>Container</transaction-type>
      <acknowledge-mode>Auto-acknowledge</acknowledge-mode>

       <message-driven-destination>
         <destination-type>javax.jms.Queue</destination-type>
       </message-driven-destination>

       <resource-ref id="ResRef_DS_1">
         <description>Transaction Demo Datasource</description>
         <res-ref-name>jdbc/TransactionDemoDS</res-ref-name>
         <res-type>javax.sql.DataSource</res-type>
         <res-auth>Container</res-auth>
         <res-sharing-scope>Shareable</res-sharing-scope>
       </resource-ref>

       <resource-env-ref>
         <resource-env-ref-name>jms/InvoiceRequestQueue
         </resource-env-ref-name>
         <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
       </resource-env-ref>

       <resource-env-ref>
         <resource-env-ref-name>jms/SendPackageQueue</resource-env-ref-name>
         <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
       </resource-env-ref>

       <resource-env-ref>
         <resource-env-ref-name>jms/ProcessedOrdersTopic
         </resource-env-ref-name>
         <resource-env-ref-type>javax.jms.Topic</resource-env-ref-type>
       </resource-env-ref>
    </message-driven>
  </enterprise-beans>

  <assembly-descriptor >
  </assembly-descriptor>
</ejb-jar>
```

Listing 6. Deskryptor komponentu MDB.

nentach MDB możemy wybrać dwa rodzaje zachowania transakcyjnego (znacznik transaction-type w deskryptorze komponentu):

- Container - transakcje zarządzane przez kontener EJB (Container Managed Transaction).
- Bean – transakcje zarządzane samodzielnie przez programistę (Bean Managed Transaction) w ramach kodu komponentu MDB.

W przypadku obsługi transakcji przez kontener najpierw jest otwierana transakcja przez kontener, następnie w ramach tej transak-

```
public class ProcessOrderMDBBMT implements
    MessageDrivenBean, MessageListener {
  ...
  public void onMessage(Message msg) {
    ObjectMessage objectMessage = (ObjectMessage) msg;

    try {
      Order order = (Order)objectMessage.getObject();

      Context ctx = new InitialContext();
      UserTransaction ut = mdbctx.getUserTransaction();

      ut.begin();

      // przetworzenie zamówienia...

      // przekazanie zamówienia do kolejek
      QueueConnectionFactory qcf =
        (QueueConnectionFactory) ctx.lookup("jms/QCF");
      QueueConnection queueConnection = qcf.createQueueConnection();;
      Queue sendPackageQueue = (Queue) ctx.lookup("jms/SendPackageQueue");
      Queue invoiceRequestQueue =
        (Queue) ctx.lookup("jms/InvoiceRequestQueue");
      QueueSession session = null;
      QueueSender sendPackageQueueSender = null;
      QueueSender invoiceRequestQueueSender = null;

      queueConnection.start();

      session = queueConnection.createQueueSession(false, 0);
      sendPackageQueueSender = session.createSender(sendPackageQueue);
      invoiceRequestQueueSender =
        session.createSender(invoiceRequestQueue);

      ObjectMessage packageMessage = session.createObjectMessage(order);
      ObjectMessage invoiceMessage = session.createObjectMessage(order);

      sendPackageQueueSender.send(packageMessage);
      invoiceRequestQueueSender.send(invoiceMessage);

      sendPackageQueueSender.close();
      invoiceRequestQueueSender.close();
      session.close();
      queueConnection.close();

      // aktualizacja statusu zamówienia
      DataSource ds=
        (DataSource)ctx.lookup("java:comp/env/jdbc/TransactionDemoDS");
      Connection conn = ds.getConnection();
      updateOrderStatus(order, conn);
      conn.close();

      // przekazanie zamówienia do tematu
      TopicConnectionFactory tcf =
        (TopicConnectionFactory) ctx.lookup("jms/TCF");
      TopicConnection topicConnection = tcf.createTopicConnection();
      Topic processedOrdersTopic =
        (Topic)ctx.lookup("jms/processedOrdersTopic");
      TopicPublisher processedOrdersTopicPublisher = null;

      topicConnection.start();
      TopicSession topicSession =
        topicConnection.createTopicSession(false, 0);
      processedOrdersTopicPublisher =
        topicSession.createPublisher(processedOrdersTopic);
      ObjectMessage processedOrderMessage =
```

```
            topicSession.createObjectMessage(order);
          processedOrdersTopicPublisher.publish(processedOrderMessage);

          topicSession.close();
          topicConnection.close();

          ut.commit();
        } catch (Exception e) {
          throw new RuntimeException(e.getMessage(), e);
        }
      }
    }
```

Listing 7. Metoda onMessage(...) dla komponentu MDB, który samodzielnie zarządza transakcją.

cji pobierany jest komunikat z kolejki i dopiero wtedy wywoływana jest metoda onMessage(...) na komponencie MDB. Jeśli podczas wywołania metody onMessage(...) zostanie rzucony wyjątek kontener ma możliwość zwrócenia komunikatu do kolejki.

W przypadku obsługi transakcji przez programistę występuje całkowicie odmiennie zachowanie. Najpierw komunikat jest pobierany z kolejki, następnie wywoływana jest metoda onMessage(...) na komponencie MDB, w której programista sam otwiera transakcję. W tym przypadku operacja pobrania komunikatu z kolejki nie uczestniczy w transakcji, a więc kontener nie ma żadnej możliwości zwrócenie komunikatu do kolejki.

Zalecanym sposobem zarządzania transakcjami w przypadku komponentów MDB jest zarządzanie przez kontener, gdyż tylko wtedy błąd w trakcie przetwarzania powoduje, że komunikat wraca do kolejki i może być ponownie przetworzony.

Dla porządku poniżej przedstawiłem przykładową implementację metody onMessage(...) dla komponentu MDB, który samodzielnie zarządza transakcją (Listing 7).

W deskryptorze takiego komponentu element transaction-type musi przyjąć następującą postać:

```
<transaction-type>Bean</transaction-type>
```

W ramach wywołania metody onMessage() komponentu MDB możemy również wywoływać inne komponenty EJB, które w zależności od ich konfiguracji mogą się podłączać do trwającej transakcji. Szerzej będę o tym pisał w kolejnej części artykułu.

Podsumowanie

W przykładach dla zwiększenia ich czytelności pominąłem właściwą obsługę wyjątków. W rzeczywistym systemie należy ją oczywiście dodać. Obsługę wyjątków w lokalnych transakcjach JMS można wykonać wzorując się na przykładzie z poprzedniej części artykułu, gdzie pokazałem w jaki sposób zrealizować obsługę wyjątków podczas realizacji lokalnych transakcji JDBC. Z kolei obsługę wyjątków w globalnych (rozproszonych) transakcjach wykonujemy w sposób identyczny jak to pokazałem w pierwszej części artykułu. Szczególną uwagę należy zwrócić na obsługę wyjątków w komponentach MDB. Specyfikacja EJB definiuje jakie powinno być zachowanie komponentów i kontenera w przypadku rzucenia wyjątku w trakcie wykonania metody biznesowej. Temat ten będę poruszał bardziej szczegółowo podczas omawiania transakcji w komponentach EJB.

Jak można zauważyć, dużą część kodu przedstawionych przykładów, stanowią operacje pomocnicze takie jak pobranie obiektów z JNDI, utworzenie połączeń, utworzenie sesji, zamykanie sesji, zamykanie połączeń, właściwa obsługa wyjątków itp. Samego kodu realizującego logikę jest stosunkowo niewiele. W praktyce mając do obsłużenia kilkadziesiąt, czy kilkaset operacji na kolejkach ciężko byłoby je pisać w przedstawiony sposób. Na szczęście można działania pomocnicze stosunkowo łatwo opakować w klasy narzędziowe. W przykładach pozostawiłem je celowo, żeby pokazać pełną złożoność współpracy z serwerem aplikacji.

Transakcje z wykorzystaniem systemów kolejkowania wydają się być idealnym narzędziem do integracji systemów, które muszą

pracować transakcyjnie, zachowując jednocześnie zalety komunikacji asynchronicznej. Jak to zwykle bywa diabeł tkwi w szczegółach. A w tym przypadku w szczegółach współpracy serwera aplikacji z systemami kolejkowania. Systemy kolejkowania dostarczane razem z serwerem aplikacji raczej nie sprawiają kłopotów. Natomiast współpraca z zewnętrznymi systemami kolejkowania napotyka na sporo trudności. Szczególnych problemów możemy się spodziewać w zakresie pracy systemów kolejkowania w ramach transakcji globalnych. Niestety specyfikacja JMS nie wymusza na dostawcy dostarczenia systemu wspierającego transakcje rozproszone. Określa ona jedynie, że jeśli dostawca zdecyduje się umożliwić pracę w transakcji rozproszonej, to musi być zrobiona według zasad określonych w specyfikacji JMS (głównie chodzi o implementację szeregu interfejsów XA). Można śmiało powiedzieć, że jest to temat zdecydowanie gorzej dopracowany niż współpraca serwera aplikacji z różnymi bazami danych.

W następnej części artykułu postaram się przedstawić zagadnienie związane z transakcjami w komponentach EJB oraz strategie, które możemy zastosować budując systemy transakcyjne w technologii Java Enterprise.

Literatura:

[1] Java Message Service 1.1

[2] Java 2 Enterprise Edition Specification 1.3 oraz 1.4

[3] EJB Specification 2.0, 2.1, 3.0

[4] Java Transaction API Specification

[5] Mark Little, Jon Maron, Greg Pavlik, Java Transaction Processing, Prentice Hall, 2004

[6] Dokumentacja do serwera aplikacji JBoss

[7] Dokumentacja do serwera aplikacji IBM Websphere

Zarządzanie sobą w czasie - Pułapki

Łukasz Lechert

W codziennych zmaganiach z obowiązkami może się wydawać, że czas którym dysponujemy alokujemy racjonalnie i jesteśmy często przekonani, że są to czynności trywialne, wręcz intuicyjne. Najprostszym tego przykładem jest poniższa anegdota.

Wyobraź sobie teraz stół na którym stoi pusty duży słoik, obok niego kamienie, żwir, piasek i woda. Spróbuj włożyć do niego najpierw kamienie, maksymalnie jak się da. Czy słoik jest pełen? Nie. Spróbuj teraz wypełnić go żwirem, miejsca które pozostały wolne. Czy słoik jest pełen? Nie. Następnie dosyp piasku, żeby zapełnić całkowicie słój. Czy słoik jest pełen? Nie. Dolej na koniec wody, jeśli jest to jeszcze możliwe. Dopiero teraz słoik jest pełny.

Anegdota jest przenośnią odnoszącą się do zarządzania sobą w czasie. Najpierw włóż duże kamienie, czyli „sprawy ważne, ale niepilne", następnie żwir - „sprawy ważne i jednocześnie pilne". Piasek to sprawy pilne i nieważne, a na końcu zajmuj się sprawami nieważnymi i niepilnymi – alokuj go bezkierunkowo, czyli dolej trochę „wody" do swojego życia.

Na pierwszy rzut oka, łatwo jest zarządzać sobą w czasie. Istnieją jednak złe nawyki i sytuacje, które można zaliczyć do negatywnych zjawisk. Wielokrotnie słyszymy zdanie „Nie mam czasu". Próbując rozmawiać z kolegami, którzy ciągle gonią czas odnosimy wrażenie, że tempo ich życia jest dynamiczne, ich czas płynie szybko, a obowiązki powodują, że wypowiadają często bez propozycji przełożenia rozmowy utarte „nie mam teraz czasu, jestem zajęty".

Przyczyną z pewnością jest podejmowanie się wielu czynności w krótkiej perspektywie czasowej. Jeżeli osoba jest w stanie sprostać zajęciom, bierze na siebie jeszcze więcej odpowiedzialności i nowych zadań, w końcu tempo jej życia przyspiesza. Z czasem przyzwyczajeni i zahartowani w bojach stajemy się zniewoleni przez życie w biegu. Szybciej mówimy, planujemy, wykonujemy zadania, których jest już sporo. Ciągle jak bumerang wraca myśl o kolejnym nowym, ambitnym i dającym satysfakcję zadaniu. Czy awansują mnie, jak dam z siebie jeszcze więcej? W tej sytuacji możemy oddalić się od bliskich, zapomnieć o zdrowym trybie życia lub najzwyklej zachorować. W optymistycznym wariancie możemy stać się zasłużonym lub wysłużonym pracownikiem, który nie jest do niezastąpienia.

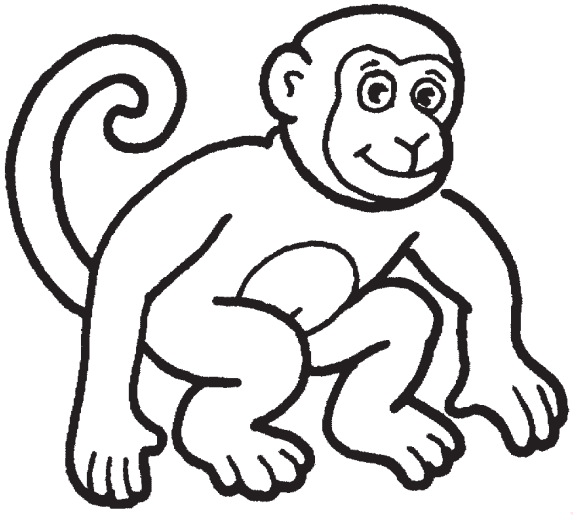
Odpoczynek jest jednym z kluczy do sukcesu. Wielu odmawia sobie tego prawa, czując pustkę w sobie i poczucie winy z powodu stereotypowego lenistwa. Warto pamiętać o zasadzie 3x8 (8 godzin pracy, 8 godzin poświęconemu własnemu ja i relacji z innymi oraz 8 godzin snu). Podobnie jak nadmiernie eksploatowane maszyny, których silniki często się przegrzewają, a następnie stają, człowiek może uzależnić się od „skoków adrenalinowych". Kolejne zadanie może stanowić przed nami wyzwanie, powoduje wzrost adrenaliny w organizmie, która jest potrzebna do dostarczenia właściwych reakcji na nowe sytuacje stresowe. Po zrealizowaniu zadania następuje jej spadek oraz przyjemne odprężenie i satysfakcja. Dodatkowo, bardzo często zapracowana osoba jest dowartościowywana przez otoczenie. Ludzie podziwiają to, że tak dobrze radzimy sobie z taką ilością obowiązków. W ten sposób jesteśmy nagradzani przez innych za fakt przepracowywania się.

Wielokrotnie dostępny czas w pracy organizuje nam szef, zlecając nowe zadania o najwyższym priorytecie. Niekiedy również system, czyli procedury, koledzy z zespołu proszący o wsparcie. Generalnie chodzi o problemy do rozwiązania, które nie są naszymi problemami.

