Numer 2/2009(4)

Java eXPress

CZASOPISMO DLA DEWELOPERÓW JAVA



Wprowadzenie do Grails

TeamCity : pre-tested commit

Rozwarstwienie

Obsługa XML w Javie – biblioteka Xstream

Graphical Modelling Framework

Recenzja GroovyMag 04/2009

>> Express killers cz. III

>> J2ME: Serializacja obiektów, cz. II

>> Problemy w aplikacjach internetowych J2EE

Patroni:
e-point
Lider biznesowych zastosowań technologii Java
ISOLUTION.PL

Maszynista



i18n

Tak, tak. Przyszła pora pokazać całemu światu co wspólnie udało nam się stworzyć. Czy to dobry krok? Zobaczymy. W każdym razie ten numer, który właśnie czytacie, będzie pierwszym numerem JAVA exPress wydanym także w wersji angielskiej. Nie oznacza to, że przechodzimy w 100% na wersję angielską. JAVA exPress nadal będzie wydawane także w wersji polskiej.

Dziękuję wszystkim autorom i korektorom za ich wkład w powstanie czasopisma. Dziękuję też Jakubowi Sosińskiemu z firmy Vriltek, za oprawe graficzną i projekt okładki. Dzięki niemu JAVA exPress wygląda coraz bardziej profesjonalnie. Dzięki także Markowi Podsiadłemu, który niestrudzony tworzy nową wersję stron www JAVA exPress i dWorld. No i na koniec nasz nowy kolega - Paweł Cegła. To głównie dzięki niemu będzie możliwe wydanie wersji angielskiej.

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

Do zobaczenia 1 września... Mam nadzieję, że już bez JavaOne'owego opóźnienia ;)

Pozdrawiam,
Grzegorz Duda

Plan podróży

Maszynista: i18n	1
Megafon: GeeCON 2009	2
Megafon: COOLuary v.2	4
Dworzec Główny: Wprowadzenie do Grails	5
Dworzec Główny: Problemy w dużych aplikacjach JEE	8
Bocznica: GMF, czyli graficzny edytor w kilka chwill	17
Bocznica: J2ME: Serializacja obiektów, cz. II	27
Bocznica: Obsługa XML w Javie - biblioteka XStream	38
Maszynownia: TeamCity: pre-tested commit	42
Rozjazd: Express killers, cz. III	46
Więcej węgla: Recenzja GroovyMag kwiecień 2009	47
Konduktor: Rozwarstwienie	50

Skład tekstu i wybór tematów: Grzegorz Duda
Korekta: Agnieszka Wadowska-Duda
Grafika: Jakub Sosiński (Vriltek)
Artykuły html i aplikacje: Marek Podsiadły
Tłumaczenia: Paweł Cegła
kontakt: kontakt@dworld.pl

GeeCON 2009

Jakub Dżon

Narodziny

Pomysł realizacji międzynarodowej konferencji poświęcone językowi Java zrodził się w głowach organizatorów przeszło rok temu, kiedy brak tego typu wydarzenia zaczął bardziej im doskwierać. Pomysł był prosty – zorganizować dużą konferencję, podczas której prelegentami będą osoby znane w międzynarodowym półświatku Javowym, a uczestnikami osoby z Europy Środkowej. Dodatkową cechą konferencji miała być jej „mobilność"; co roku (tak, w założeniu miała być cykliczna) miała się odbywać w innym mieście Środkowej Europy. Po wielu żmudnych ustaleniach i przygotowaniach została ustalona data i miejsce pierwszej edycji GeeCON – Kraków - Multikino, 7 – 8 maja.

Zaproszenia do wygłoszenia prelekcji podczas konferencji zostały wysłane do wielu osób, spośród których część zgodziła się na przyjazd. Nieoficjalnym mottem konferencji jest „społeczność dla społeczności", więc ogłoszone zostało również Call for Papers, dzięki któremu udało się nam zaprosić do współpracy kolejne osoby z ciekawymi tematami.

Dzień I



Pierwszy dzień konferencji rozpoczął się (dla uczestników) już o godzinie 8:30 rejestracją i śniadaniem. Wykładem otwierającym konferencję była prezentacja Simona Rittera pod tytułem *„JavaFX: The Platform for Rich Internet Applications"*, po którym to konferencja potoczyła się dwutorowo. Pozostałymi prelegentami tego dnia byli Alef Arendsen, Corneliu Vasile Creanga, Miško Hevery, Waldemar Kot, Luc Duponcheel, Jacek Laskowski, Václav Pech, Szczepan Faber, Piotr Walczyszyn i Hans Docter.



W połowie dnia uczestnicy mieli zapewniony ciepły lunch, na temat którego jakości i ilości opinie są bardzo podzielone. W mojej opinii lunch był dobry;).

Dzień pierwszy zakończył się zgodnie z planem o godzinie 18.

Dzień II

Cały drugi dzień konferencji przebiegał dwoma równoległymi ścieżkami. Z przyczyn osobistych nie mógł tego dnia przyjechać zaproszony Michael Hüttermann. Zamiast wykładu Michaela odbyła się druga prezentacja Antonio Goncalvesa; tematem był serwer Glassfish.

Pozostałymi występującymi tego dnia byli Arjen Poutsma, Adam Bien, Stephan Janssen, Giorgio Natili, Paweł Wrzeszcz, Tomasz Kaczanowski, Thomas Enebo, Bruno Bossola, Lubomir Petrik i Jakub Podlesak.

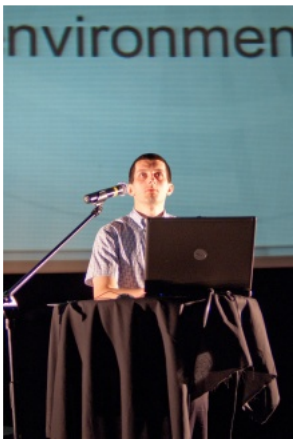
Ostatni dzień konferencji w planach miał być krótszy od pierwszego, ale za to bardziej emocjonujący – na jego koniec, pośród uczestników którzy wypełnili ankiety satysfakcji, rozlosowane zostały prezenty (netbook, licencje na IntelliJ Idea, książki oraz wejściówki na konferencje JDD 2009 i Confidence 2009).

Wydarzenia towarzyszące

Oprócz uczestnictwa w wykładach goście konferencji mogli wziąć udział w warsztatach, które w jej przeddzień zostały przeprowadzone na terenie Akademii Górniczo-Hutniczej. Dodatkowe szkolenia zostały przygotowane przez

> *...the greatest Java conference in Poland, very good speakers*

Sun Learning Services i odbywały się w czterech równoległych ścieżkach w godzinach 11:00 – 18:00.



W trakcie konferencji uczestnicy mogli wziąć również udział w minikonkursie wiedzy o Javie na stanowisku wysłanników serwisu JavaBlackBelt. Nagrodami w konkursie były książki dostarczone przez wydawnictwo Helion.

Firma Google także dostarczyła gościom konferencji dodatkowej rozrywki w postaci losowania firmowych upominków między osobami, które wypełniły rozdawane wcześniej kupony.

Wieczorem 7 maja uczestnicy mieli również zapewnioną rozrywkę – mogli wziąć udział w organizowanej przez Centrum Certyfikacyjne Compedium *„Piwnej Ścieżce Certyfikacyjnej”*, podczas której ich zadaniem było odwiedzenie czterech krakowskich barów, wypicie w każdym z nich darmowego piwa i zebranie czterech pieczątek tworzących napis JAVA. Nagrodami za zebranie całego napisu były upominki od Sun Microsystems.

Opinie uczestników

Poniżej pozwolę sobie zacytować kilka opinii pochodzących z ankiet satysfakcji wypełnianych przez uczestników konferencji.

"It's really good idea to have Java conference in central Europe",
"Amazing work!"
"Lots of valuable information for everybody"
"No failures. Everything was fine!"
"...the greatest Java conference in Poland, very good speakers"



"In general, you did great job!"
"You brought really great speakers"
"... in the fact whole conference is very good and I'm glad to be here, hope to see you next year!"
"I'm impressed what organizator had made. Great job!"
"Most speakers were good (fresh subjects, well-prepared) or VERY GOOD! In general: Good work!"

Winni

Za pomysł i realizację konferencji GeeCON 2009 odpowiedzialni są członkowie grup użytkowników języka Java z Polski i Czech oraz stowarzyszenie GiK; Adrian Nowak, Radosław Holewa, Jakub Dżon, Grzegorz Duda, Andrzej Grzesik, Marcin Gadamer, Miroslav Kopecky, Adam Dudczak i Adam Parchimowicz.



COOLuary

2 otwarte dyskusje o Javie - Gdansk, 27-28 czerwca 2009

COOLuary jest pierwszą w Polsce konferencją o Javie realizowaną w technologii Open Space (http://en.wikipedia.org/wiki/Open_Space_Technology). Wiele osób wracających z najlepszych konferencji o Javie niezaprzeczalnie twierdzą, że najlepsze na konferencji były… przerwy i możliwość dyskusji z innymi programistami w kuluarach. Konferencja COOLuary w ponad 70% to rozmowy kuluarowe znane z przerw na zwykłych konferencjach.



Podczas tej edycji, oprócz standardowej UnConference, zostaną przeprowadzone warsztaty, podczas których będziesz mógł zobaczyć na żywo nowe technologie i warsztat programistyczny swoich kolegów. Zostały zaplanowane warsztaty dotyczące tematów takich jak: Java EE, Groovy i Grails, Scala, OSGi, refaktoryzacja oraz Randori Session.

Dodatkowo nasz główny sponsor, e-Point, wygłosi wykład "Nie samą Javą deweloper żyje".

Wśród uczestników zostaną także rozlosowane książki ufundowane przez wydawnictwo Helion oraz gadżety Javowe.



Jeśli jesteś zwolennikiem JSF lub GWT, to będziesz mógł stać się posiadaczem jednej z wielu Refcardz na ten temat.

Dodatkowo każdy z uczestników zarejestrowanych na obydwa dni otrzyma zniżkę 15% na konferencje Java Developers' Day.

Mam nadzieję, że to nie koniec dobrych wiadomości i tym razem nie zabraknie Cię na COOLuarach.

Zarezerwowaliśmy też tanie noclegi. Jeśli jesteś nimi zainteresowany, to ważna informacja:

Noclegi można rezerwować tylko do 12 czerwca.

Wprowadzenie do Grails

Mateusz Mrozewski

W niniejszym artykule chciałbym przedstawić framework Grails na praktycznym przykładzie bazującym na mechanizmie generowania aplikacji na podstawie modelu (*scaffolding*). Krok po kroku skonfigurujemy środowisko do pracy i uruchomimy naszą pierwszą aplikację. Będzie to stan wyjściowy do dalszej pracy z Grails.

Czym jest Grails

Grails to framework oparty na języku Groovy. Wprowadza on w życie zasadę „konwencji ponad konfigurację". Dynamiczność Groovy wraz z łatwą integracją z rozwiązaniami Java (w końcu korzystamy z tej samej maszyny wirtualnej) pozwalają nam na naprawdę szybkie tworzenie aplikacji i to nie tylko prototypowej, ale w pełni działającej. Dokładając do tego DSL (*Domain Specific Language*) otrzymujemy doskonały zestaw do pracy. Warto wspomnieć też, że Grails to nie wymyślanie koła od nowa. O ile można zauważyć szereg podobieństw do innych frameworków (nie tylko Java), o tyle pod maską Grails znajdziemy powszechnie stosowane rozwiązania: Spring, Hibernate, Jetty, HSQLDB, SiteMesh. Krzywa uczenia (ang. *Learning Curve*) jest w tym wypadku bardzo korzystna dla praktycznie każdego programisty Java, niezależnie od doświadczenia i umiejętności. Sprawdźmy to w praktyce, do dzieła.

Pierwsza aplikacja

Aby rozpocząć pracę z Grails musimy najpierw poświęcić chwilkę na przygotowanie środowiska. I tutaj miłe zaskoczenie, ten proces naprawdę zajmuje chwilkę. Co potrzebujemy? Na początek proponuję wykorzystać Netbeans 6.5 oraz Grails 1.1. Taki zestaw da nam praktycznie gotowe środowisko do pracy bez potrzeby dodatkowych instalacji i konfiguracji. Pobieramy NetBeans 6.5 (*http://netbeans.org*) oraz Grails 1.1 (*http://grails.org*). NetBeans posiada domyślnie możliwość tworzenia projektów Grails. Natomiast sama paczka z Grails, oprócz framework'u i Groovy, zawiera również serwer Jetty, bazę danych HSQLDB oraz wszelkie potrzebne biblioteki (m. in. Hibernate, Spring, SiteMesh). Po zainstalowaniu NetBeans i rozpakowaniu Grails w dowolnej lokalizacji jedyne co musimy zrobić to wskazać w NetBeans gdzie znajduje się Grails. W menu wybieramy `Tools -> Options`, zakładka `Miscellaneous -> Groovy`, a tam w pole `Grails Home` wpisać odpowiednią lokalizację.

Po przygotowaniu środowiska czas utworzyć nasz pierwszy projekt. Wybieramy z menu `File -> New Project ...`, a następnie `Groovy -> Grails Application`. Jako nazwę projektu podajemy `AddressBook` (zrobimy sobie małą bazę adresów). Klikamy `Finish` i gotowe. Nasz pierwszy projekt Grails czeka na uruchomienie. W tym momencie możemy już uruchomić nasz projekt (przycisk *Play* lub `[F6]`). Zobaczymy ekran powitalny, który pozwoli się nam upewnić, że wszystko zrobiliśmy dobrze.

Przejdziemy teraz do implementacji naszej książki adresowej. Na początek zajmiemy się utworzeniem modelu. W Grails klasy modelu nazywane są klasami domenowymi. Klasa domenowa to obiektowa reprezentacja naszego bytu z bazy da-



> *Grails to framework oparty na języku Groovy.*

nych. Z menu kontekstowego projektu wybieramy `New -> Grails Domain Class` ..., podajemy nazwę *Contact* i zatwierdzamy przyciskiem `Finish`. W folderze *grails-app/domain* została utworzona klasa *Contact.groovy*. W niej możemy zdefiniować właściwości charakteryzujące nasz kontakt z książki adresowej. Dla uproszczenia przykładu zdefiniujemy 4 właściwości: `name`, `phone`, `email` oraz `address`. Nasza klasa powinna wyglądać jak na listingu.

```
class Contact {
  String name
  String phone
  String email
  String address

  static constraints = {
  }
}
```

Ponieważ Grails wykorzystuje wzorzec MVC, do stworzenia działającej aplikacji potrzebujemy stworzyć jeszcze kontroler, który będzie obsługiwał naszą aplikację. W tym celu z menu kontekstowego projektu wybieramy `New -> Grails Controller` ..., podajemy *Contact* jako nazwę i zatwierdzamy przyciskiem Finish. W katalogu *grails-app/controllers* została utworzona klasa *ContactController.groovy* (na pewno zauważyliście już, że NetBeans ładnie prezentuje te foldery jako „*Controllers*" i „*Domain Classes*" dla klas domenowych). Utworzony kontroler posiada domyślnie akcję index, jednak na ten moment będziemy chcieli skorzystać z dobrodziejstwa Grails jakim jest *scaffolding*.

Scaffolding można przetłumaczyć jako rusztowanie. Opcja ta pozwala nam na wygenerowanie interfejsu użytkownika i logiki do wykonywania podstawowych operacji CRUD na naszych danych (Create, Read, Update, Delete). W tym celu potrzebujemy zmodyfikować nasz kontroler, aby wyglądał tak jak na listingu.

```
class ContactController {
  def scaffold = true
}
```

Teraz jesteśmy gotowi do ponownego uruchomienia naszej aplikacji (przycisk *Play* lub `[F6]`). Tym razem powinniśmy zobaczyć stronę powitalną z wypisanymi wszystkimi istniejącymi kontrolerami. W naszym wypadku będzie to tylko *ContactController*. Po wybraniu kontrolera powinniśmy zobaczyć pustą listę kontaktów, a także ikonkę pozwalającą nam na dodanie nowego kontaktu. Śmiało możemy to wypróbować. Nasza aplikacja nie należy jeszcze do najbardziej rozbudowanych, ale jest już w pełni funkcjonalna. To co rzuca się od razu w oczy, to brak walidacji danych. Możemy tutaj skorzystać z wbudowanego mechanizmu walidacji danych, który doskonale pracuje z mechanizmem *scaffolding*. Do deklaracji ograniczeń na danych wykorzystamy sekcję *constraints* w naszej klasie domenowej. Służy ona do określenia nie tylko jakie wartości mogą być przyjmowane przez poszczególne właściwości, ale również do określenia ich kolejności w wygenerowanej aplikacji. Nasza sekcja *constraints* powinna wyglądać jak na listingu.

```
static constraints = {
  name(blank:false)
  phone(matches:"^\\+*\\d+")
  email(email:true)
}
```

Kolejność właściwości w tej sekcji zostanie odwzorowana na ekranach wyświetlania szczegółów rekordu, na liście, a także w formularzach dodawania i edycji. Dla właściwości *name* określiliśmy, że nie może być ona pusta. Należy zapamiętać, że *blank* stosowane jest dla właściwości tekstowych i określa, czy dana właściwość może być pustym łańcuchem. Istnieje też ograniczenie *nullable*, jednak określa ono czy dana właściwość może przyjąć wartość *null*, tak więc może być stosowana nie tylko dla łańcuchów tekstowych. Domyślnie Grails nie pozwala żadnej właściwości na przyjęcie wartości *null*. W przypadku właściwości

> *...pod maską Grails znajdziemy powszechnie stosowane rozwiązania: Spring, Hibernate, Jetty...*

phone do walidacji wykorzystujemy wyrażenie regularne (opcjonalny plus na początku „^\\+*” a następnie jedna lub więcej cyfr „\d+”). Dla pola *email* wykorzystujemy wbudowaną walidację adresów email. Zwykłe adresy pozostawiamy tak jak są, dlatego nie umieszczanych ich w sekcji contraints (chyba, że chcielibyśmy określić ich kolejność inaczej niż domyślna). Po ponownym uruchomieniu aplikacji możemy przetestować naszą walidację. Komunikaty błędów walidacji nie będą może najczytelniejsze na świecie, w szczególności dla numeru telefonu, ale walidacja będzie działała poprawnie.

Gdzie dalej?

Grails i Groovy stały się ostatnio niezwykle modne, więc wszyscy zainteresowani na pewno nie będą mieli problemu ze znalezieniem ciekawej lektury, ciekawych eksperymentów innych programistów, a także wielu wskazówek i dodatków. Poniżej krótka lista odnośników, która może posłużyć jako dobry start:

Oficjalna strona Grails – *http://grails.org*

Oficjalna strona Groovy - http://groovy.codehaus.org/

Notatnik Jacka Laskowskiego – *http://jacek-laskowski.pl*

Stronice Chlebika - *http://chlebik.wordpress.com/*

Wszystkich chętnych do dyskusji zapraszam na swojego bloga, http://tech.mrozewski.pl, gdzie można znaleźć trochę informacji na temat Grails, a także bezpośredni kontakt do mnie.



Problemy w dużych aplikacjach J2EE

Jarosław Błąd

Problemy w dużych aplikacjach internetowych Java 2 Enterprise Edition (J2EE) możemy zasadniczo podzielić na dwie kategorie. Pierwszą stanowią problemy związane z implementacją systemów, drugą – problemy występujące w trakcie eksploatacji i utrzymania systemów. Ponieważ na temat implementacji dużych systemów wiele już napisano i powiedziano, w tym artykule skupię się wyłącznie na drugiej grupie problemów, którym w literaturze poświęca się zdecydowanie mniej uwagi. Swoje rozważania będę opierać na doświadczeniach firmy e-point SA w rozwiązywaniu problemów utrzymaniowych i eksploatacyjnych w dużych aplikacjach internetowych.

Na początek spróbujmy zdefiniować, **czym jest „duży system”**.

Podstawowym kryterium jest **wolumen użytkowników**, **transakcji** i **danych**. Jeśli jeden z tych elementów jest duży, wówczas możemy już mówić o dużym systemie. Zazwyczaj, choć nie jest to warunek konieczny, w dużych systemach występuje również pewien poziom komplikacji logiki systemu, czyli tzw. **logika biznesowa**. Niektórzy postrzegają wielkość systemu przez pryzmat funkcji, jakie dostarcza on użytkownikowi końcowemu. Nie sądzę jednak, aby to podejście było słuszne. Świadczy o tym chociażby przykład wyszukiwarki Google, której interfejs oferuje użytkownikom niewiele funkcjonalności, natomiast z całą pewnością nie można o niej powiedzieć, że jest małym systemem. I wreszcie: duży system to taki, który jest krytyczny biznesowo dla klienta, czyli jego awarie i/lub błędne działanie powodują wymierne straty finansowe.

Zastanówmy się teraz, **w jakich obszarach mogą wystąpić problemy w dużych systemach.**

Po pierwsze, mogą mieć one źródło w kodzie **aplikacji**, który napisaliśmy. Drugi typ problemów dotyczy **oprogramowania aplikacyjnego**, z którego nasza aplikacja musi korzystać. Mowa tu o problemach występujących w serwerach http, serwerach aplikacyjnych, bazie danych, systemach kolejkowania czy systemach zewnętrznych, z którymi komunikuje się nasz system. Kolejne obszary, na których mogą wystąpić problemy, to **system operacyjny i protokoły sieciowe**, gdzie również czai się wiele niespodzianek. I oczywiście jest jeszcze **sprzęt**, na którym to wszystko działa, a który również może być źródłem problemów.

Liczba błędów spada zgodnie z przedstawianą hierarchią elementów systemu – w aplikacji jest ich najwięcej, w sprzęcie najmniej. Natomiast im bliżej sprzętu znajduje się problem, tym jest on bardziej poważny i trudniejszy do rozwiązania.

Skoro już wiemy, gdzie mogą wystąpić problemy, zobaczmy, **co może nas zaboleć**.

Pierwsza rzecz to **stabilność systemu**. Użytkownicy oczekują, że w długim okresie czasu system będzie realizował pewne funkcje biznesowe, przynosząc im określone korzyści. System, który pracuje niestabilnie, automatycznie powoduje nieufność użytkowników końcowych, a tym samym jest powodem utraty wiarygodności przez właściciela systemu.

Kolejna sprawa to **wydajność**. Jest ona inaczej postrzegana od strony użytkownika końcowego niż od strony zamawiającego system. Użytkownik końcowy w większym stopniu oczekuje szybkiego czasu odpowiedzi, czyli czegoś, co określa się mianem *performance*. Natomiast klient zamawiający system będzie oczekiwać głównie przepustowości (*throughput*), czyli jak największej liczby transakcji biznesowych zrealizowanych w jednostce czasu. Proponuję za-



Jarosław Błąd
Dyrektor ds. Realizacji w e-point SA

Zarządza pionem technicznym, organizuje środowisko pracy dla programistów, web deweloperów i administratorów. Odpowiada za proces implementacji systemów informatycznych, a następnie za ich techniczne utrzymanie.
Zainteresowany przede wszystkim procesami implementacji systemów informatycznych, zarządzaniem zespołem deweloperów i relacyjnymi bazami danych.

> *Liczba błędów spada zgodnie z przedstawianą hierarchią elementów systemu – w aplikacji jest ich najwięcej, w sprzęcie najmniej.*

tem przyjąć, że wydajność jest to pewna przepustowość systemu przy akceptowalnym czasie odpowiedzi.

Jest również coś takiego jak **dostępność systemu**, określająca przez jaki czas system jest dostępny dla użytkowników. Jest ona związana w dużej mierze ze stabilnością i wydajnością.

I jeszcze dwa dodatkowe aspekty: **bezpieczeństwo**, którego znaczenie jest oczywiste, oraz **administrowanie systemem**. Zdarzają się systemy zbudowane w sposób, który znacząco utrudnia administrowanie. Przy pewnej skali systemu i obciążenia, czy też dużej liczbie serwerów, klastrów itd. zarządzanie takim systemem może być bardzo uciążliwe, a koszty jego utrzymania będą ekstremalnie wysokie.

Jeśli zestawimy te dwa aspekty, tzn. miejsce wystąpienia problemu oraz jego charakter, to otrzymamy całkiem sporą przestrzeń problemów (rysunek 1).

Nie sposób jest w tym stosunkowo krótkim artykule omówić całą pokazaną powyżej przestrzeń problemów. Dlatego przedstawię tu trzy wybrane przykłady, z jakimi zetknęliśmy się w e-point SA.

Zacznę od rzeczy dosyć łagodnych, czyli **problemów ze współbieżnością aplikacji**.

Załóżmy, że mamy system, który po wdrożeniu pracuje stabilnie przez kilka miesięcy. Stale przybywa użytkowników, rośnie też obciążenie poszczególnych elementów systemu, ale pomimo tego wszystko działa raczej bezproblemowo. Do czasu. Dalszy wzrost liczby użytkowników powoduje pojawienie się niepokojących objawów – system zaczyna zamierać na kilkadziesiąt sekund, czasami na kilka minut. Część z tych przypadków kończy się całkowitą „śmiercią" systemu i jesteśmy zmuszeni go restartować. Przy czym, co jest istotne, w trakcie takiego „przymierania" obserwujemy dosyć mocny spadek obciążenia na procesorach, nie widzimy też jakichś intensywnych operacji I/O, czy to na bazie, czy na serwerze aplikacyjnym. I zaczynamy obserwować w serwerze aplikacyjnym – a konkretnie w logach aplikacji – wyjątki z zakleszczeniami (deadlock), czyli informację, którą zwraca nam pośrednio driver JDBC, że baza danych wykryła zakleszczenie transakcji, którą właśnie wykonywaliśmy, po czym ta transakcja została wycofana. Pojawiają się również wyjątki pokazujące, że transakcje JTA w serwerze aplikacyjnym timeout'ują. Analiza sytuacji nie była prosta, ale w końcu odkryliśmy, jakie były przyczyny tego

	Aplikacja	Oprogramowanie aplikacyjne	System operacyjny / protokoły sieciowe	Sprzęt
Stabilność	BUM!		BUM!	
Wydajność		BUM!		
Dostępność			BUM!	
Bezpieczeństwo	BUM!			
Administracja				BUM!

Rysunek 1. Przestrzeń problemów

> *Zdarzają się systemy zbudowane w sposób, który znacząco utrudnia administrowanie.*

stanu rzeczy.

Po pierwsze, okazało się, że w dwóch konkretnych transakcjach biznesowych mieliśmy niekorzystny przeplot odczytów i zapisów. Chodziło o operacje na bazie danych. Już samo to mogło prowadzić do deadlocków, ale akurat w naszym przypadku musiało być coś jeszcze. I tym czymś było pełne skanowanie (full scan) jednej z tabel w bazie danych. Była to tabela, która występowała w obu transakcjach. Kiedy my żądaliśmy odczytu jednego rekordu, optymalizator uznawał, że szybciej będzie przeskanować całą tabelę. Z logicznego punktu widzenia działanie optymalizatora było poprawne, natomiast z punktu widzenia wydajności powodowało drastyczne obniżenie współbieżności, przez co kluczowe transakcję zaczynały się ze sobą blokować.
Kiedy już doszliśmy do tego, co powodowało problem, wówczas jego rozwiązanie nie było zbyt pracochłonne. Włącznie ze zmianą aplikacji i wgraniem na produkcję zajęło nam kilka godzin. Natomiast sama analiza i dojście do przyczyn problemu trwało kilka dni.

Rozwiązanie było proste. Operacje modyfikujące przenieśliśmy na koniec tych transakcji, co akurat było możliwe z punktu widzenia wymagań biznesowych. Następnie przygotowaliśmy specjalną podpowiedź dla optymalizatora bazy danych, która spowodowała, że preferowanym sposobem dostępu do tabeli było użycie indeksu, a nie pełne przeglądanie jej zawartości.
Podsumujmy teraz problemy ze współbieżnością.

Na uczelni mówiono nam, że jeśli mamy problem z deadlockami, to wystarczy w ustalonej kolejności blokować zasoby i wszystko wróci do normy, w szczególności nie będzie deadlocków. Niestety to rozwiązanie akademickie działa tylko teoretycznie, natomiast jest trudno realizowalne w praktyce, nawet w małych systemach. Dlaczego? Przede wszystkim, w realnym systemie biznesowym mamy do czynienia z tysiącami transakcji, które nawzajem się przeplatają i każda z tych transakcji zajmuje pewne zasoby – zazwyczaj kilka, kilkanaście, a czasami nawet kilkadziesiąt. I, co gorsza, nie mamy praktycznie żadnej kontroli nad blokowaniem tych zasobów, ponieważ zazwyczaj operujemy na bazie danych. To baza danych decyduje w dużej mierze o tym, które zasoby zablokować. Jeżeli chcemy pobrać jeden wiersz z określonej tabeli, to zazwyczaj oczekujemy, że tylko on zostanie zajęty. Ale niekoniecznie musi to być prawda. Równie dobrze może to być strona tabeli (wiersze w bazie danych grupowane są zazwyczaj w większe jednostki danych) lub nawet cała tabela. Jeżeli nie mamy tej kontroli, to absolutnie nie ma mowy o ustalaniu jakiejś kolejności, bo i tak to nic nie da. Czasami też z wymagań biznesowych dla poszczególnych transakcji wynika, że nie można odwrócić pewnych działań w transakcji, co oczywiście już w pierwszym podejściu rozkłada akademickie podejście na łopatki.

Jeśli jest tak źle, to co możemy zrobić w praktyce? Przede wszystkim możemy testować system za pomocą realnych scenariuszy biznesowych. Chodzi o to, aby po wdrożeniu systemu przyglądać się aktywności użytkowników i na podstawie uzyskanych w ten sposób danych budować scenariusze biznesowe. W ten sposób, przy wprowadzaniu zmian w aplikacji lub rozbudowie systemu, możemy w warunkach laboratoryjnych realizować określone scenariusze, mocno obciążając system i obserwując wszystkie parametry związane ze współbieżnością – czyli to, co dzieje się na bazie danych oraz w serwerze aplikacyjnym. Niezwykle istotne jest również monitorowanie parametrów związanych ze współbieżnością na bieżąco w trakcie działania systemu produkcyjnego.

Kolejny temat to **styk z systemami zewnętrznymi**. Przyjmijmy, że mamy dojrzały system pracujący stabilnie od dwóch lat. Wszystko działa poprawnie, nie ma żadnych problemów wydajnościowych, użytkownicy są zadowoleni. Na życzenie klienta wdrażamy funkcję potwierdzania transakcji SMS-em. Po tym wdrożeniu system pracuje stabilnie przez kilka tygodni i nagle staje się niedostępny dla użytkowników. Po naszej stronie mamy całkowity

> *Rozwiązanie było proste. Operacje modyfikujące przenieśliśmy na koniec transakcji.*

spadek obciążenia na wszystkich elementach systemu. W systemie nie dzieje się nic, jeśli nie liczyć brzegowego serwera http przyjmującego liczne nieudane żądania. Tym, co obserwujemy, jest wysycenie puli wątków w serwerach aplikacyjnych. Wszystkie wątki w serwerze, które mogły przetwarzać żądania, są zajęte. W logach cicho. Co próbujemy zrobić?