W przypadku szefa i procedur firmowych sprawa nie podlega dyskusji, wykonujemy zadanie najszybciej i najlepiej jak potrafimy – przynajmniej się staramy. Warto jednak negocjować priorytety przed wykonaniem tych czynności, jeśli zadanie koliduje z dotychczas rozpoczętymi.

Życie nie zawsze jest przyjemne, a los sprawiedliwy. Posługując się pojęciem „małpy" - metaforą problemu, można pokusić się o stwier-

> Rolą osoby zarządzającą sobą w czasie jest sprawna obrona przed „obcymi małpami”.



dzenie – właściciel małpy nie zawsze przychodzi po wsparcie. Zdarza się, że celem jest przekazanie małpy na nasze barki i szybkie pozbycie się niewygodnego zadania.

Czy sytuacja jest patowa? Czy mogę odmówić koledze, bezwarunkowego przejęcia jego obowiązków?

Rolą osoby zarządzającą sobą w czasie jest sprawna obrona przed „obcymi małpami”. Na pewno warto nauczyć małpę cierpliwego czekania na swoją kolej. Małpę należy karmić o ustalonych porach, karmienie małpy powinno dotyczyć wsparcia i propozycji, a nie serwowania gotowych rozwiązań na talerzu. Chcąc sprawnie poruszać się w „małpim świecie” warto przyjąć zasadę „Właściciel przyszedł z małpą i z małpą wychodzi”.

Informacje o autorze

Autor jest absolwentem specjalizacji Inżynieria Oprogramowania Politechniki Wrocławskiej oraz Studium Podyplomowego Zarządzanie Projektem na Poznańskim Uniwersytecie Ekonomicznym. Interesuje się systemami wspierającymi procesy logistyczne, zagadnieniami zarządzania projektem oraz oprogramowaniem o otwartym kodzie.

Kontakt : lukasz.lechert@gmail.com

Tworzenie komponentów i separacja odpowiedzialności w aplikacjach Flex

Bill Bejeck (Tłum. Elżbieta Jarecka)

Warunki wstępne

W celu odniesienia jak największej korzyści z przeczytania tego artykułu wskazana jest wcześniejsza znajomość Flex Builder'a, technik programowania obiektowego i ActionScript w wersji 3.0.

Adobe Flex i Flex Builder są doskonałymi narzędziami przeznaczonymi do budowania bogatych, wysoko funcjonalnych aplikacji internetowych. Jako programista w Javie byłem zaskoczony faktem, jak bardzo znajomy wydawał się dla mnie być ActionScript 3.0, co sprawiało że krzywa uczenia się go była względnie płaska. Kiedy tylko zacząłem używać Flex'a, wspaniałe było też zauważyć, iż mogłem używać większości, jeżeli nie wszystkich, znanych mi metod tworzenia oprogramowania.

Ten artykuł prezentuje dwie moje ulubione metody: tworzenie komponentów przy użyciu mieszanki dziedziczenia i kompozycji oraz egzekwowanie zasady podziału odpowiedzialności (ang. separation of concerns), szczególnie pomiędzy warstwą prezentacyjną aplikacji a logiką biznesową.

Wstęp

Kiedy zacząłem używać narzędzia Visual Designer dostępnego z Flex Builder, zaimponowało mi jak szybko byłem w stanie stworzyć interfejs graficzny. Byłem już na najlepszej drodze do ukończenia znaczącej części aplikacji kiedy zacząłem zauważać coś znajomego. Doświadczenie zdobyte przy programowaniu aplikacji sieciowych w Javie mówiło mi, że strona JSP powinna być użyta wyłącznie do prezentacji danych, natomiast cała logika biznesowa powinna znajdować się w klasach. Jeżeli już koniecznie musi się dodać kod do strony JSP, powinien być on zamknięty w specjalnie w tym celu zdefiniowanych znacznikach (ang. custom tags).

Zacząłem więc przyglądać się mojej aplikacji w Flex'ie i rzeczywiście znalazłem w niej grupę plików MXML ze znacznikami <mx:Script></mx:Script>, które opisywały funkcjonowanie moich komponentów. Odczułem potrzebę zrobienia dokładnie tego, czego nauczyłem się pracując z inną platformą – oddzielenia prezentacji danych od logiki biznesowej. W tym poradniku opiszę technikę nazywana „ukryty kod" (ang. code behind), w której wizualne komponenty są zdefiniowane w całości w ActionScript 3.0, ale używają MXML kiedy trzeba zdecydować jak mają być wyrenderowane.

Uwaga: Opisana metoda jest ustrukturyzowanym podejściem do tworzenia aplikacji. Jeżeli czytelnik musi szybko utworzyć funkcjonujący prototyp, używanie techniki „code behind" nie jest w każdym przypadku konieczne.

Na pierwszy rzut oka używanie tej metody może być nieco onieśmielające, ale jak tylko nabierze się w niej wprawy staje się ona faktycznie dużo łatwiejsza.

Krótko mówiąc, wszystko co trzeba zrobić to rozszerzyć istniejącą klasę typu komponent Flex'a i dodać do niej parametry które same są wizualnymi komponentami. Pokażę ten proces na przykładzie bardzo prostego interfejsu używającego DataGrid, TextInput i dwa przyciski. Oto krótki opis kroków które trzeba w tym celu wykonać:

- utwórz nową klasę dziedziczącą po klasie DataGrid i dodaj do niej funkcje umożliwiające przedstawienie rezultatów wyszukiwania,
- utwórz nową klasę dziedziczącą po klasie Panel w której będą prezentowane wszystkie inne komponenty,
- dodaj do powyższej klasy prywatne zmienne reprezentujące zawarte w niej obiekty,
- dodaj metody pobierania i ustawiania dla powyższych zmiennych,
- dodaj metody przechwytujące zdarzenia (ang. event listeners) do swoich komponentów i dodaj metody opisujące zachowanie poszczególnych komponentów w zależności od wykonanej akcji,

- stwórz plik MXML oparty na klasie stworzonej w pierwszym kroku i zmapuj w nim komponenty z drugiego kroku do prywatnych zmiennych,
- dodaj tak przygotowany widok komponentów to głównego pliku MXML aplikacji.

A teraz przyglądnijmy się każdemu z tych kroków z osobna.

Dziedziczenie po DataGrid

Nowa klasa, dziedzicząca po DataGrid, będzie nazywać się SearchResultGrid. Będzie ona względnie prostą klasą, wyświetlającą rezultaty poszukiwania dostępne w formacie XML.

Aby ją utworzyć należy wykonać następujące kroki:

- dodać trzy tabele: jedną odwzorowującą wartości elementów do kolumn, drugą przechowującą etykiety kolumn i trzecią zawierającą obiekty typu DataGridColumn,
- dodać metodę displaySearchResults.

W pierwszym kroku dodaj trzy zmienne klasowe typu Array i nazwij je fields, headers and cols. Tabela fields będzie zawierała stringi mapujące elementy otrzymanego pliku XML do ich wizualnej reprezentacji w kolumnie o tym samym indeksie. Tabela labels będzie zawierała stringi używane jako etykiety nagłówków kolumny o tym samym indeksie. Tabela cols powinna być zainicjalizowana jako pusta tabela o tej samej długości co tabele fields i headers.

Następnie należy nadpisać chronioną metodę commitProperties w klasie SearchResultGrid:

- uzyskaj po kolei dostęp do pól tabeli fields (albo labels),
- w każdej pętli stwórz obiekt typu DataGridColumn,
- ustaw własności dataField i headerText objektu DataGridColumn używając wartości z odpowiednich elementów tabel fields i labels,
- dodaj nowo utworzony obiekt typu DataGridColumn do tabeli cols,
- ustaw tabele cols jako wartość właściwości columns odziedziczonej po klasie DataGrid.

Metoda commitProperties powinna wyglądać mniej więcej tak:

```
override protected
    function commitProperties():void{
  super.commitProperties();
  _cols = new Array(_fields.length);
  for(var i:int=0;
    i<_fields.length;i++){

  var column:DataGridColumn =
    new DataGridColumn();
    column.dataField = _fields[i];
    column.headerText = _labels[i];
    _cols[i] = column;
  }
  this.columns = _cols;
}
```

To mniej więcej wszystko co należy zainicjalizować w klasie SearchResultGrid. Jedyne co nam teraz pozostaje to napisanie metody displaySearchResults, która jako parametr otrzyma obiekt typu XML, zainicjalizuje nim obiekt typu XMLListCollection, a następnie użyje tego ostatniego do ustawienia właściwości dataProvider naszej klasy. Rezultaty wyszukiwania są wtedy automatycznie wyświetlone w SearchResultGrid. Funkcja ta powinna wyglądać tak:

```
public function
    displaySearchResults(
      search:XML):void{
  this.dataProvider =
    new XMLListCollection(
      search.result);
}
```

Moim prywatnym zdaniem aktualizacja wyświetlanych danych nie mogła by być prostsza! Chciałbym przy tym zwrócić uwagę, że przyjąłem tu dwa założenia:

- plik XML z wynikami poszukiwania ma jeden albo więcej elementów <result>...</result>,
- ilość i nazwy elementów zawartych pomiędzy znacznikami <result>...</result> odpowiadają zdefiniowanej wcześniej tabeli fields.

> W tym momencie nasza aplikacja jest prawie gotowa

Stwórz klasę dziedziczącą po klasie Panel i dodaj do niej prywatne zmienne

Na tym etapie możesz dodać klasę SearchPanel, dziedziczącą po klasie Panel która będzie kontenerem zawierającym cały interfejs. Następnym krokiem będzie dodanie prywatnej zmiennej do tej klasy, typu SearchResultGrid, a potem zatroszczenie się o zmienną typu TextInput i dwa obiekty typu Button. W tym momencie Twoja klasa SearchPanel powinna wyglądać mniej więcej tak:

```
public class SearchPanel
    extends Panel{
  private var _searchResultGrid:
    SearchResultGrid;
  private var _searchButton:Button;
  private var _clearButton:Button;
  private var _searchTerm:TextInput;
```

Dodaj metody pobierania (get) i ustawiania (set)

Na tym etapie musisz zdefiniować jak „podłączyć" znajdujące się w klasie SearchPanel obiekty do niej samej, co nie powinno sprawić większego kłopotu. Dla każdej zdefiniowanej wcześniej prywatnej zmiennej należy mianowicie dodać metodę pobierania (get) i metodę ustawiania (set).

Metody te powinny mieć następującą formę:

```
public function set searchResultGrid
  (searchResultGrid:
    SearchResultGrid):void{
  this._searchResultGrid =
    searchResultGrid;
}
public function get
  searchResultGrid():
    SearchResultGrid{
  return this._searchResultGrid;
}
```

Oczywiście równie łatwo można byłoby zdefiniować każdą wspomnianą zmienną jako zmienną publiczną, ale określmy je raczej jako prywatne i użyjmy metod dostępowych (ang. accessor) i mutujących (ang. mutator). Takie podejście ułatwi nam w przyszłości zdefiniowanie ewentualnych dodatkowych akcji, które miałyby być wykonane przez nasz obiekt w trakcie ustawiania czy pobierania owych zmiennych. Para metod get i set musi być dodana dla każdego obiektu, który ma być „wstrzyknięty" przez Flex w nasz panel do poszukiwań. Jak to dokładnie zrobić zostanie przedyskutowane później, kiedy dojdziemy do etapu tworzenia pliku MXML dla klasy SearchPanel.

Dodaj metody przechwytujące zdarzenia

W tym momencie nasza aplikacja jest prawie gotowa – jedyne co nam pozostało zrobić to dodać metody przechwytujące zdarzenia (ang. event listeners) do przycisków. Naturalne mogłoby się wydawać dodanie tych metod już w konstruktorze klasy SearchPanel (ja w każdym razie zrobiłem dokładnie coś takiego kiedy po raz pierwszy używałem metody „code behind") ale powoduje to generację błędu dostępu do metody albo zmiennej przy użyciu niezdefiniowanej referencji – w chwili gdy wywoływany jest konstruktor klasy SearchPanel, obiekty zawarte w tej klasie jeszcze nie istnieją! Prawidłowym rozwiązaniem jest nadpisanie metody childrenCreated i dodanie w niej metod przechwytujących zdarzenia, oraz ustawienie wszystkich innych właściwości obiektów współpracujących.

Na tym etapie wszystkie zdefiniowane przez Ciebie metody mutujące zostały już wywołane, więc można bez obaw używać każdy z zależnych obiektów. W metodzie childrenCreated dodasz kod który sprawi że przyciski SearchButton i ClearButton będą nasłuchiwać zdarzeń wywołanych przez kliknięcia w nie. SearchButton wywoła wtedy metodę doSearch i użyje w niej tekst zczytany z obiektu TextInput. ClearButton z kolei po prostu usunie jakikolwiek tekst widniejący w TextInput.

Dodatkowo konieczne jest zadbanie o prawidłowe zainicjowanie tabel fields i labels używanych w obiekcie typu SearchResultGrid opisanym wcześniej.

> Jak łatwo rozpoznać, otrzymujemy w ten sposób bardzo nieskomplikowany plik MXML

Metoda childrenCreated będzie wyglądała mniej więcej tak:

```
override protected function
  childrenCreated():void{
  _searchButton.addEventListener(
    MouseEvent.CLICK,doSearch);
  _clearButton.addEventListener(
    MouseEvent.CLICK,clearText);
  _searchResultGrid.fields =
    new Array(
      „title","author","date");
  _searchResultGrid.labels =
    new Array(
    „Title","Author","Publish Date");
}
```

Jedyne co jeszcze musimy zrobić to dodać metodę doSearch. Skorzysta ona z klasy HttpService, aby wysłać string z kryteriami wyszukiwania do wybranego programu wyszukującego. Metoda onSearchComplete będzie z kolei odpowiedzialna za wyświetlenie przesłanego z serwera rezultatu poszukiwań – wywoła ona metodę displaySearchResults obiektu searchResultsGrid.

W ten sposob zakończyliśmy pracę nad panelem.

Stwórz warstwę prezentacyjną dla SearchPanel

Następnym krokiem jest stworzenie pliku MXML odpowiedzialnego za wizualną prezentację komponentu. W omawianym przez nas projekcie nazwaliśmy pliki MXML analogicznie do klas ActionScript które reprezentują, z dodaniem sufiksu „View" - tak więc plik MXML dla SearchPanel powinien nazywać się SearchPanelView i wyglądać mniej więcej tak:

```
<?xml version="1.0"
  encoding="utf-8"?>
<SearchPanel
  xmlns="bbejeck.example.*"
  xmlns:comp="bbejeck.example.*"
  xmlns:mx=
    "http://www.adobe.com/2006/mxml"
  layout="absolute">
  <comp:SearchResultGrid
    id="searchResultGrid"
    percentWidth="100"
    percentHeight="60"/>
  <mx:TextInput
    id="searchTerm" x="215" y="320" />
  <mx:Button x="215" y="350"
    id="searchButton"
    label="Search" />
  <mx:Button x="315" y="350"
    id="clearButton" label="Clear"/>
</SearchPanel>
```

Jak łatwo rozpoznać, otrzymujemy w ten sposób bardzo nieskomplikowany plik MXML zawierający wyłącznie informacje dotyczące wizualnej prezentacji SearchPanel. Najważniejszą rzeczą do zauważenia jest fakt, iż identyfikator (id) każdego komponentu jest zmapowany do odpowiedniej metody ustawiania zdefiniowanej w klasie SearchPanel – w ten właśnie sposób klasa ActionScript i plik MXML są ze sobą powiązane. Warto też odnotować obecność samodzielnie zdefiniowanego obiektu <comp:SearchResultGrid../> pomiędzy standardowymi obiektami Flex'a jak Button czy TextInput.

Dodaj tak zdefiniowany SearchPanelView do głównego pliku MXML

Aby móc skorzystać z świeżo zdefiniowanego SearchPanelView we własnej aplikacji, należy dodać następujący kod do głównego pliku MXML:

```
<comp:SearchPanelView
  id="searchPanel"
  title="Search Panel"
  horizontalCenter="0"
  verticalCenter="0"
  width="600" height="500" />
```

Reasumując, łatwo jest zauważyć, iż stosowanie metody „code behind" prowadzi do tworzenia bardzo zwięzłego i łatwego do utrzymania kodu. Dzięki stosowaniu tego podejścia, w miarę budowania coraz bardziej skomplikowanych aplikacji, otrzymuje się nie tylko gamę komponentów, ale opieka nad istniejącym kodem i wprowadzanie zmian w aplikacji pozostają jak najbardziej wykonalne.

> Nie jest konieczne tworzenie pliku MXML dla każdego objektu

Następne kroki

Na koniec parę praktycznych wskazówek:

- aby podłączyć SearchPanel do autentycznego źrodła danych, dostosuj tabele headers i arrays tak aby pasowały do pliku XML będącego wynikiem poszukiwań oraz ustaw własność URL objektu HTTPService w SearchPanel,
- nie jest konieczne tworzenie pliku MXML dla każdego objektu, lepszym rozwiązaniem jest dodanie pojedynczego pliku MXML dla komponentu zawierającego inne objekty,
- nazywając swoje pliki MXML, używaj wzorca „nazwa klasy“ plus „View“,
- omówiona aplikacja jest bardzo uproszczonym przykładem na budowanie komponentów w aplikacjach Flex. Zapoznaj się z innymi metodami ich tworzenia i wypróbuj rożne podejścia aby przekonać się co jeszcze jest możliwe,
- książki i artykuły dotyczące programowania obiektowego i algorytmów są bardzo pomocne,
- dwa źrodła informacji które dla mnie samego były bardzo wartościowe to:
  - Essential ActionScript 3.0 Colin Moock
  - Advanced ActionScript 3 z Wzorcami Projektowymi.

Testowanie z użyciem Mockito

Mateusz Mrozewski

Testy modułowe to wiecznie żywy temat regularnie pojawiający się na blogach i w różnych artykułach. Bez trudu można odnaleźć niezwykle dużo informacji o tym jak testować, jak tego nie robić, o różnych ideach, które na testach modułowych wyrosły, takich jak TDD i innych. Prędzej czy później każdego programistę dopada potrzeba zapoznania się z tym tematem. Rozpoczynamy pierwsze kroki, przerabiamy pierwsze samouczki, testy naszych klas Calculator i MoneyConverter działają niczym marzenie. Później wracamy do rzeczywistości.

Okazuje się, że nasze klasy nie są tak łatwo testowalne. Przyczyn bywa wiele. Jednym z problemów do przeskoczenia w testach modułowych prawdziwego kodu są zależności od innych klas. Nasza testowana klasa ma zależności od innych klas (np. pochodzących z gotowych bibliotek), których testować nie chcemy i nie musimy (albo nawet nie wiemy jak). Chcemy przetestować tylko naszą część kodu, a odnośnie klasy testowanej przyjąć tylko pewne założenia. W takiej sytuacji klasy, które testujemy zastępujemy „imitacjami", z ang. mock. Zobaczymy też, jak tą ideę wspierają biblioteki Mockito i PowerMock.

Poziom trudności

Artykuł przeznaczony dla osób mających już podstawową wiedzę o testowaniu modułowym. Dla osób nie posiadających takiego doświadczenia sugerowane jest zapoznanie się z wprowadzeniem do testowania na stronach http://junit.org

Przykładowa klasa do przetestowania

Jak przykład rozpatrzymy klasę UserService. Jest to klasa usługowa, która wykonuje dla nas pewne operacje związane z obsługą użytkowników, takie jak logowanie, rejestracja. Klasa UserService wszelkie operacje bazodanowe deleguje do klasy UserDAO.