Ponieważ na pierwszy rzut oka nic nie można z tego wywnioskować, więc decydujemy się na restart systemu. System wstaje, ale po 2 minutach ponownie pada. Więc restartujemy jeszcze raz i znowu to samo. Pomaga dopiero wyłączenie modułu do wysyłania SMS-ów. System wstaje i działa poprawnie. Wiemy już, gdzie leży problem. Okazało się, że nowa funkcjonalność zepsuła coś w sposób dosyć drastyczny. Po przeanalizowaniu sytuacji odkryliśmy, że bezpośrednią przyczyną problemu była niedostępność bramki SMS-owej. Przy czym bramka nie odrzucała połączenia, tylko po prostu nic nie działo się z wysłanymi przez nas żądaniami, które tam sobie najzwyczajniej w świecie „wisiały". Timeouty TCP na poziomie systemu operacyjnego są bardzo długie, więc w tym czasie nic nie zdążyło się zerwać i nie mieliśmy żadnych wyjątków. Prawdziwa przyczyna problemu leżała gdzie indziej. Wynikała ona z niedostatecznej separacji naszego systemu i bramki SMS. Pewna separacja została przez nas przewidziana, ale okazała się niewystarczająca. W naszym rozwiązaniu, gdy użytkownik wykonywał transakcję, to jej dane trafiały do odpowiedniej struktury w bazie danych, a dodatkowo do oddzielnej tabeli trafiał komunikat SMS, który był następnie pobierany i wysyłany przez całkowicie asynchroniczny proces. Niestety proces ten powodował blokowanie do zapisu tabeli z komunikatami, co w przypadku problemów z bramką SMS skutkowało zawieszeniem pracy całego systemu.
Jakie mamy tutaj rozwiązania?

Tego typu problemy rozwiązuje się zazwyczaj przez wprowadzenie kolejkowania. Może to być np. JMS. Ponadto, obowiązkowe jest wprowadzenie timeoutów przy komunikacji z takimi systemami zewnętrznymi. I to timeoutów na różnych poziomach, nie tylko na poziomie logicznym, ale również na poziomie TCP. My akurat nie mogliśmy zastosować kolejkowania, więc musieliśmy użyć triku z bazą danych. Po prostu zreorganizowaliśmy dostęp do bazy danych w taki sposób, aby odczyt z tabeli komunikatów nie blokował nam żadnych zapisów, które mogą tam trafić.
Styk z systemami zewnętrznymi, z którymi się integrujemy lub współdziałamy, jest najczęstszą przyczyną problemów w dużych systemach. Zawsze na tych interakcjach, gdzie w grę wchodzi sieć, bardzo dużo może się wydarzyć. Kolejne przykłady takich punktów styku to:

- **Baza danych** – bardzo często uważamy, że baza danych jest integralną częścią systemu, ale tak naprawdę – jeśli patrzymy z punktu widzenia aplikacji – jest to taki sam punkt styku, jak w przypadku systemu backend'owego czy bramki SMS. Dlatego też musimy dobrze skonfigurować odpowiednie timeout'y i parametry pól połączeń, aby „nie iskrzyło" na styku między serwerami aplikacyjnymi a bazą danych.
- **Serwery poczty** – które też czasami niedomagają w specyficzny dla siebie sposób.
- **Systemy zewnętrzne** – współcześnie budowane systemy praktycznie nigdy nie istnieją niezależnie od innych systemów zewnętrznych. Zawsze jest z boku jakiś inny – mniejszy lub większy – system, który uczestniczy w pracy naszego systemu.
- **Podsystem logowania zdarzeń** – ciekawostką jest, że on też może dać nam w kość. Znam przykład serwera aplikacyjnego, który logował zdarzenia, a gdy log urósł do rozmiarów 2 GB serwer nagle zniknął, tzn. zniknął JVM i nie dał się później uruchomić. Rozwiązaniem było wyczyszczenie loga, ale żeby dojść do tego rozwiązania, jakiś czas należało mocno pogłówkować.

Wydawałoby się, że **pule wątków** nie wysycają się tak szybko, zwłaszcza jeśli mamy sporej wielkości rozwiązanie typu 10 maszyn wirtualnych po 50 wątków w każdej maszynie. To da-

> *Możemy jeszcze bardziej zoptymalizować już zoptymalizowaną aplikację*

je nam 500 wątków, które tylko czekają, żeby przyjąć ruch z Internetu i obsłużyć użytkowników. Gdy jednak w systemie obsługiwanych jest jednocześnie 100 requestów na sekundę i poszczególne wątki zaczną się nawzajem blokować, to kwestią tylko kilku lub kilkudziesięciu sekund, czasami 2 minut, jest zablokowanie wszystkich wątków, czego skutkiem jest totalna zapaść systemu.

Na koniec rozważań o problemach z systemami zewnętrznymi chciałbym zwrócić uwagę na **warstwę sieci**. Tu niestety bardzo dużo może się wydarzyć. Stosy TCP/IP w systemach operacyjnych, firewall'e, rutery, switch'e, kanały VPN do systemów, z którymi w jakiś sposób się integrujemy... Wszystko to może przestać działać i to, jak zwykle, w najmniej oczekiwanym momencie. Gorsza sprawa, że może tylko udawać, że działa, co w przypadku sieci jest częstym zjawiskiem. Do tego dochodzą awarie typu uszkodzony kabelek, który trochę przepuszcza, ale nie do końca.

Kolejny temat to **pamięć w serwerach aplikacyjnych jako ograniczenie wydajnościowe**. I od razu „mocny" przykład. Jak zwykle zacznę od opisu sielanki: mamy system stabilnie pracujący od kilkunastu miesięcy, liczba użytkowników systematycznie rośnie, system jest trochę bardziej obciążony, ale to niespecjalnie daje się we znaki. Na serwerze aplikacyjnym 70% zużycia procesora. Zamawiający jest zadowolony. Jednak w momentach zwiększonego ruchu zaczynamy obserwować niepokojące objawy. Na początku coraz dłuższy czas odpowiedzi systemu i bardzo często maksymalne obciążenie procesora na serwerach aplikacyjnych, które czasami doprowadzają do załamania systemu. W logach nie obserwujemy niczego wyjątkowego, więc wydaje się, że są to klasyczne objawy zwykłego przeciążenia systemu. Po prostu Java wypala procesor i wypadałoby dołożyć więcej mocy.

Mamy tutaj dwie metody rozwiązania tego klasycznego problemu z wydajnością Javy. Możemy jeszcze bardziej zoptymalizować już zoptymalizowaną aplikację, co wymaga zmian w kodzie i późniejszych testów. Stanowi więc pewne ryzyko. Ale przede wszystkim podnosi późniejsze koszty eksploatacji systemu, o czym nie wszyscy pamiętają. Drugie rozwiązanie to dołożenie sprzętu, co wydaje się lekkie, łatwe i przyjemne. Można to zrobić szybko – postawić kolejną maszynę, zainstalować odpowiednie oprogramowanie, dołączyć maszynę do klastra i oczekiwać pozytywnych efektów.

My wybraliśmy tę drugą drogę. Podłączyliśmy kolejny sprzęt i faktycznie odnotowaliśmy poprawę, ale nie taką, jakiej się spodziewaliśmy, czyli daleką od liniowej. Na nowo przeanalizowaliśmy problem. Skoro Java tak „paliła", to zaczęliśmy bardziej wnikliwie przyglądać się pracy maszyny wirtualnej. I co się okazało? Przyczyną problemu nie była niewystarczająca moc procesora, tylko niewystarczająca ilość pamięci dla aplikacji i w związku z tym zbyt częste uruchamianie procesu odśmiecania pamięci.

Tak się składa, że współczesne aplikacje mają tendencję do alokowania dużej liczby nowych obiektów, a następnie zostawiana ich „na pożarcie" maszynie wirtualnej. Na każdym żądaniu tworzymy bardzo dużo obiektów i potem liczymy na to, że zostaną w elegancki sposób posprzątane. W przypadku maszyny wirtualnej, z którą mieliśmy do czynienia, tak się nie działo i przysłowiowy „garbaty" bardzo często musiał pracować. Tak naprawdę procesor głównie obsługiwał proces odśmiecania, zamiast wykonywać kod aplikacji.

Jakie rozwiązanie zastosowaliśmy? Przede wszystkim znacząco rozbudowaliśmy pamięć fizyczną w serwerach i uruchomiliśmy na każdym z serwerów fizycznych kilka maszyn wirtualnych. Dlaczego tak, a nie inaczej? Dlaczego np. nie zwiększyliśmy pamięci na maszynie wirtualnej? Ponieważ była to maszyna 32-bitowa i po prostu nie dało się tego zrobić. Na tym systemie operacyjnym, z którym mieliśmy do czynienia, mogliśmy pamięć sterty powiększyć tylko do tysiąca sześciuset megabajtów, czyli niecałych 2 GB. Więcej się nie dało, zatem trzeba było to rozwiązać inaczej.
Ponieważ przypadek jest dosyć ciekawy, pod-

> *Dla wydajności całego systemu istotny jest czas pracy poświęcony na proces odśmiecania*

dam go szczegółowej analizie.

Przede wszystkim: co znajduje się na stercie pamięci w serwerze aplikacyjnym, gdzie nasza aplikacja jest zainstalowana? W zasadzie są tam dwa typy obiektów. Obiekty długo żyjące, związane z serwerem aplikacyjnym, z sesją użytkowników, pamięcią podręczną, które podczas procesu odśmiecania pamięci nie podlegają usunięciu z pamięci, albo też dzieje się to bardzo rzadko, np. dla sesji użytkownika. Z drugiej strony, mamy obiekty krótko żyjące, czyli w zasadzie wszystkie te, które są związane z obsługą każdego żądania przesyłanego do serwera. Te obiekty, praktycznie przy każdym uruchomieniu garbage collectora, są usuwane z pamięci systemu.

Dla wydajności całego systemu istotny jest czas pracy poświęcony na proces odśmiecania (zmienna `Garbage Collector time`, GC_{time}) oraz czas, jaki występuje pomiędzy poszczególnymi uruchomieniami garbage collector'a (zmienna `Allocation Failure Distance`, $AF_{distance}$, określająca jak często on się uruchamia).

Jak teraz określić zużycie procesora dla aplikacji i dla garbage collectora? W pierwszym przypadku bierzemy po prostu czas odśmiecania (GC_{time}) i dzielimy go przez sumę czasu odśmiecania i odstępów pomiędzy uruchomieniami garbage collectora ($AF_{distance}$), czyli przez całkowity czas, który jest potrzebny na odśmiecanie pamięci i normalną pracę (patrz: rys. 2).

$$CPU_{GC}\,[\%] = \frac{GC_{time}}{GC_{time} + AF_{distance}}$$

Rysunek 2.

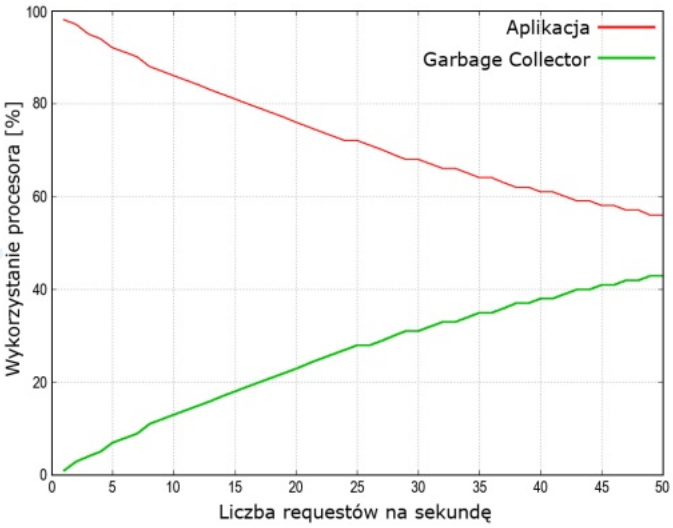
W przypadku aplikacji mamy zależność odwrotną, czyli czas procesora poświęcony na działanie aplikacji ($AF_{distance}$) dzielimy przez czas, w którym nie działa garbage collector (patrz: rys. 3).

$$CPU_{APP}\,[\%] = \frac{AF_{distance}}{GC_{time} + AF_{distance}}$$

Rysunek 3.

Jeśli chodzi o czas odśmiecania, to z dużym przybliżeniem można powiedzieć, że jest on wprost proporcjonalny do wielkości pamięci zajętej przez obiekty długo żyjące. Przynajmniej taką właśnie zależność obserwujemy dla algorytmu *mark and sweep*.

Natomiast odległość pomiędzy poszczególnymi uruchomieniami procesu odśmiecania ($AF_{distance}$) możemy określić odejmując wielkość obiektów długo żyjących (`LongLivedObjectsSize`) od wielkości sterty ustalonej na serwerze (`HeapSize`) i dzieląc wszystko przez iloczyn liczby requestów na sekundę (`RPS`) oraz wielkości obiektów krótko żyjących generowanych przez każdy request (`ShortLivedObjectsSizePerRequest`). Zależność tę ilustruje rys. 4.



Rysunek 4.

Cofnijmy się teraz do przykładu naszej aplikacji, która ma do dyspozycji stertę wielkości około 2GB. Zakładamy, że idą kolejne requesty. Okazuje się, że w naszym przypadku pamięć na obiekty długo żyjące wynosi ok. 700

> *Zobaczmy, czy w praktyce da się zapewnić klientowi te kolejne dziewiątki*

MB, co stanowi mniej więcej 1/3 sterty. Na każdy request przypada nieco ponad 1 MB. Doświadczalnie wyznaczamy, że czas odśmiecania tego 1 MB sterty zajmuje jakieś 2-3 milisekundy. Korzystając z wcześniej przedstawionej zależności możemy wyznaczyć czas wykorzystania procesora na pracę aplikacji i na proces odśmiecania pamięci, co ilustruje rysunek 5.

$$AF_{distance} = \frac{HeapSize - LongLivedObjectsSize}{RPS * ShortLivedObjectsSizePerRequest}$$

Rysunek 5. Czas procesora poświęcony na pracę aplikacji i proces odśmiecania pamięci

Przy 50 requestach proces odśmiecania zajmuje 40% czasu procesora, czyli – delikatnie mówiąc – niewiele nam pozostaje na pracę aplikacji. I teraz cały trik polega na tym, aby maksymalnie spłaszczyć tę krzywą. Jeżeli nie mamy możliwości dokonania żadnego manewru w maszynie wirtualnej (np. zmiany algorytmu odśmiecania lub „pokręcenia" innymi parametrami), to najprostszym sposobem jest dołożenie pamięci i spowodowanie, żeby przestrzeń na obiekty krótko żyjące była dużo większa. W naszym przypadku dokładnie tak zrobiliśmy.

Na koniec artykułu proponuję przyjrzeć się z bliska tematowi **dostępności** czyli mitycznym dziewiątkom…
W niektórych dokumentacjach do serwerów aplikacyjnych opisane są scenariusze, jak osiągać kolejne dziewiątki – pierwszą, drugą, trzecią… piątą… a nawet ósmą! Zobaczmy, czy w praktyce da się zapewnić klientowi te kolejne dziewiątki, a jeśli tak, to jak to zrobić.

Pierwsza dziewiątka – dostępność systemu na poziomie **90%**. Oznacza to 73 godziny niedostępności w miesiącu. Mamy więc dużo czasu, aby coś naprawić, czy wręcz kupić nowy sprzęt w supermarkecie i zainstalować wszystko od nowa lub skorzystać z dowolnego centrum hostingowego. Aby wytworzyć taki system, zespół nie musi posiadać zaawansowanej wiedzy, a z jego obsługą poradzi sobie nawet dochodzący administrator. Bułka z masłem :-)

Zobaczmy, co się stanie, gdy dołożymy drugą dziewiątkę. Otóż **99%** dostępności to nieco ponad 7 godzin w zapasie. Zaczynają się schody. Po pierwsze, sprzęt z supermarketu już nie wystarczy. Niezbędne minimum, jakie musimy zagwarantować, to serwer zapasowy, przygotowany tak, aby w każdej chwili zastąpić ten, który potencjalnie może się zepsuć na produkcji. Nie mamy bowiem czasu, aby wszystko instalować od zera. Kolejna sprawa: należy zadbać o porządne wykonanie pewnych elementów systemu, np. przemyśleć aspekty współbieżności, interakcje z innymi systemami – tak, aby wszystko składało się w jedną całość. I wreszcie: taki poziom dostępności wymaga nadzoru w systemie 24/7. Czy zatem jeden administrator poradzi sobie z tym zadaniem? Powiedzmy, że tak – że są tacy, którzy w pojedynkę daliby radę, ale z pewnością nie obejdą się bez mechanizmów automatycznego monitorowania systemu, które na tym poziomie są już niezbędne. Nadal jednak, mimo pewnych trudności, osiągnięcie dwóch dziewiątek nie jest szczególnie trudne.

Pójdźmy dalej i skomplikujmy sytuację dokładając kolejną dziewiątkę. **99,9%** dostępności daje nam 43 minuty w miesiącu na awarie. I to jest już, moim zdaniem, wyzwanie wyłącznie dla profesjonalistów. Przede wszystkim niezbędna jest pełna redundancja sprzętu, oprogramowania systemowego i aplikacyjnego. To znaczy musimy zapewnić sobie klastry wydajnościowo niezawodne – zarówno w serwerach http, serwerach aplikacyjnych, jak i w serwerach bazy danych. Oczywiście to samo dotyczy sprzętu – musimy mieć zapewnione klastrowane switch'e, firewall'e, macierze dyskowe itd. Potrzebny jest też bardzo staranny projekt i pre-

> *99,99% oznacza mniej niż 5 minut niedostępności w miesiącu, a więc takie trochę mission impossible*

cyzyjna implementacja takiego systemu, ponieważ nie ma tu już miejsca na większą awarię. Kolejna sprawa to analiza stanów przedawaryjnych – system należy przez cały czas monitorować i szybko reagować na potencjalne problemy. Ważna jest bowiem stała prewencja i zapobieganie, a nie tylko gaszenie pożaru już po wystąpieniu awarii. Do tego wszystkiego musimy zapewnić bardzo doświadczone zespoły – deweloperski i hostingowy – które uczestniczyły już w podobnych zadaniach. Muszą one pracować w trybie ciągłym, na bieżąco monitorować system i szybko reagować nawet na najmniejsze symptomy zbliżającej się awarii. No i musimy wyposażyć administratorów w zautomatyzowane procedury naprawcze. 43 minuty to naprawdę niewiele i jeśli weźmiemy pod uwagę stres, który dochodzi w przypadku awarii, to okazuje się, że w tak krótkim czasie człowiek nie jest w stanie zrobić wiele więcej niż nacisnąć guzik stop/start. Przy czym implementacja takiego guzika w dużych systemach jest mocno nietrywialna. Wyłączenie i postawienie od zera systemu, który pracuje na kilkudziesięciu maszynach i składa się ze 100 czy 200 komponentów softwarowych, zajmuje zazwyczaj parę minut, nawet jeśli jest on dobrze wykonany. Wymagane jest również przygotowanie systemu do rekonfiguracji w locie. Dotyczy to zarówno samej aplikacji, jak i komponentów, na których ta aplikacja pracuje. Co jeszcze? Przydałoby się również podwójne monitorowanie, a najlepiej monitorowanie monitoringu. Chociaż, być może, jest to warunek bardziej adekwatny dla kolejnej dziewiątki. A podsumowując temat trzech dziewiątek: z mojego doświadczenia wynika, że to jeszcze da się osiągnąć.

Zatem doszliśmy do czwartej dziewiątki i tu… niespodzianka – nie opiszę, jak ją osiągnąć, ponieważ sam nie potrafię tego zrobić. **99,99%** oznacza mniej niż 5 minut niedostępności w miesiącu, a więc takie trochę *mission impossible*. Niestety klienci czasami tego właśnie oczekują, a najbardziej chcieliby mieć 100% dostępności i to na dodatek za darmo. Takie podejście klientów można jeszcze zrozumieć. Natomiast zaskakujące jest, że dostawcy oprogramowania J2EE obiecują nam nie tylko cztery, ale nawet pięć czy więcej dziewiątek! Przedstawiają w dokumentacji wspaniałe scenariusze, wielkie wykresy, rozbudowane diagramy różnych sprzętów, klastrów i, Bóg wie, czego jeszcze. I obiecują właśnie te osławione pięć dziewiątek czy też więcej – czasami osiem, co dla mnie jest już czystą abstrakcją.
Dlaczego uważam, że w praktyce jest to niewykonalne? Ponieważ nawet najlepsze centra hostingowe dostarczają obecnie usługi z dostępnością 99,95%, czyli zostawiają sobie w zapasie 21 minut na własne awarie. A przecież trzeba jeszcze doliczyć czas na ewentualne awarie po naszej stronie. Widać więc, że tego za bardzo nie da się zrobić, nawet gdyby poszczególne komponenty działały na poziomie dostępności 99,99% czy nieco więcej.

Zastanówmy się teraz, ile może kosztować każda kolejna dziewiątka? Według obliczeń dokonanych za oceanem, każda następna dziewiątka zwiększa koszt wytworzenia systemu o rząd wielkości i podwaja roczny koszt jego utrzymania. Według moich szacunkowych kalkulacji, ta prawidłowość działa co najmniej w przypadku trzech pierwszych dziewiątek. Zatem decyzja o planowaniu dostępności systemu musi być decyzją biznesową, poprzedzoną szczegółowymi wyliczeniami. Każda firma powinna sobie indywidualnie obliczyć, ile może ponieść straty, a ile zaoszczędzić na tym, że system będzie bardziej lub mniej dostępny.

Na sam koniec małe podsumowanie.

Bardzo często spotykamy się z takim twierdzeniem, że TO nie ma prawa się zdarzyć w przypadku naszego systemu, a prawdopodobieństwo, że TO nas spotka, można mierzyć w skali niemalże kosmicznej. Nic bardziej błędnego. Drobne obliczenie. W systemie internetowym średniej skali mamy około 500 odsłon na sekundę, każdy użytkownik „ciągnie” 10 elementów na stronie – powiedzmy flash’e i inne ozdobniki. Mamy 3600 sekund w godzinie, 24 godziny na dobę, 365 dni w roku,

> *Zasadnicze pytanie nie jest więc takie, czy coś się może wydarzyć, ale kiedy to coś się wydarzy.*

a system utrzymujemy przez 5 lat – bo taki jest średni czas życia systemu internetowego. I dochodzimy do wniosku, że mamy prawie 800 miliardów szans na to, aby coś poszło źle. W Drodze Mlecznej, według ostatnich obliczeń, jest 400 miliardów gwiazd, więc skala kosmiczna jest tu jak najbardziej adekwatna. To oznacza, że problemy (takie odpowiedniki wybuchu supernowej) mogą nam się przytrafiać codziennie.

Zasadnicze pytanie nie jest więc takie, czy coś się może wydarzyć, ale kiedy to coś się wydarzy. I trzeba być na to przygotowanym. W dużych systemach, mam nadzieję, że udało mi się to pokazać, mamy do czynienia z naprawdę dużymi problemami, ale jest też ogromna satysfakcja, jeśli te problemy udaje nam się pokonać. Sądzę, że jest sporo osób, szczególnie z dłuższym stażem deweloperskim, które znajdują prawdziwą satysfakcję z rozwiązywania problemów właśnie w takich systemach. Jestem jedną z nich.

Masz pytanie do autora? Napisz: jaroslaw.blad@e-point.pl

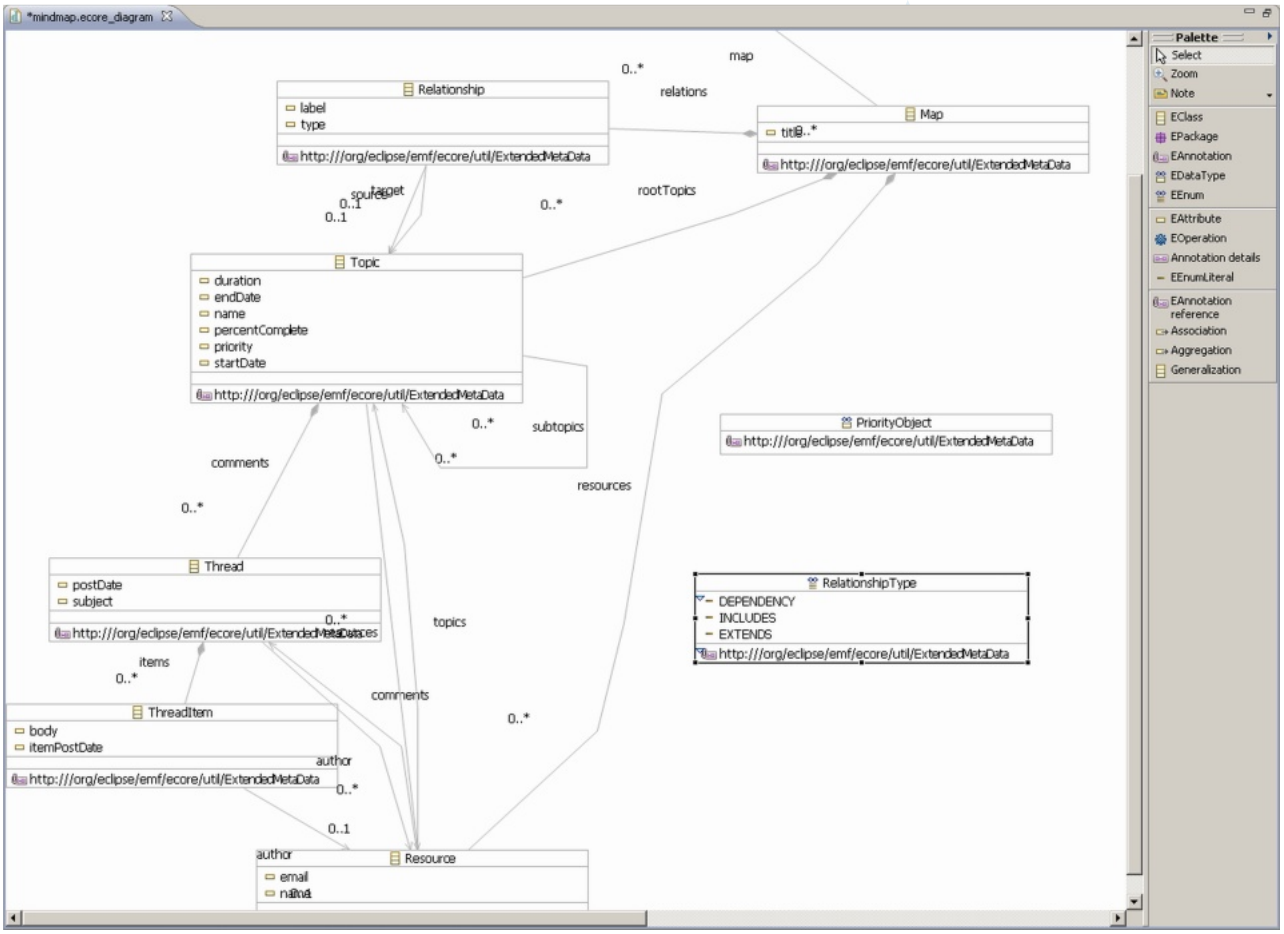
GMF, czyli graficzny edytor w kilka chwil

Jakub Jurkiewicz

Nie od dziś wiadomo, że dobry rysunek potrafi wyrazić więcej niż tysiąc słów, dlatego też tak cenna jest możliwość wizualizacji danych. Do tej pory tworzenie narzędzi umożliwiających w łatwy sposób przedstawienie danych w postaci graficznej było kosztowne i bardzo pracochłonne. Problemy te były podstawą do stworzenia *GMF'a* (*GMF - Graphical Modeling Framework*), który jest jednym z projektów rozwijanych w ramach *Eclipse'a*. Jak sama nazwa wskazuje, *GMF* jest technologią do graficznego operowania na modelu danych. Praca z *GMFem* polega głównie na tworzeniu i edytowaniu odpowiednich plików XML (są do tego przygotowane specjalne kreatory oraz edytory), a sam wynikowy edytor graficzny jest generowany automatycznie i gotowy do użycia bez konieczności wykonywania dodatkowej pracy. Warto zwrócić uwagę, że wiele z dostępnych opcji (np. zbliżanie i oddalanie, widok *Outline*, możliwość drukowania, itp.) dostajemy „za darmo" i nie musimy poświęcać naszego czasu na ich implementację. Rysunek 1. przedstawia edytor stworzony właśnie z wykorzystaniem *GMFa*. Wraz z wydaniem najnowszej wersji platformy *Eclipse Ganymede* ukazała się również najnowsza wersja *GMF'a* oznaczona jako 2.1 (i właśnie ta wersja będzie omawiana w tym artykule).

Części składowe GMFa

Nazwę *Graphical Modeling Framework* można z łatwością rozbić na dwie części: pierwsza część dotyczy operacji graficznych (*graphical*), druga dotyczy modelowania (*modeling*). W *Eclipsie* do modelowania najczęściej wykorzystywany jest *EMF* (*Eclipse Modeling Framework*)



Rysunek 1. Zrzut ekranu przedstawiający przykładowy edytor wykorzystujący GMF'a.

> *GMF jest technologią do graficznego operowania na modelu danych.*

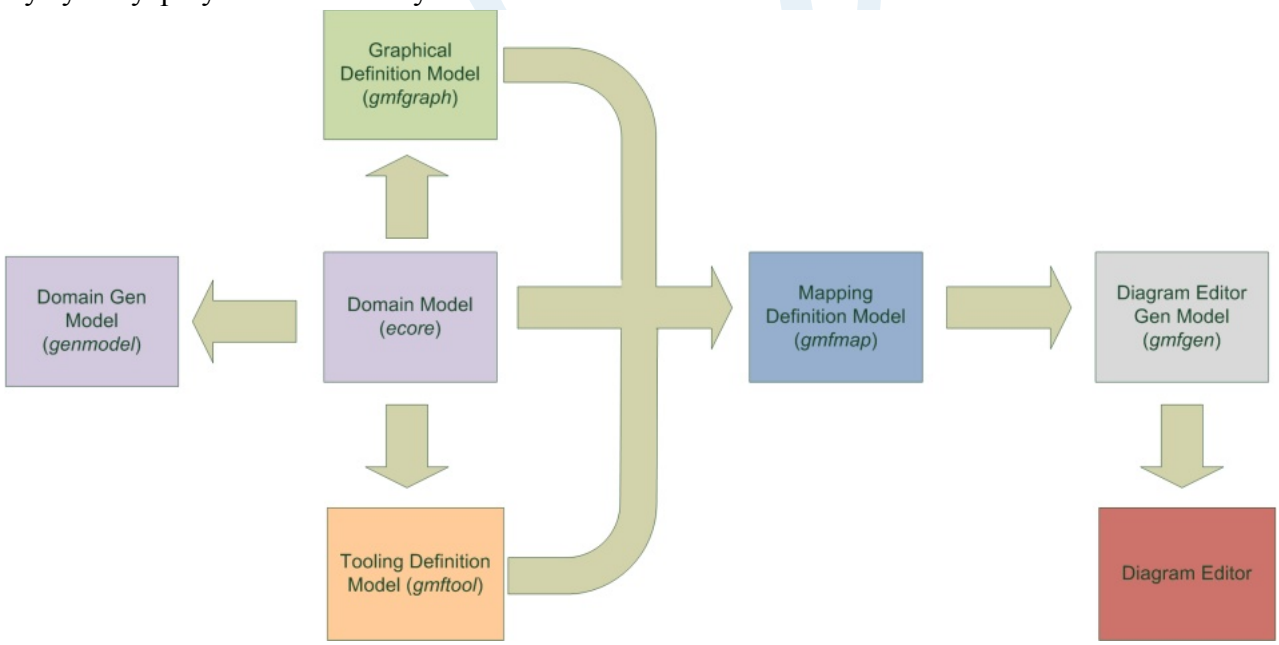
i dlatego właśnie ta technologia jest fundamentem *GMFa*. Jeśli chodzi o umożliwienie wykonywania operacji graficznych, to w ramach *Eclipse'a*, odpowiada za to *GEF* (*Graphical Editing Framework*). Jak łatwo się można domyślić, właśnie *GEF* jest podstawą *GMFa* dla budowania edytora graficznego.