```
public class UserService {
  public User register(String email,
      String password) {
    User user =
      new User(email, password);
    UserDAOImpl dao =
      new UserDAOImpl();
    dao.save(user);
    return user;
  }
}
```

Wywołanie metody register powinno utworzyć nam nowego użytkownika o podanym adresie email i haśle, zapisać go w bazie i zwrócić do dalszego przetwarzania. O ile fakt utworzenia i zwrócenia użytkownika jest łatwy do przetestowania, o tyle zależność od klasy UserDAO nie jest już taka trywialna:

- Instancja tej klasy jest tworzona w metodzie
- Ta klasa nie ma wydzielonego interfejsu
- Wiemy, że metoda save odwołuje się do bazy danych (a być może i do innych zasobów) co łamie zasadę niezależności testów modułowych

A gdyby tak podmienić implementację klasy UserDAO na czas wykonania testu, zakładając, że robi ona to co powinna i robi to dobrze (w końcu to test modułowy)?

Ręczna podmiana implementacji

Zmodyfikujmy trochę naszą klasę UserService, aby umożliwić podstawienie innej implementacji klasy UserDAO.

```
public class UserService {
  private UserDAO dao;
  public UserService(UserDAO dao) {
    this.dao = dao;
  }
  public User register(String email,
      String password) {
    User user =
      new User(email, password);
    user = dao.save(user);
    return user;
  }
}
```

> Problem jaki odkryliśmy powyżej to brak „efektów ubocznych”

Zauważmy, że teraz w konstruktorze możemy podać dowolną klasę dziedziczącą po UserDAO. Możemy też klasę UserDAO zamienić w DefaultUserDAO i wydzielić interfejs UserDAO, co będzie jeszcze lepszym posunięciem, gdyż będziemy trzymać się zasady programowania do interfejsu.

W tym momencie na czas testu możemy podstawić do klasy UserService naszą imitację:

```
public class MockUserDAO
    implements UserDAO {
  public void save(User user) {
  }
}
```

Nasza imitacja nie robi nic (przypomnijmy, że prawdziwa implementacja wykonałaby najpierw pewne operacje na bazie danych). Możemy już zatem modułowo przetestować metodę register: przekażemy adres email i hasło, a dostaniemy z powrotem instancję klasy User o określonych wartościach poszczególnych pól. Nasz test mógłby wyglądać tak:

```
public class UserServiceTest {
  @Test
  public void testRegister() {
    UserDAO dao = new MockUserDAO();
    UserService service =
      new UserService(dao);
    User user = service.register(
      "matixo@gmail.com",
      "secretpassword");
    Assert.assertEquals(
      "matixo@gmail.com",
      user.getEmail());
    Assert.assertEquals(
      "secretpassword",
      user.getPassword());
  }
}
```

Oczywiście metoda podstawienia implementacji UserDAO przez konstruktor nie jest jedyną, ale to temat na oddzielny artykuł.

Czy taki test jest w pełni poprawny i wiarygodny? Jeśli tak, to spróbujmy zakomentować następującą linijkę kodu w metodzie register:

```
// user = dao.save(user);
```

Czy test nadal wykonuje się poprawnie?

Czas na Mockito

Problem jaki odkryliśmy powyżej to brak „efektów ubocznych” kodu wywoływanego z naszych klas. Jeśli nasz testowany kod wchodzi w interackję z innymi klasami, to ta interakcja jest ważna z punktu widzenia testu. Jeśli ona nie następuje, to nasz kod nie działa prawidłowo. Ale jak zatem zbadać tą interakcję, skoro nie powoduje ona zmiany żadnych wartości? Zwykły assert już nam nie pomoże. Jedym skutkiem ubocznym tej interakcji byłby zapis do bazy danych, ale tego właśnie próbujemy uniknąć wprowadzając imitację prawdziwej implementacji.

Z pomocą przychodzi nam Mockito. Jest to biblioteka ułatwiająca nam życie asystując nam w tworzeniu imitacji, ale pozwala nam dodatkowo na weryfikację interakcji pomiędzy klasami.

```
@Test
public void testRegister() {
  UserDAO mockDao =
    Mockito.mock(UserDAO.class);
  UserService service =
    new UserService(mockDao);
  User user = service.register(
    "matixo@gmail.com",
    "secretpassword");

  Mockito.verify(mockDao).save(
    Mockito.any(User.class));
  Assert.assertEquals(
    "matixo@gmail.com",
    user.getEmail());
  Assert.assertEquals(
    "secretpassword",
    user.getPassword());
}
```

Co zmieniło się w kodzie naszego testu?

Przede wszystkim sposób utworzenia imitacji. Wykorzystaliśmy metodę mock(), która dla danego interfejsu (albo klasy) utworzy „wydmuszkę” - klasę z domyślną (wg. Mockito) implementacją. Klasa taka zwraca domyślnie false dla boolean, puste kolekcje, null zamiast obiektów, itp. W naszym wypadku będzie to nieistotne, gdyż nasza metoda save i tak nic nie zwracała. Jeśli jednak zajdzie po-

> Dzięki Mockito nasze testy mogą stać się pełniejsze, a życie spokojniejsze

trzeba zwracania pewnych domyślnych wartości, to Mockito jak najbardziej to umożliwia.

Warto nadmienić również, iż istnieje inny sposób tworzenia imitacji poprzez oznaczenie pól klasy adnotacją @Mock. Dzięki temu raz zadeklarowana imitacja może być wykorzystana w kilku testach.

Druga nowinka w naszym teście to linia:

```
Mockito.verify(mockDao).save(
  Mockito.any(User.class));
```

Jest to nic innego jak weryfikacja interakcji, której nam wcześniej brakowało. Określamy, że chcemy sprawdzić, czy zaszła interakcja z instancją mockDao, a dokładniej czy została wywołana metoda save z parametrem będącym instancją klasy User.

Mockito pozwala na budowanie bardziej złożonych weryfikacji, takich jak sprawdzanie dokładnej ilości wywołań czy sprawdzanie wartości przekazanych parametrów (pominąłem to dla jasności przykładu).

Co jeszcze dla mnie może zrobić Mockito?

Mockito to biblioteka posiadająca o wiele więcej ciekawych funkcjonalności. Powyższy prosty przykład miał na celu jedynie przybliżenie jednej z podstawowych funkcjonalności, która dla mnie stała się też najczęściej wykorzystywaną. Inne ciekawe funkcjonalności to:

- wyrzucanie wyjątków z imitacji pod określonymi warunkami
- weryfikacja kolejności wywołań w interakcji
- upewnienie się, że interakcja nigdy nie nastąpiła (użyteczne podczas refaktoringu)
- Śledzenie interakcji z prawdziwymi implementacjami

Dzięki Mockito nasze testy mogą stać się pełniejsze, a życie spokojniejsze :) O ile początkowe próby i wgryzienie się w idee imitacji i weryfikacji może być nie do końca naturalne, o tyle z czasem na pewno każdy zauważy zysk z uzupełnienia swoich testów. W szczególności staje się to wartościowe, gdy pracujemy z kodem, na który nie do końca mamy wpływ.

Alternatywy

Mockito nie jest jedynym rozwiązaniem tego typu. Są też biblioteki takie jak Jmock i EasyMock. Jednak Mockito zdaję się zdobywać ostatnio bardzo dużą popularność. Dodatkowo istnieje również biblioteka PowerMock, która rozszerza Mockito (oraz EasyMock). Uzupełnia ona Mockito o np. imitowanie wywołań metod statycznych (a jest to niezwykle istotne, gdy nasz kod współpracuje z bibliotekami, których niestety nie możemy zmodyfikować).

Referencje

- http://mockito.org/
- http://code.google.com/p/powermock/
- http://monkeyisland.pl/2008/04/26/asking-and-telling/

Mistrz Programowania: Refaktoryzacja, cz. V

Mariusz Sieraczkiewicz

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

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

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

A książka Mariusza jest warta swej ceny ;)

Grzegorz Duda

Strategia skutecznych programistów: Usuwaj powtórzenia

Powtórzenia w kodzie to źródło wszelkiego zła! Jak drobne nie byłoby to powtórzenie, jest duże prawdopodobieństwo, że prędzej czy później ujawnią się efekty uboczne. Statystyki oparte na moich własnych obserwacjach prowadzą do wniosku — w około 70% przypadków zastosowanie antywzorca *Kopiuj-Wklej* powoduje powstanie trudnych do wykrycia błędów. Dlatego

Ważne

Kiedy widzisz powtórzenie, zastanów się, czy warto zrefaktoryzować kod.

Podobnie jest w klasach `DictPageIterator` oraz `OnetPageIterator` — metoda `dispose`, `next`, `hasNext`, `hasNextLine`, konstruktor, pola klasy są identyczne lub niemal identyczne i prawdopodobnie w kolejnych implementacja również takie będą.

Refaktoryzacja: Wydzielenie klasy abstrakcyjnej

Można pokusić się o refaktoryzację *Wydzielenia klasy abstrakcyjnej* — stworzyć klasę, która będzie zawierać wspólne definicje. Znowu chciałbym zwrócić uwagę na fakt, iż stworzenie klasy abstrakcyjnej nastąpiło, gdyż pojawiła się taka potrzeba w trakcie rozwoju aplikacji. Często spotykam się z przypadkami, kiedy takie klasy tworzone są na w razie czego, „być może w przyszłości się przyda" — niepotrzebnie mnożąc byty. Przykładowy kod znajduje się poniżej.

```
package pl.bnsit.webdictionary;

import java.io.BufferedReader;
import java.io.IOException;
import java.util.Iterator;
import java.util.List;

abstract public class AbstractPageIterator implements PageIterator {

    protected BufferedReader bufferedReader = null;
    protected Iterator<String> wordIterator = null;

    public AbstractPageIterator() {
        super();
    }

    protected void init(String wordToFind) {
```

```
        List<String> words = prepareWordsList(wordToFind);
        wordIterator = words.iterator();
    }

    abstract protected List<String> prepareWordsList(String wordToFind);

    @Override
    public boolean hasNext() {
        return wordIterator.hasNext();
    }

    @Override
    public String next() {
        return wordIterator.next();
    }

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

    protected void dispose() {
        try {
            if (bufferedReader != null ) {
                bufferedReader.close();
            }
        } catch (IOException ex) {
            throw new WebDictionaryException(ex);
        }
    }