Mimo zależności od *EMFa* i *GEFa*, wystarczy minimalna ich znajomość, aby rozpocząć pracę z *GMFem*. Dopiero przy tworzeniu bardziej zaawansowanych edytorów trzeba znać głębiej *EMFa* i *GEFa*. W artykule tym skupimy się na elementach *GMFa,* abstrahując (na ile jest to możliwe) od szczegółów pozostałych technologii, z którymi *GMF* jest związany.

Jak to działa? - trochę teorii

Zanim przejdziemy do konkretnego przykładu, warto zapoznać się z elementami i mechanizmami składającymi się na *GMFa* - schemat zależności między elementami przedstawiony jest na Rysunku 2.

Aby rozpocząć pracę potrzebny jest model, na którym chcemy operować. Model musi być w postaci pliku *ecore*, później plik ten jest wykorzystywany przy tworzeniu innych elementów. Jeśli tworzenie modelu bezpośrednio w pliku *ecore* sprawia nam trudność, możemy skorzystać z dostarczonego wraz z *GMFem* edytora umożliwiającego tworzenie modelu w sposób graficzny (zrzut ekranu tego edytora przedstawia Rysunek 1.). Aby skorzystać z tej możliwości wystarczy w widoku *Package Explorer* prawym przyciskiem myszy kliknąć na wybranym pliku *ecore* i z menu kontekstowego wybrać opcję `Initialize ecore_diagram diagram file`. Spowoduje to wygenerowanie pliku z rozszerzeniem *ecore_diagram*, który będziemy mogli edytować w sposób graficzny (zmiany automatycznie będą propagowane do odpowiedniego pliku *ecore*). Gdy mamy już gotowy model, to potrzebny jest plik *genmodel*, z którego później generujemy kod naszego modelu oraz projekt *edit*. W tym momencie kończy się nasza praca z modelem i zaczynamy się zajmować elementami typowymi dla *GMFa*. Na podstawie pliku *ecore* generujemy plik *gmfgraph*. Będzie on zawierał definicje elementów graficznych, które będziemy chcieli, aby pojawiły się w naszym edytorze. Aby móc umieszczać zdefiniowane elementy graficzne na edytorze potrzebujemy odpowiednie narzędzia umieszczone na palecie



Rysunek 2. Schemat zależności między elementami GMF'a.

> *Punktem centralnym GMF'a jest model w postaci pliku ecore,*

edytora – dostępne narzędzia definiujemy w pliku *gmftool*, który również powstaje na podstawie pliku *ecore*. Teraz nachodzi moment, gdy te trzy elementy (czyli definicje modelu, elementów graficznych oraz narzędzi na palecie) trzeba połączyć ze sobą – robimy to w pliku *gmfmap* (definiujemy tutaj m.in. który element modelu ma zostać stworzony po użyciu wybranego narzędzia i przez jaki element graficzny ma on być reprezentowany). Następnie z pliku *gmfmap* generujemy plik *gmfgen*, w którym możemy zdefiniować wybrane właściwości generowanego edytora. Gdy plik *gmfgen* jest już gotowy, możemy wygenerować kod edytora w postaci wtyczki (ang. *plugin*) do *Eclipse'a*.

Tworzymy edytor

Po wstępie teoretycznym przyszedł czas, żeby spróbować stworzyć własny edytor przy pomo-