}
```

Żeby jednak nie produkować nadmiernej ilości kodu źródłowego w tej książce, nie będę zamieszczał kodu źródłowego klas dziedziczących z `AbstractPageIterator`. Możesz to Czytelniku potraktować jako ćwiczenie.

Najważniejsze odkrycie!

Chciałbym zwrócić twoją uwagę na konstrukcję klasy `AbstractPageIterator`. Wydzielona **metoda `init` realizuje pewien algorytm, którego jednym z kroków jest metoda `prepareWordsList`, ta zaś konkretyzuje się w klasach odziedziczonych z `AbstractPageIterator`**. Jest to nic innego jak realizacja wzorca *Metody szablonu*. Tak! A przecież nic takiego nie planowaliśmy.

Ważne

Konsekwentne stosowanie podstawowych technik refaktoryzacji oraz odpowiedniego rozdzielania odpowiedzialności, prowadzi do wzorców projektowych!

To odkrycie było jednym z najważniejszym przesunięć paradygmatu[1] w moim życiu programisty. Dotarło do mnie, że duża część wzorców projektowych objawia się światu, jeśli stosuje się proste zasady refaktoryzacji i wydzielania odpowiedzialności. Wszystko nagle stało się proste, a zasady programowania obiektowego nabrały nowego sensu.

Z tą myślą chciałbym cię Czytelniku pozostawić w tym miejscu. Temat refaktoryzacji, wzorców projektowych, testowania, to wspaniałe tematy, które można eksplorować całe życie. Przez kilkadziesiąt stron tej książki przebyliśmy drogę poprzez krainę refaktoryzacji. Jeśli chcesz eksplorować temat głębiej, zachęcam do dalszej lektury książek źródłowych, uczestnictwa w szkoleniach czy treningach związanych z tym tematem . . . i używania w praktyce.

Jeśli raz zarazisz się chorobą zwaną refaktoryzacją, nie ma już później odwrotu. Gwarantuję ci, że jest to jedna z najwspanialszych zmian, jaką możesz wprowadzić do sposobu programowania.

Mistrzostwo . . . zobacz co się zmieniło

Refaktoryzacja to jedna z technik mistrzów — najlepszych programistów. Im częściej dokonujesz refaktoryzacji, tym staje się to łatwiejsze, tak iż **w locie** rozpoznajesz sytuacje, w których możesz stworzyć zrefaktorowane rozwiązanie.

Żeby odnaleźć różnicę, porównaj implementację końcową z tej książki z początkową. W zasadzie obie wersje robią to samo, w prawie taki sam sposób. Jednak prawie czyni wielką różnicę.

Ta książka przedstawiła najbardziej użyteczne techniki refaktoryzacji. Owe 20% które najczęściej się przydaje. Jeśli chcesz dowiedzieć się więcej — w ostatnim rozdziale znajdziesz kilka wskazówek, gdzie szukać dalej.

[1] Stephen Covey, 7 nawyków skutecznego działania

Rozdział 5

Pragmatyzm przede wszystkim

W poprzednim rozdziale wspomniałem, nie przez przypadek, że stosowanie refaktoryzacji jest bardzo zaraźliwe, ale też niesie niebezpieczeństwo negatywnych skutków jej nieodpowiedniego stosowania.
Szczególnie na początku fascynacja refaktoryzacją jest bardzo niebezpieczna, choć niezwykle przyjemna. Najchętniej refaktoryzację robiłoby się na każdym kroku, dążąc do tego, aby program był idealny. Jednak nie o to chodzi.

Ważne

Tworzenie oprogramowania polega na tworzeniu najprostszego możliwego kodu, które realizuje założone wymagania.

Ważne

Przedstawione w tej książce przykłady miały na celu pokazać sposób myślenia towarzyszący procesowi refaktoryzacji. Najważniejszy jest kontekst, w jakim powstaje dane rozwiązanie i to on jest bazą do tego, aby podjąć decyzję, czy należy zastosować daną refaktoryzację czy nie.

Dlaczego refaktoryzacja nie jest dobra na wszystko

Tworzenie oprogramowania jest częścią biznesu, który z kolei służy przede wszystkim wytwarzaniu wartości w sposób efektywny. Dlatego chociażby z tego punktu widzenia, refaktoryzacji należy używać z rozwagą. Jeśli będziemy stosować ją nieodpowiednio,

w pewnym momencie staniemy się całkowicie nieefektywni. Jednak jedno nie ulega wątpliwości — refaktoryzować trzeba.

Dziesięć przykazań dotyczących refaktoryzacji

Oto dziesięć przykazań dotyczących refaktoryzacji:[1]:

1. Jeśli kod już istnieje, refaktoryzuj, gdy dany fragment przynajmniej dwa, a najlepiej trzy razy sprawił ci kłopot, bo był źle napisany.
2. Jeśli kod już istnieje, refaktoryzuj, gdy dany fragment często ulega zmianom.
3. Jeśli kod piszesz po raz pierwszy, staraj się na bieżąco eliminować zapachy kodu.
4. Naucz się refaktoryzować w locie — gdy nabierzesz wprawy, wiele refaktoryzacji odbędzie się w głowie.
5. Refaktoryzuj ewolucyjnie, a nie rewolucyjnie — stosuj metodę *Małych kroków*, wprowadzaj jak najmniejsze zmiany.
6. Refaktoryzuj regularnie — tylko wtedy efekty prawa wzrostu entropii nie zdominują cię.
7. Refaktoryzuj wtedy, gdy masz napisane testy lub jeśli refaktoryzacja jest automatycznie kontrolowana przez środowisko programistyczne.
8. Przede wszystkim stosuj najprostsze refaktoryzacje: wydzielanie odpowiedzialności (wydzielanie klasy, metody lub pola), zmiana nazwy, nazywanie warunków i dekompozycja algorytmu na składowe.
9. Używaj refaktoryzacji z rozwagą — jeśli widząc kod pierwszy raz na oczy, chcesz go natychmiast refaktoryzować, bez względu na to, czy będziesz się nim zajmował czy nie, to oznacza, że jesteś w poważnych kłopotach.
10. Ciesz się tym co robisz — dzięki refaktoryzacji, programowanie jest jeszcze przyjemniejsze.

[1] brak wiarygodnych danych odnośnie źródła pochodzenia ;-)

Rozdział 6

I co dalej . . . - inne źródła

Refaktoryzacja to rozległy temat, któremu można poświęcić niemal całe życie. Poniżej znajduje się kilka wskazówek, gdzie można dalej szukać informacji dotyczących tej techniki.

Warsztaty

Na stronie http://www.mistrzprogramowania.pl można znaleźć warsztaty wideo, przedstawiające techniki refaktoryzacji w formie przystępnych przykładów, przy wykorzystaniu środowiska Eclipse. Jeśli chcesz zobaczyć jak w praktyce **przebiega** refaktoryzacja, jest to miejsce, które musisz odwiedzić.

Szkolenia

Moim zdaniem najlepiej napisana książka, czy materiał wideo nigdy nie da tego, co bezpośredni kontakt z doświadczonym trenerem, który przeprowadzi przez niuanse technik refaktoryzacji i innych technik obiektowości. A co najważniejsze, trener odpowie na twoje pytania. Jeśli szukasz sposobu na to, aby jak najszybciej i jak najlepiej się nauczyć opisywanych technik, zapraszamy na szkolenia. Więcej na http://www.bnsit.pl.

Trening indywidualny

Jest formą zdobywania umiejętności, która umożliwia pełne dostosowanie sposobu nabywania kompetencji. Trening to indywidualne spotkania online, dzięki którym razem

z trenerem poznasz dokładnie to, czego potrzebujesz lub będziesz potrzebował w realizowanych projektach. Więcej na http://www.bnsit.pl

Książki

Zdecydowanie dwa najlepsze, bezdyskusyjne źródła zaawansowanej wiedzy:

Refaktoryzacja. Ulepszanie struktury istniejącego kodu, Martin Fowler i inni, WNT 2006

Refaktoryzacja do wzorców projektowych, Joshua Kerievsky, Helion 2005

Inne źródła w sieci

Oczywiście Google twoim zbawieniem. Hasło: refactoring, refaktoryzacja, refaktoring.

Ponadto:

http://www.refactoring.com/ — blog utrzymywany przez Martina Fowlera

http://mbartyzel.blogspot.com/ — blog Michała Bartyzela

http://msieraczkiewicz.blogspot.com/ — mój własny blog

Zaglądaj na jdn.pl http://jdn.pl/ — główny portal poświęcony językowi Java w języku polskim, czytaj Java exPress http://dworld.pl/java-express/ i zaglądaj na blogi polskich bloggerów. Polecamy szczególnie blog Jacka Laskowskiego http://www.jaceklaskowski.pl/. W końcu — wiedza to najcenniejsza inwestycja.