Listing 1. Model w postaci pliku ecore

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="company"
    nsURI="gmf.example" nsPrefix="gmf.example">
  <eClassifiers xsi:type="ecore:EClass" name="Company">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="companyName"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="employees" upperBound="-1"
        eType="#//Employee" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="ownedComputers"
upperBound="-1"
        eType="#//Computer" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Manager" eSuperTypes="#//Employee">
    <eStructuralFeatures xsi:type="ecore:EReference" name="managedDevelopers"
upperBound="-1"
        eType="#//Developer"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Employee">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="usedComputers"
eType="#//Computer"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Computer">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Developer" eSuperTypes="#//Employee"/>
</ecore:EPackage>
```

> *Zaczniemy od graficznej strony naszego przyszłego edytora.*

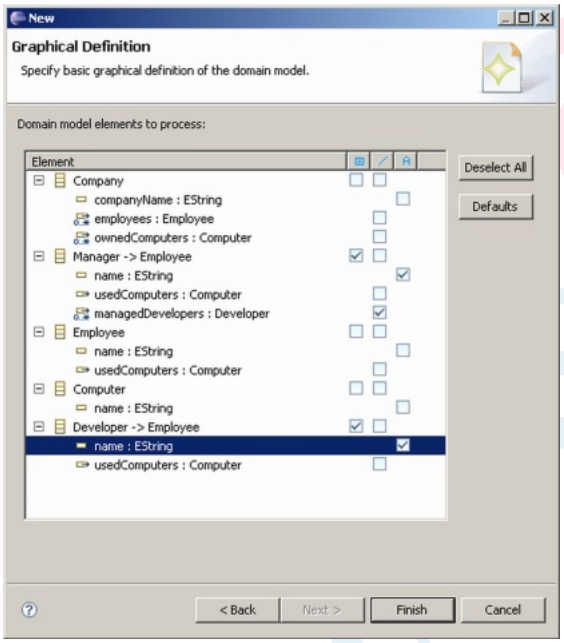
cy *GMFa*. Aby móc korzystać z możliwości *GMF'a,* trzeba go najpierw zainstalować w swoim *Eclipsie*. Jako że *GMF* jest częścią *Eclipse Ganymede,* to bez problemu możemy go pobrać za pomocą P2. Z głównego menu wybieramy `Help - > Software Updates...` W nowym oknie przechodzimy na zakładkę *Available Software,* rozwijamy węzeł *Ganymede Update Site*, następnie zaglądamy do kategorii *Models and Model Developement* i wybieramy *Graphical Modeling Framework SDK*. Klikamy przycisk *Install...* znajdujący się w prawym górnym rogu okna i po krótkiej chwili zostaniemy poproszeni o potwierdzenie, klikamy *Next*, na ostatniej stronie akceptujemy licencję (opcja *I accept the terms of the license agreement*) i klikamy *Finish*. Zostaniemy zapytani czy uruchomić ponownie *Eclipse'a*, klikamy *Yes* i *GMF* powinien już być częścią naszego środowiska. Po poprawnym zainstalowaniu *GMF'a* możemy rozpocząć pracę. Zaczynamy od stworzenia nowego projektu, czyli w głównym menu wybieramy `File - > New - > Project`, a następnie z listy dostępnych projektów wybieramy *New GMF Project*. W kreatorze podajemy nazwę projektu (niech będzie to *gmf.example*) i klikamy `Finish`. Punktem centralnym *GMF'a* jest model w postaci pliku *ecore*, dlatego też pierwszym krokiem w stronę stworzenia edytora jest zdefiniowanie modelu, który będziemy chcieli edytować. W celach dydaktycznych nasz model będzie dość prosty, aby nie komplikował nam przykładu. Listing 1 przedstawia nasz model w postaci pliku *ecore*. Aby z niego skorzystać, wystarczy w projekcie, w katalogu *model,* stworzyć nowy plik z rozszerzeniem *ecore* (u nas będzie to *company.ecore*). Powinniśmy zobaczyć informację, że plik jest błędny – nie przejmujemy się tym jednak, zamykamy edytor, a stworzony plik otwieramy w trybie tekstowym (w widoku *Package Explorer* z menu kontekstowego wybieramy opcję `Open With - > Text Editor`).

Do otwartego edytora wpisujemy treść listingu, zapisujemy plik i model w postaci pliku *ecore* mamy już gotowy (warto otworzyć plik *ecore* w edytorze *Sample Ecore Model Editor*). Zalecane jest zapoznanie się z modelem, gdyż jego znajomość będzie wymagana w dalszej części przykładu. Do dalszej pracy potrzebny nam będzie model w postaci kodu Javy oraz projekt *edit* (pozwala on na sprawne operacje na modelu). Dlatego też na podstawie naszego pliku *ecore* generujemy plik *genmodel* - kreator *EMF Model dostępny przez* `File - > New - > Other - >Eclipse Modeling Framework - > EMF Model`. W kreatorze podajmy nazwę pliku *company.genmodel* oraz zaznaczamy, że ma on zostać utworzony w katalogu *model* naszego projektu. Klikając `Next` przenosimy się na stronę, na której wybieramy w jakiej postaci jest nasz model (w naszym przypadku jest to *Ecore* model). Na kolejnej stronie podajemy ścieżkę do pliku z modelem – najlepiej wybrać `Browse Workspace`, a potem nasz plik *company.ecore*. Klikamy `Next`, a potem `Finish` – w katalogu *model* powinien pojawić się plik *company.genmodel.* Otwieramy wygenerowany przez kreator plik z rozszerzeniem *genmodel* i klikając prawym przyciskiem myszy na najwyższym elemencie wybieramy opcje: `Generate Model Code` oraz `Generate Edit Code`. W tej chwili możemy zostawić *EMF'a* i zająć się całkowicie *GMF'em*.

Zaczniemy od graficznej strony naszego przyszłego edytora. Klikamy prawym przyciskiem myszy na pliku *ecore* i z menu kontekstowego wybieramy opcje `New - > Other... - >Simple Graphical Definition Model` (kategoria *Graphical Modeling Framework*). W kreatorze wybieramy nazwę pliku – np. *company.gmfgraph* (najlepiej plik ten umieścić w katalogu *model*, aby wszystkie elementy definiujące nasz edytor były w jednym miejscu) i klikamy `Next`. Na kolejnej stronie kreatora wybieramy element, który będzie kontenerem dla pozostałych kontenerów – w naszym przypadku wybieramy element *Company* i klikamy `Next`. Na następnej stronie wy-

> *Kolejnym krokiem jest stworzenie definicji narzędzi*

bieramy, które elementy z modelu mają być wyświetlane w edytorze. Na razie chcemy, aby można było tworzyć obiekty typu *Manager* oraz *Employee*, a także połączenia między nimi oraz ich nazwy (wynik operacji na tej stronie przedstawiony jest na Rysunku 3.). Po zaznaczeniu odpowiednich pozycji kończymy pracę kreatora klikając `Finish`.

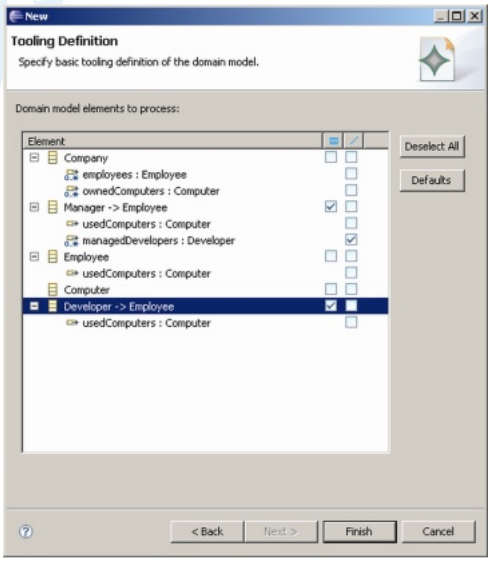


Rysunek 3. Strona kreatora definicji graficznej wskazująca, które elementy modelu mają być wyświetlane

Jak łatwo można zauważyć, w katalogu *model* pojawił się nowy plik z rozszerzeniem *gmfgraph*. Warto się przyjrzeć zawartości tego pliku, gdyż wszelkie zmiany dotyczące elementów graficznych naszego edytora będą dokonywane właśnie tutaj.

Kolejnym krokiem jest stworzenie definicji narzędzi, które będą dostępne na palecie edytora. W tym celu klikamy prawym przyciskiem myszy na pliku *ecore* i wybieramy `New -> Other... -> Simple Tooling Definition Model` (kategoria *Graphical Modeling Framework*). Podajemy nazwę pliku – np. *company.gmftool* (podobnie jak wcześniej proponuję umieścić nowy plik w katalogu *model*) i klikamy `Next`. Znów wskazujemy, co jest kontenerem dla naszego edytora (podobnie jak wcześniej wskazujemy *Company*) i przechodzimy do kolejnej strony, klikając przycisk `Next`. Ostatnia strona kreatora pomaga nam określić, do których elementów modelu mają zostać stworzone narzędzia na palecie. Chcemy dodawać elementy typu *Manager* i *Employee* oraz połączenia między nimi, więc efekt końcowy pracy na tej stronie powinien wyglądać tak samo, jak na Rysunku 4. Klikamy `Finish`, żeby zakończyć pracę kreatora.

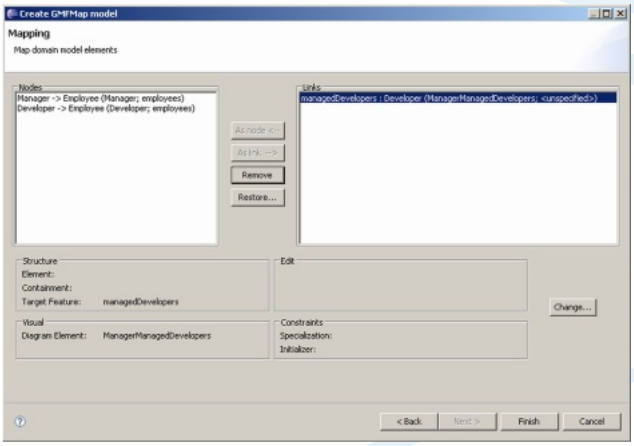
Skoro mamy już nasz model, definicję elementów graficznych oraz narzędzi do ich tworzenia pozostaje nam to wszystko połączyć. Klikamy prawym przyciskiem myszy na pliku *ecore* i wybieramy `New -> Other -> Guide Mapping Model Creation` (kategoria *Graphical Modeling Framework*). Podobnie jak w przypadku poprzednich kreatorów, również tutaj najpierw podajemy nazwę pliku, który będzie przechowywał definicje mapowań – np. *company.gmfmap* (jako folder docelowy ponownie sugeruję folder *model*). Na drugiej stronie kreatora wskazujemy kontener (ponownie będzie to *Company*) i klikamy `Next`. Na następnej stronie wskazujemy definicję narzędzi, w tym celu klikamy przycisk



Rysunek 4. Strona kreatora definicji narzędzi wskazująca, dla których elementów modelu będą stworzone narzędzia.

> *Mapowanie jest gotowe...*

`Browse Workspace...` i wskazujemy nasz plik *gmftool*, a następnie klikamy `Next`. Kolejna strona wymaga od nas wskazania pliku z definicją elementów graficznych. Tak samo jak poprzednio klikamy przycisk `Browse Workspace...`, tylko tym razem wybieramy nasz plik *gmfgraph* i klikamy `Next`. Ostatnia strona kreatora stanowi jego istotę, gdyż pozwala określić, które elementy będą wierzchołkami (ang. *nodes*) w naszym edytorze, a które połączeniami (ang. *links*). Z sekcji *Nodes* usuwamy pozycję *Computer (Manager; ownedComputers)*, a z sekcji *Links* usuwamy *usedComputers : Computer (ManagerManagedDevelopers; <unspecified>* - strona powinna wyglądać tak jak na Rysunku 5. Klikamy `Finish`, aby zakończyć pracę kreatora.



Rysunek 5. Strona kreatora wskazująca wierzchołki, połączenia i atrybuty.

Mając wygenerowany plik *gmfmap,* musimy zajrzeć do niego i sprawdzić (w widoku *Properties*), czy ustawione są właściwości *Diagram Label* dla elementów *Feature Label Mapping*. Jeśli nie są one ustawione (kreator nie zawsze jest w stanie odgadnąć nasze intencje), to należy wprowadzić do nich następujące wartości:

- *DiagramLabel DeveloperName* dla elementu tego typu znajdującego się pod węzłem *Node Mapping <Developer/Developer>*
- *Diagram Label ManagerName* dla elementu pod węzłem *Node Mapping <Manager/Manager>*

Możemy jeszcze sprawdzić, czy odpowiednie narzędzia są przypisane do odpowiednich elementów. I tak:

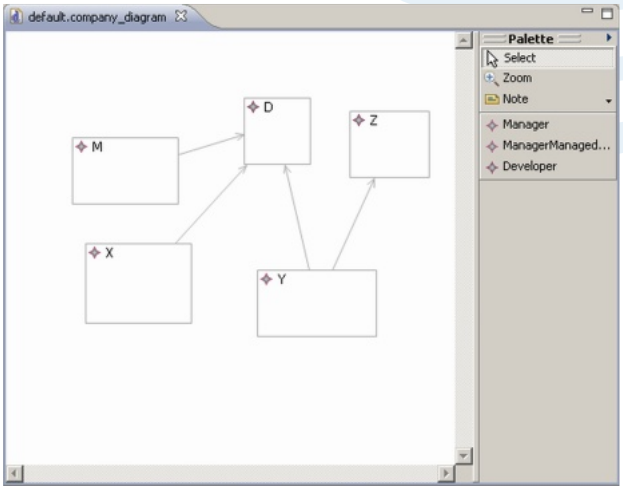
- węzeł *Node Mapping <Developer/Developer>* powinien mieć właściwość *Tool* ustawioną na wartość *Tool Developer*
- węzeł *Node Mapping <Manager/Manager>* powinien mieć właściwość *Tool* ustawioną na wartość *Tool Manager*
- węzeł *Link Mapping* powinien mieć właściwość *Tool* ustawioną na wartość *Tool ManagerManagedDevelopers*

Mapowanie jest gotowe, teraz jeszcze musimy wygenerować model generatora. W tym celu klikamy prawym przyciskiem myszy na pliku *gmfmap* i wybieramy opcję `Create generator model...`. Podajemy nazwę pliku – np. *company.gmfgen* (znów jako folder docelowy sugeruję folder *model*) i klikamy `Next`, aż do osiągnięcia ostatniej strony, na której kończymy pracę kreatora przez kliknięcie przycisku `Finish`.

Możemy teraz już wygenerować kod naszego edytora. Prawym przyciskiem myszy klikamy na plik *gmfgen* i wybieramy opcję `Generate diagram code`. W naszej przestrzeni pracy (ang. *workspace*) powinien pojawić się nowy projekt, który jest gotową wtyczką zawierającą kod edytora. Aby przetestować nasz edytor, musimy uruchomić *Eclipse'a* z tą właśnie wtyczką (a także z wtyczką zawierającym nasz model oraz wtyczką *edit*). W tym celu z menu głównego wybieramy `Run -> Run Configurations`, w nowym oknie na liście po lewej stronie zaznaczamy *Eclipse Application* i klikamy na ikonę *New launch configuration.* W tym momencie stworzy nam się nowa konfiguracja uruchomienia (ang. *launch configuration*) pozwalająca nam uruchomić nową instancję *Eclipse'a* z wybranymi przez nas wtyczkami (nie będziemy wchodzić w szczegóły, gdyż jest to poza zakresem tego artykułu). U góry w polu *Name* podajemy nazwę naszej

Zastanówmy się najpierw...

konfiguracji, a na zakładce *Plug-ins* sprawdzamy, czy zaznaczone są nasze nowe wtyczki. Zatwierdzamy zmiany przyciskiem `Apply` i uruchamiamy naszą nową konfigurację klikając przycisk `Run`. W nowej instancji *Eclipse'a* tworzymy nowy projekt, a następnie klikając prawym przyciskiem myszy na projekcie wybieramy `New -> Other` i z listy dostępnych opcji wybieramy *Company Diagram* (nazwę tę oczywiście można zmienić, lecz w tej chwili nie jest to kluczowe w naszym przykładzie). W kreatorze podajemy nazwę pliku i klikamy `Finish`. Powinien otworzyć się nasz edytor w którym możemy dodawać nowe elementy z palety znajdującej się po prawej stronie edytora (Rysunek 6).



Rysunek 6. Zrzut ekranu edytora zaraz po jego wygenerowaniu

Dodajemy nowe elementy

W ciągu kilku chwil bez większego wysiłku udało nam się stworzyć w pełni funkcjonalny edytor bez pisania ani jednej linijki kodu, a jedynie tworząc i edytując kilka plików *XML* (przy pomocy kreatorów i edytorów upraszczających znacznie to zadanie). Teraz czas na zrobienie czegoś samodzielnie bez korzystania z kreatorów – dodamy do naszego edytora możliwość tworzenia nowych elementów typu *Computer* (i tym razem obejdzie się bez pisania kodu).

Zastanówmy się najpierw – w których miejscach musimy dokonać zmian? Po pierwsze, musimy zdefiniować nowe elementy graficzne (czyli czeka nas modyfikacja pliku *gmfgraph*). Po drugie, musimy dodać nowe narzędzia do palety (czyli zajmiemy się plikiem *gmftool*). Po trzecie - będziemy musieli w odpowiedni sposób zmapować elementy naszego modelu na poszczególne elementy graficzne oraz wskazać, przy użyciu którego narzędzia elementy te mają być tworzone (czyli będziemy musieli dokonać zmian w pliku *gmfmap*).

Skoro wiemy już, gdzie trzeba wprowadzić modyfikacje, to możemy się zabrać do pracy. Zacznijmy od dodania nowych elementów graficznych. Po otworzeniu pliku *gmfgraph* widzimy, że jego głównym elementem jest element o nazwie *Canvas*. Elementy wchodzące w jego skład to:

- kształty/figury dostępne w ramach aplikacji (są one tworzone w węźle *Figure Gallery Default)*
- wierzchołki (ang. *nodes*) – graficzna reprezentacja elementów modelu
- etykiety diagramu (ang. *diagram labels*) – etykiety wierzchołków
- połączenia (ang. *connections*) – linie wskazujące na związki między elementami modelu

Załóżmy, że chcemy, aby elementy typu *Computer* wyświetlane były jako elipsa. Zdefiniujemy najpierw ich wygląd: zaczniemy od stworzenia deskryptora figury (ang. *figure descriptor*), który będzie przechowywał wszystkie informacje o naszej nowej figurze. Aby stworzyć nowy deskryptor figury, klikamy prawym przyciskiem na element *Figure Gallery Default* i wybieramy `New Child -> Figure Descriptor`. Stworzy się nowy węzeł, któremu nadajemy (w widoku *Properties*) nazwę (najlepiej jak najbardziej deskryptywną, np. *ComputerFigureDescriptor*). Teraz musimy zdefiniować wygląd naszej figury, w naszym przypadku będzie to elipsa, ale w ogólności mo-

> *Ostatnim etapem pracy, który musimy wykonać jest połączenie modelu, definicji graficznej oraz narzędzi*

że to być dowolny kształt (nawet składający się z wielu figur składowych). Klikamy prawym przyciskiem na nowo stworzonym deskryptorze i wybieramy `New Child - > Ellipse`. Nowemu węzłowi nadajemy nazwę (np. *ComputerEllipse*), możemy oczywiście również zmienić jego pozostałe właściwości. Chcemy, aby wewnątrz figury pojawiła się jego nazwa w postaci etykiety (ang. *label*), więc klikamy prawym przyciskiem na *ComputerEllipse* i wybieramy `New Child - >Label`. Nowemu węzłowi nadajemy nazwę (np. *ComputerNameLabel*). Musimy jeszcze stworzyć regułę dostępu do tej etykiety, aby później można się było do niej odwołać (w dalszej części artykułu zobaczymy, kiedy ma to zastosowanie) – klikamy prawym przyciskiem na *ComputerFigureDescriptor* i wybieramy `New Child - > Child Access`. We właściwościach nowego węzła wybieramy element, do którego chcemy mieć dostęp – w naszym przypadku właściwość *Figure* ustawiamy na *ComputerNameLabel*. Zdefiniowaliśmy już jak ma wyglądać nowa figura, teraz musimy jeszcze stworzyć dla niej definicję wierzchołka. Klikamy prawym przyciskiem myszy na *Canvas* i wybieramy `New Child - > Nodes Node`. Nowemu węzłowi nadajemy nazwę (np. *ComputerFigure*) i wiążemy go z naszą figurą przez podanie nazwy deskryptora figury (czyli *ComputerFigureDescriptor*) we właściwości *Figure*. Dla stworzonej etykiety trzeba stworzyć element typu *diagram label*, aby edytor był świadom, że chcemy ją wyświetlić. Klikamy na *Canvas* i wybieramy `New Child - > Labels Diagram Label`. Tradycyjnie już nadajemy nazwę nowemu węzłowi (np. *ComputerName*) i wiążemy go z deskryptorem przechowującym definicję figury, czyli podobnie, jak poprzednio, ustawiamy właściwość *Figure* na wartość *ComputerFigureDescriptor*. Teraz właśnie zastosowanie ma reguła dostępu do etykiety, którą tworzyliśmy chwilę wcześniej – właściwość *Accessor* ustawiamy na wartość *Child Access getFigureComputerNameLabel*.

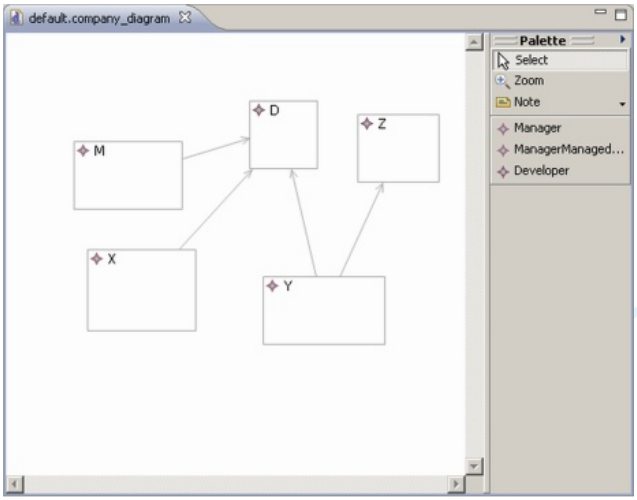
Definicję graficzną mamy już gotową, możemy teraz zdefiniować narzędzia, które chcemy dodać (w naszym przypadku chcemy mieć możliwość dodania elementów typu *Computer*, więc potrzebujemy jedno nowe narzędzie na palecie). Otwieramy plik *gmftool*, klikamy prawym przyciskiem na węźle *Tool Group* i z menu wybieramy `New Child - > Creation Tool`. Nowemu węzłowi nadajemy tytuł (właściwość *Title*), *np. ComputerTool* i opis (właściwość *Description*). Musimy jeszcze zdefiniować ikony, które pojawią się na palecie i w edytorze dla naszego elementu – mogą to być ikony domyślne (tak będzie w naszym przypadku) lub wskazane przez nas. Aby dodać standardowe ikony klikamy prawym przyciskiem myszy na stworzonym przed chwilą węźle i wybieramy New Child - > Large Icon Default Image, a następnie New Child - > Small Icon Default Image. Na tym kończymy edycję pliku *gmftool*.

Ostatnim etapem pracy, który musimy wykonać jest połączenie modelu, definicji graficznej oraz narzędzi, czyli modyfikacja pliku *gmfmap*. Gdy otworzymy wspomniany plik zobaczymy, że główny węzeł nazywa się *Mapping*. Przechowuje on wszystkie zdefiniowane mapowania. Właśnie do węzła *Mapping* musimy dodać nasze mapowanie, a ponieważ dotyczy ono wierzchołka edytora, to dodamy mapowanie typu *Top Node Reference*. Klikamy na *Mapping* prawym klawiszem myszy i wybieramy `New Child - > Top Node Reference`. W nowym węźle musimy ustalić, gdzie w modelu ma być przechowywany element reprezentowany przez wierzchołek (właściwość *Containment Feature*). W naszym przypadku będzie to *ownedComputers*. Następnie na nowym węźle klikamy prawym przyciskiem myszy i wybieramy `New Child - >Node Mapping` i właśnie tutaj wprowadzamy szczegóły dotyczące mapowania: właściwość *Element* ustawiamy na wartość odpowiadającą elementowi z modelu (czyli u nas będzie to *Computer*), dla właści-

> *Możliwości GMF'a są oczywiście dużo szersze*

wości *Diagram Node* ustawiamy nazwę wierzchołka z definicji graficznej (u nas *ComputerFigure*), dla właściwość *Tool* podajemy nazwę narzędzia zdefiniowanego w pliku *gmftool* (u nas *ComputerTool*). Chcemy jeszcze wyświetlić nazwę elementu *Computer* jako etykietę, w tym celu klikamy prawym przyciskiem myszy na stworzonym przez nas węźle i wybieramy *New Child- >Feature Label Mapping*. Dla nowego węzła musimy ustawić właściwość *Features* na wartość *name* (bo właśnie ten atrybut elementu *Computer* chcemy wyświetlić w etykiecie), a właściwość *Diagram Label* powinna wskazywać na etykietę diagramu zdefiniowaną w pliku *gmfgraph* (czyli na wartość *ComputerName*).

I w ten oto sposób kończy się nasza praca. Teraz trzeba wygenerować ponownie plik gmfgen, a później na jego podstawie kod edytora i sprawdzić, czy nasz edytor działa tak, jak się tego spodziewaliśmy. Rysunek 7. prezentuje efekt naszej pracy.



Rysunek 7. Zrzut ekranu edytora zaraz po wprowadzonych zmianach

Podsumowanie

W artykule zaprezentowano podstawowe możliwości, jakie daje nam projekt *GMF*, a także pokazano, jak w prosty i szybki sposób można stworzyć w pełni funkcjonalny graficzny edytor modelu danych. Możliwości *GMF'a* są oczywiście dużo szersze, dlatego zachęcam do zapoznania się z przykładami i tutorialami umieszczonymi w internecie oraz do własnego eksperymentowania z tą technologią.

W sieci

- Strona domowa GMF'a: http://www.eclipse.org/gmf/
- Tutorial GMF'a: http://wiki.eclipse.org/index.php/GMF_Tutorial
- GMF FAQ: http://wiki.eclipse.org/Graphical_Modeling_Framework_FAQ
- GMF w 15 minut: http://www-128.ibm.com/developerworks/opensource/library/os-ecl-gmf/
- Artykuł o dodatkowym widoku GMF'a: http://eclipser-blog.blogspot.com/2007/06/gmf-project-in-5-minutes-with-gmf.html
- Zbiór wpisów dotyczących *EMF'a:* http://eclipse-po-polsku.blogspot.com/search/label/EMF

Autor jest doktorantem na Politechnice Poznańskiej, zajmuje się problemami inżynierii oprogramowania. Pracuje w IBM Eclipse Support Center, jest aktywnym członkiem społeczności Eclipse, prowadzi blog http://eclipse-po-polsku.blogspot.com/
Kontakt z autorem: jakub.jurkiewicz@gmail.com



Java Team

dołącz do nas www.isolution.pl

J2ME: Serializacja obiektów, cz. II

Adam Dec

Przygotowanie środowiska i stworzenie projektu

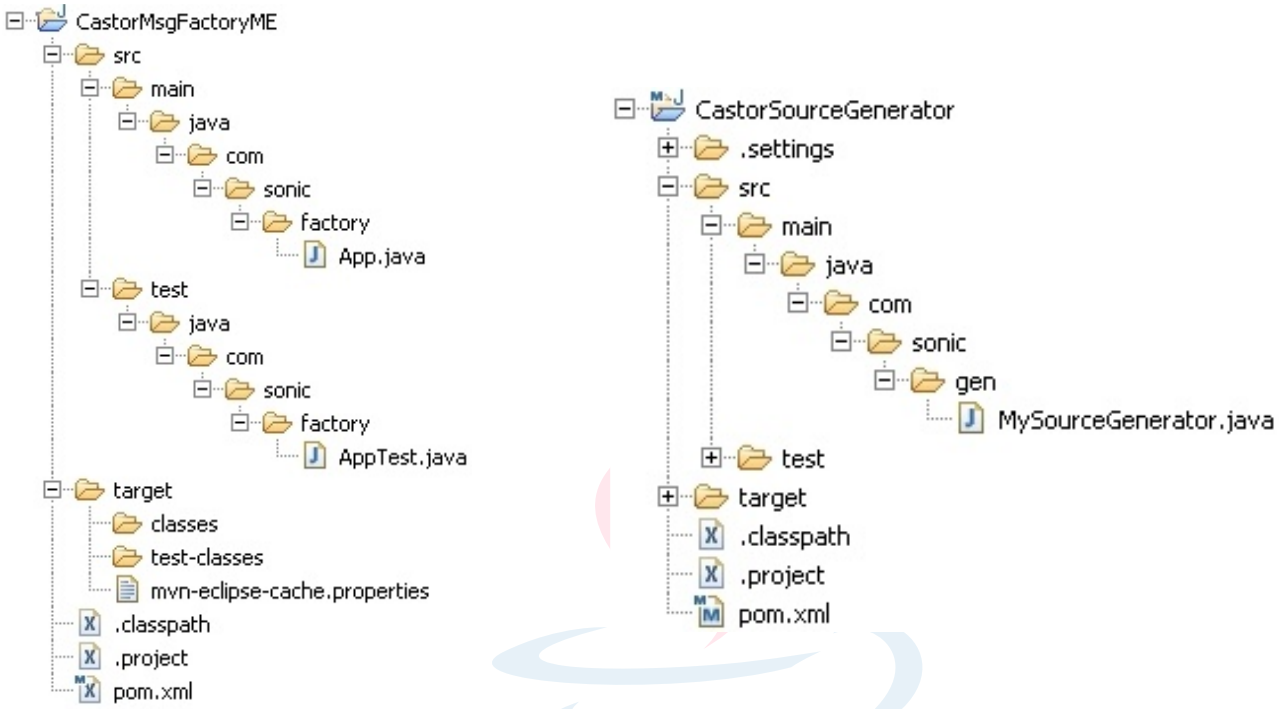
Narzędziem, którego będziemy używać do budowania projektu będzie oczywiście Maven 2.x [1]. Aby było ciekawiej w fazie generowania źródeł (`generate-sources`), użyjemy plugin'u (`maven-antrun-plugin`). Zadaniem Ant'a [2] będzie generowanie kodu wykorzystując klasę:
`org.castor.anttask.CastorCodeGenTask`
Przed dalszą częścią artykułu radzę o zaznajomienie się ze świetnym artykułem na temat Maven'a z drugiego wydania Java Express, Grudzień 2008 (Maven 2 - jak ułatwić sobie pracę, cz. I, Rafał Kotusiewicz).
Stworzymy dwa projekty:
1. `CastorSourceGenerator`
2. `CastorMsgFactoryME`

Projekty możemy stworzyć albo wydając komendy z linii poleceń albo poprzez wtyczke do Eclipse (M2 **[3]**). Ja wykorzystam polecenia:
`mvn archetype:create -DgroupId=com.sonic.gen -DartifactId=CastorSourceGenerator`
`mvn archetype:create -DgroupId=com.sonic.factory -DartifactId=CastorMsgFactoryME`
Maven stworzy nam 2 katalogi z nazwami takimi jak podaliśmy w atrybutach `artifactId`. Każdy katlog zawiera podkatalogi wraz z plikami typu „HelloWorld”:
`src/main/java/com/sonic/gen/App.java`
`src/test/java/com/sonic/gen/TestApp.java`
oraz plik POM (Project Object Model), z przykładową zawartością:

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

Jeżeli chcemy zaimportować utworzone szablony jako projekt w Eclipse/IntelliJ, to w każdym z katalogów (CastorSourceGenerator, CastorMsgFactoryME, CastorTester) wywołujemy komendę:
`mvn eclipse:eclipse lub mvn idea:idea`
Jeżeli wybraliśmy Eclipse to zostaną dodane 2 pliki: `.project oraz .claspath`
Zostanie utworzony także katalog `target` a w nim plik `mvn-eclipse-cache.properties`.
Po zaimportowaniu szablonów jako projekt java w eclipse (File->New->Java Project...Create project from existing source) powinniśmy otrzymać strukturę przedstawioną na rys. 1

> *Narzędziem, którego będziemy używać do budowania projektu będzie oczywiście Maven*



Rys. 1 Widok w Eclipse 3.5M3

Najpierw zajmiemy się projektem `CastorSourceGenerator`. Utworzymy klasę o nazwie `MySourceGenerator`, która przejmie kontorolę nad domyślną generacją kodu wynikowego. Jej w pełni kwalifikowaną nazwę podamy jako wartość atrybutu: `org.exolab.castor.builder.jclassPrinterTypes` w pliku `castorbuilder.properties`. W klasie tej zaimplementujemy interfejs `org.exolab.castor.builder.printing.JClassPrinter`. Przykład jego użycia znajdziemy w klasie `org.exolab.castor.builder.printing.WriterJClassPrinter` (listing 2). Klasa `WriterJClassPrinter` posłuży nam jako szablon, na jej bazie zbudujemy wła-

```
package org.exolab.castor.builder.printing;
import org.exolab.javasource.JClass;
import org.exolab.javasource.JComment;
public class WriterJClassPrinter implements JClassPrinter {

    public void printClass(final JClass jClass, final String outputDir,
    final String lineSeparator, final String header) {

        // hack for the moment
        // to avoid the compiler complaining with java.util.Date
        jClass.removeImport("org.exolab.castor.types.Date");
        // add header
        JComment comment = new JComment(JComment.HEADER_STYLE);
        comment.appendComment(header);
        jClass.setHeader(comment);
        // print
        jClass.print(outputDir, lineSeparator);
    }
}
```

> *Radzę zrobić sobie kopie tego katalogu*

sną implementację.

1. Zamieniamy nazwę klasy `App.java` na `MySourceGenerator.java`, usuwamy wszystkie zbędne rzeczy i implementujemy interfejs `JclassPrinter`. Klasy, których głównie będziemy używać, znajdują się w pakiecie: `org.exolab.javasource`. Ich nazwy tak naprawdę odzwierciedlają to co chcemy zrobić. Czyli chcąc stworzyć metodę, tworzymy obiekt JMethod natomiast chcąc dodać parametr do metody, utworzymy obiekt `Jparameter`.

2. Na razie nie przejmujemy się tym, że projekt się nie buduje, ściągamy źródła Castor'a (uwaga musi to być wersja 1.2!!!) [4] i przechodzimy do katalogu codegen. Radzę zrobić sobie kopie tego katalogu, kopiując jego zawartość np. do katalogu codegen-me. Musimy trochę pogrzebać w klasach. Zmianie ulegną 3 pliki: `Jtype.java`, `CollectionMemberAndAccessorFactory.java` oraz `SourceFactory.java`.

Pamiętając o sygnaturach metod do serializacji/deserializacji:

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

w klasie `org.exolab.javasource.JType` dodajemy następujące rzeczy:

```
/** JType instance for a void (void). */
public static final JPrimitiveType VOID = new JPrimitiveType("void", "void");
 /** JType for a DataInputStream. */
public static final JPrimitiveType DATA_INPUT_STREAM = new JPrimitiveType("java.io.DataInputStream", "java.io.DataInputStream");
/** JType instance for a DataOutputStream. */
public static final JPrimitiveType DATA_OUTPUT_STREAM = new JPrimitiveType("java.io.DataOutputStream", "java.io.DataOutputStream");
```

3. Szukamy klasy `org.exolab.castor.builder.factory.CollectionMemberAndAccessorFactory`. Musimy zadbać aby kod wygenerowany dla kolekcji takich jak np. `java.util.Vector` i `java.util.Hashtable` wołał tylko metody zgodne z J2ME API [5]. Prawie wszystkie metody dostepne w Java 1.1 [6] są zawarte w CLDC 1.1 (JSR 139). Nie znajdziemy tam tylko metody `public synchronized Object clone();`

Musimy zadbać aby w kodzie wynikowym istniały odwołania tylko do takich metod jak:

```
void addElement(Object obj)
Object elementAt(int index)
void insertElementAt(Object obj, int index)
Enumeration elements()
void removeAllElements()
boolean removeElement(Object obj)
void removeElementAt(int index)
void setElementAt(Object obj, int index)
```

3.1 Pierwsza zmiana nastąpi w metodzie:

```
private void createGetAsArrayMethod(final CollectionInfo fieldInfo,
    final JClass jClass, final boolean useJava50,
     AnnotationBuilder[] annotationBuilders) { … }
```

Według JSR 139 w klasie `java.util.Vector` nie mamy takich metod jak `Object[] toArray()` oraz `Object[] toArray(Object[] a)`.

W takim wypadku albo usuwamy ciało metody `createGetAsArrayMethod(...)` albo szukamy metody:

```
private void createGetAndSetMe-
```

> *Na razie nie przejmujemy się tym, że projekt się nie buduje*

```
thods(final CollectionInfo fieldInfo,
     final JClass jClass, final boole-
an useJava50,
     final AnnotationBuilder[] annota-
tionBuilders) { … }
```

i usuwamy w niej linijkę:

```
this.createGetAsArrayMethod(fieldIn-
fo, jClass, useJava50, annotationBuil-
ders);
```

3.2 Ta sama operację powtarzamy w przypadku metod:

```
private void createGetAsReferenceMe-
thod(final CollectionInfo fieldInfo,
    final JClass jClass) { ... }
private void createSetAsReferenceMe-
thod(final CollectionInfo fieldInfo,
     final JClass jClass, final boole-
an useJava50) { ... }
private void createSetAsCopyMethod(fi-
nal CollectionInfo fieldInfo,
    final JClass jClass) { ... }
```

W moim przypadku wszystkie implementacje metod zostały usunięte a metoda `createGetAndSetMethods` wygląda jak poniżej.

```
private void createGetAndSetMethods(fi-
nal CollectionInfo fieldInfo,
     final JClass jClass, final boole-
an useJava50,
     final AnnotationBuilder[] annota-
tionBuilders) {
      this.createGetByIndexMethod(fiel-
dInfo, jClass);
      this.createSetByIndexMethod(fiel-
dInfo, jClass);
}
```

3.3 Kolejna zmiana nastąpi w metodzie:

```
protected void createGetByIndexMe-
thod(final CollectionInfo fieldInfo,
    final JClass jClass) { ... }
```

w linijce:

```
      String value = fieldInfo.getNa-
me() + ".get(index)";
```

słowo get zamieniamy na elementAt. Czyli otrzymamy:

```
      String value = fieldInfo.getNa-
me() + ".elementAt(index)";
```

Czyli zmieniamy `Object get(int index);` na `Object elementAt(int index);`

3.4 Szukamy metody:

```
protected void createAddByIndexMe-
thod(final CollectionInfo fieldInfo,
    final JClass jClass) { ... )
```

Kawałek kodu:

```
sourceCode.append(fieldInfo.getNa-
me());
sourceCode.append(".add(index, ");
sourceCode.append(fieldInfo.getCon-
tentType().
      createToJavaObjectCode(parame-
ter.getName()));
sourceCode.append(");");
void add(int index, Object element)
```

zamieniamy na:

```
sourceCode.append(fieldInfo.getNa-
me());
sourceCode.append(".insertElemen-
tAt(");
sourceCode.append(fieldInfo.getCon-
tentType().
      createToJavaObjectCode(parame-
ter.getName()));
sourceCode.append(", index);");
void insertElementAt(Object obj, int
index)
```

3.5 Usuwamy ciało metody:

```
protected void createIteratorMe-
```

> *W moim przypadku wszystkie implementacje metod zostały usunięte*

```
thod(final CollectionInfo fieldInfo,
    final JClass jClass, final boole-
an useJava50) { ... }
```

A fragment implementacji metody:

```
private void createRemoveAllMethod(fi-
nal CollectionInfo fieldInfo,
    final JClass jClass) { ... }
sourceCode.append(".clear();");
```

zamieniamy na `sourceCode.append(".remo-veAllElements();");`

C

zyli `void clear();` na `void removeAl-lElements();`

3.6 W nastepnej metodzie zmieniamy ciało:

```
protected void createRemoveByIndexMe-
thod(final CollectionInfo fieldInfo,
        final JClass jClass) { ... }
```

Z:

```
JMethod method = new JMethod("remove" + fieldInfo.getMethodSuffix() + "At",
    fieldInfo.getContentType().getJType(),
    "the element removed from the collection");
method.addParameter(new JParameter(JType.INT, "index"));
JSourceCode sourceCode = method.getSourceCode();
sourceCode.add("java.lang.Object obj = this.");
sourceCode.append(fieldInfo.getName());
sourceCode.append(".remove(index);");
if (fieldInfo.isBound()) this.createBoundPropertyCode(fieldInfo, sourceCode);

sourceCode.add("return ");
if (fieldInfo.getContentType().getType() == XSType.CLASS) {
    sourceCode.append("(");
    sourceCode.append(method.getReturnType().getName());
    sourceCode.append(") obj;");
} else {
    sourceCode.append(fieldInfo.getContentType().createFromJavaObjectCode("obj"));
    sourceCode.append(";");
}
jClass.addMethod(method);
```

Na:

```
JMethod method = new JMethod("remove" + fieldInfo.getMethodSuffix() + "At", Jty-
pe.VOID, "the element removed from the collection");
method.addException(SGTypes.INDEX_OUT_OF_BOUNDS_EXCEPTION,
    "if the index given is outside the bounds of the collection");
method.addParameter(new JParameter(JType.INT, "index"));
JSourceCode sourceCode = method.getSourceCode();
this.addIndexCheck(fieldInfo, sourceCode, method.getName());
sourceCode.add("this.");
sourceCode.append(fieldInfo.getName());
sourceCode.append(".removeElementAt(index);");
if (fieldInfo.isBound()) this.createBoundPropertyCode(fieldInfo, sourceCode);
jClass.addMethod(method);
```

czyli po krótce `Object remove(int index)` na `void removeElementAt(int index)`

Metoda `this.addIndexCheck(...)` doklei do naszego kodu coś takiego:

```
// check bounds for index
if (index < 0 || index >= this.VECTOR .si-
ze()) {
        throw new IndexOutOfBoundsExcep-
tion("getElement: Index value '" + index
+ "' not in range [0.." +
(this.VECTOR.size() - 1) + "]");
}
```

3.7 Kolejna zmiana dotyczy metody:

> ... i to już koniec "podmianek"

```
private void createRemoveObjectMethod(final CollectionInfo fieldInfo,
    final JClass jClass) { ... }
```

Podmieniamy kod `sourceCode.append(".remove(");` na `sourceCode.append(".removeElement(");`

3.8 Usuwamy ciało metody

```
private void createSetAsArrayMethod(final CollectionInfo fieldInfo,
    final JClass jClass, final boolean useJava50) { ... }
```

3.9 ostatnia zmiana będzie dotyczyła metody:

```
protected void createSetByIndexMethod(final CollectionInfo fieldInfo,
    final JClass jClass) { ... }
```

Zamieniamy metodę `Object set(int index, Object element)` na `void setElementAt(Object obj, int index)` podmieniając kod:

```
JMethod method = new JMethod("set" + fieldInfo.getMethodSuffix());
method.addException(SGTypes.INDEX_OUT_OF_BOUNDS_EXCEPTION,
    "if the index given is outside the bounds of the collection");
method.addParameter(new JParameter(JType.INT, "index"));
method.addParameter(new Jparameter(fieldInfo.getContentType().getJType(), fieldInfo.getContentName()));
JSourceCode sourceCode = method.getSourceCode();
this.addIndexCheck(fieldInfo, sourceCode, method.getName());
sourceCode.add("this.");
sourceCode.append(fieldInfo.getName());
sourceCode.append(".set(index, ");
sourceCode.append(fieldInfo.getContentType().createToJavaObjectCode(fieldInfo.getContentName()));
sourceCode.append(");");
if (fieldInfo.isBound()) this.createBoundPropertyCode(fieldInfo, sourceCode);
jClass.addMethod(method);
```

na:

```
JMethod method = new JMethod("set" + fieldInfo.getMethodSuffix(), JType.VOID,
    "the element added to the collection");
method.addException(SGTypes.INDEX_OUT_OF_BOUNDS_EXCEPTION,
    "if the index given is outside the bounds of the collection");
method.addParameter(new JParameter(JType.INT, "index"));
method.addParameter(new JParameter(fieldInfo.getContentType().getJType(),
fieldInfo.getContentName()));
JSourceCode sourceCode = method.getSourceCode();
this.addIndexCheck(fieldInfo, sourceCode, method.getName());
sourceCode.add("this.");
sourceCode.append(fieldInfo.getName());
sourceCode.append(".setElementAt(");
sourceCode.append(fieldInfo.getContentType().createToJavaObjectCode(fieldInfo.getContentName()));
sourceCode.append(", index);");
if (fieldInfo.isBound()) this.createBoundPropertyCode(fieldInfo, sourceCode);
jClass.addMethod(method);
```

3.10 Krótkie wyjaśnienie

Spis klas, których możemy użyć do generowania kodu źródłowego można znaleźć tutaj: http://www.castor.org/1.3/javadoc/org/exolab/javasource/package-summary.html

S

zukamy klasy `org.exolab.castor.builder.factory.Sourcefactory` a w niej metody `private void initialize(final JClass jClass) { ... }`. Musimy wykomentować tylko jedną linijkę

> *Teraz już projekt powinien się budować*

```
jClass.addInterface("java.io.Serializable");
```

Nie chcemy aby ten interfejs domyślnie był doklejany do każdej nowo wygenerowanej klasy, ponieważ jak już wcześniej wspominałem interfejs ten nie istnieje w Java ME. Zamiast niego użyjemy `de.enough.polish.io.Externalizable`

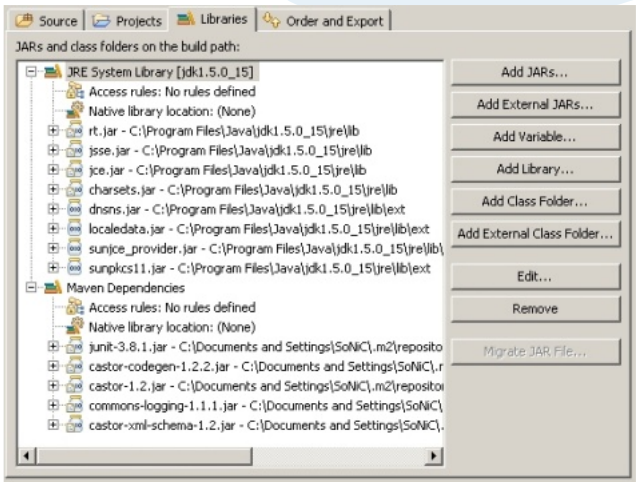
...i to już koniec „podmianek" :)

Edytujemy plik `pom.xml` i zmieniamy nazwę artefaktu na `castor-codegen` i dodajemy wersję: `<version>1.2.1</version>`. Wydajemy polecenie `mvn clean install`. Artefakt zostanie zainstalowany w naszym lokalnym repozytorium w katalogu: `M2_REPO/repository/org/codehaus/castor/castor-codegen/1.2.1/`

Tak stworzony artefakt dodajemy jako zależność w pliku `pom.xml`:

```
<dependency>
    <groupId>org.codehaus.castor</groupId>
  <artifactId>castor-codegen</artifactId>
 <version>1.2.1</version>
</dependency>
```

Jeżeli mamy zainstalowany plugin M2 to klikamy na projekcie prawym przyciskiem myszki i wybieramy `Enable Dependency Management`. Wtedy wszystkie zależności, które dodaliśmy w pliku pom.xml znajdą się automatycznie w naszym classpath (rys. 2)



Rys. 2 Java Build Path

Teraz już projekt powinien się budować, a instalacja w lokalnym repozytorium musi zakończy się pomyślnie :)

4. Czas teraz przejść do implementacji metody:

```
public void printClass(final JClass jClass, final String outputDir,
      final String lineSeparator, final String header) { ... }
```

która doda do wygenerowanej klasy (obiekt `JClass`) dodatkowy kod (metodę) `modifiedClass.addMethod(someMethod)` oraz np. jakiś komentarz `modifiedClass.setHeader(someComment)` Do wygenerowania mamy następujący kod:

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

Krok 1. Tworzymy sygnaturę metody (listing 4)

```
JMethod readMethod = new JMethod("read");
JModifiers readMethodModifiers = new JModifiers();
readMethodModifiers.makePublic();
readMethod.setModifiers(readMethodModifiers);
JParameter readMethoParameter = new JParameter(JType.DATA_INPUT_STREAM, "dis");
readMethod.addParameter(readMethoParameter);
readMethod.addException(new JClass("java.io.IOException"), "");
```

...zostawiam to bez komentarza :)

Krok 2. Tworzymy ciało metody

```
JField[] fields = modifiedClass.getFields();
  if( fields.length > 0) {
    for(JField field : fields) {
      readSourceCode.append(returnProperRe-
```

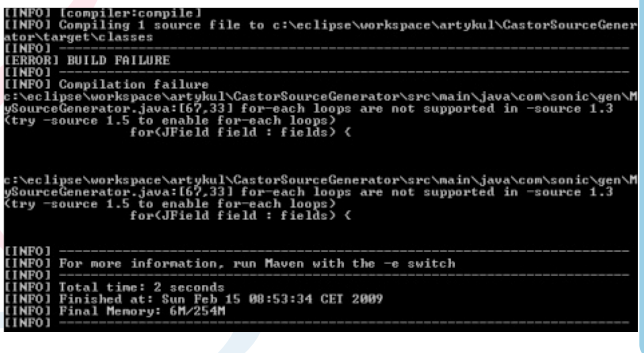
> *Nadszedł czas przetestowania naszej biblioteki*

```
adMethod(field));
      readSourceCode.append("\n");
    }
     readMethod.setSourceCode(readSourceCo-
de.toString());
  } else {
       readMethod.setSourceCode("super.re-
ad(dis);");
  }
```

...czyli iterujemy po wszystkich polach w klasie i tworzymy kod :)
Przykładowe implementacja metody `returnProperReadMethod(...)`

```
private String returnProperReadMethod(JField
field) {
  final String name = field.getName();
  final String type = field.getType().getName();
  if(type.compareTo("java.lang.String") == 0) {
    return "this." + name + " = dis.readUTF();";
  } else if(type.compareTo("int") == 0 || ty-
pe.compareTo("java.lang.Integer") == 0) {
    return "this." + name + " = dis.readInt();";
  } else if(type.compareTo("boolean") == 0 || ty-
pe.compareTo("java.lang.Boolean") == 0) {
    return "this." + name + " = dis.readBoole-
an();";
  } else if(type.compareTo("java.util.Date") ==
0) {
    return "this." + name + " = new java.util.Da-
te(dis.readLong());";
  } else if(type.compareTo("double") == 0 || ty-
pe.compareTo("java.lang.Double") == 0) {
    return "this." + name + " = dis.readDo-
uble();";
  } else {
    return "this." + name + " = (" + type+ ")Se-
rializer.deserialize( dis );";
  }
}
```

Tak stworzony kod dodajemy do naszej klasy `modifiedClass.addMethod(readMethod);`
Na koniec dodam, że należy pamiętać o dodaniu wpisu o imporcie klasy `modifiedClass.addImport("de.enough.polish.io.Serializer")` i implementacji metody `write(...)` !!!
Zagadka :) Czy projekt się zbuduje? :) Przeanalizujcie uważnie listing 5. Chodzi o linijkę:
`for(JField field : fields)`
próba kompilacji zakończy się takim błędem:



Na stronie Maven'a [7] możemy dowiedzieć się:
„...The default source setting is 1.3 and the default target setting is 1.1, independently of the JDK you run Maven with...”
Przechodzimy więc do konfiguracji wtyczki do Mavena, Compiler Plugin, edytujemy plik pom.xml i dodajemy:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plu-
gins</groupId>
      <artifactId>maven-compiler-plu-
gin</artifactId>
      <configuration>
        <source>1.5</source>
        <target>1.5</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

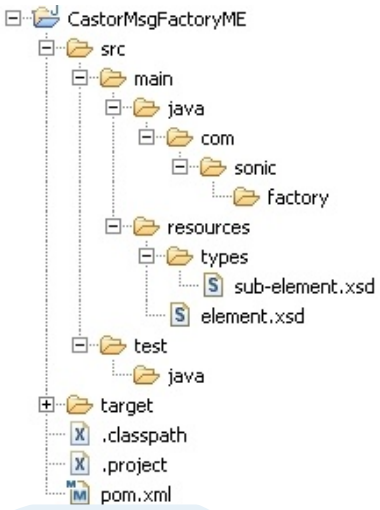
Należy pamiętać o właściwym ustawieniu w pliku castorbuilder.properties odpowiednich parametrów:

> *Generowanie plików będziemy uruchamiąć korzystając z Maven*

```
org.exolab.castor.builder.javaVer-
sion=1.4
org.exolab.castor.builder.jclassPrin-
terTypes=\
      com.sonic.gen.MySourceGenera-
tor,\
      org.exolab.castor.builder.prin-
ting.TemplateJClassPrinter
```

Nadszedł czas przetestowania naszej biblioteki, ale zanim to nastąpi, musimy mieć jakieś dane testowe. Musimy stworzyć przykładowy plik Schema i zaimplementować automat, który będzie nam generował gotowy kod. Zatem do dzieła:

Przechodzimy do projektu `CastorMsgFactoryME` i tworzymy katalog `resources` tu umieszczamy plik `element.xsd`, następnie tworzymy katalog `types` i tam umieszczamy plik `sub-element.xsd` (rys. 4). Przy edycji



plików schema warto mieć zainstalowany XML Schema Editor w Eclipse [8]. Na listingu 7 i 8 przedstawię przykładową zawartość obu plików.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema attributeFormDefault="unqualified"
    elementFormDefault="qualified"
    version="1.0"
    xmlns="http://com.sonic/element"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xmime="http://www.w3.org/2005/05/xmlmime"
    xmlns:es="http://com.sonic/types/complexTypes"
    targetNamespace="http://com.sonic/element">
  <xsd:import
    namespace="http://com.sonic/types/complexTypes"
    schemaLocation="types/sub-element.xsd"
  />
<xsd:element name="MyElement">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded"
        name="MySubElement"
        type="es:SubElementType" />
      </xsd:sequence>
      <xsd:attribute name="attr1" type="xsd:string"   />
      <xsd:attribute name="attr2" type="xsd:int"      />
      <xsd:attribute name="attr3" type="xsd:double"     />
      <xsd:attribute name="attr4" type="xsd:dateTime" />
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

> *Przedstawię jeszcze Dependency Graph w obu projektach*

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema attributeFormDefault="unqualified"
    elementFormDefault="qualified"
    version="1.0"
    xmlns="http://com.sonic/types/complexTypes"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xmime="http://www.w3.org/2005/05/xmlmime"
    targetNamespace="http://com.sonic/types/complexTypes">
  <xsd:complexType name="SubElementType">
    <xsd:attribute name="attr1" type="xsd:int"    />
    <xsd:attribute name="attr2" type="xsd:double" />
    </xsd:complexType>
</xsd:schema>
```

Użyjemy klasy `org.castor.anttask.CastorCodeGenTask`
Tworzymy plik build.xml. W zasadzie możemy usunąć katalogi `com/sonic/factory` bo nie będą nam potrzebne. Generowanie plików będziemy uruchamiąć korzystając z Maven'a. W fazie generowania źródeł `<phase>generate-sources</phase>` uruchomimy task w Ant przy pomocy wtyczki `<artifactId>maven-antrun-plugin </artifactId>` **[9]**.
Property o nazwie `compile_classpath` jest referencją do classpath w Maven. Wykorzystamy ją w pliku `build.xml`. Plugin ten uruchomi domyślny `task` z pliku `build.xml`

```
<project name="CastorMsgfactoryME" default="castor:gen:src" basedir=".">
```

można to obejść definiując w tagu `<ant>` coś takiego: `<target name="nazwa"/>`
Generowanie kodu odbywać się będzie przy pomocy zdefiniownego taska:

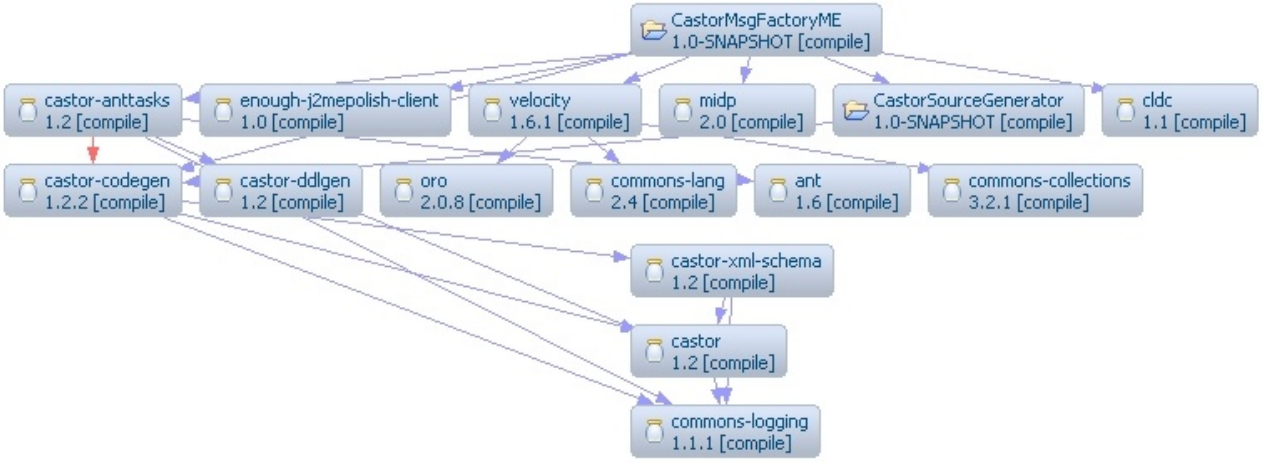
```
<taskdef name="castor-srcgen"
  classname="org.castor.anttask.CastorCodeGenTask"
classpathref="castor.class.path" />
```

.. i wywołanie:

```
<castor-srcgen   file="src/main/resources/element.xsd"    todir="${src.dir}"
package="${package.name}"          warnings="true"    nodesc="true"    nomar-
```

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-antrun-plugin</artifactId>
  <version>1.3</version>
  <executions>
    <execution>
      <id>generate-me-sources</id>
      <phase>generate-sources</phase>
      <configuration>
        <tasks>
          <property name="compile_classpath"
refid="maven.compile.classpath"/>
          <ant antfile="build.xml" dir="${basedir}" />
        </tasks>
      </configuration>
      <goals>
        <goal>run</goal>
      </goals>
    </execution>
  </executions>
</plugin
```

> *XML wyrósł na niekwestionowany standard wymiany danych w sieci WWW*




```
shal="true"/>
```

gdzie:

```
<property    name="package.name"    va-
lue="com.sonic.dto"/>
<property name="src.dir" value="src/ma-
in/java"/>
<path id="castor.class.path">
  <path path="${compile_classpath}"/>
</path>
```

Domyślnie CastorCodeGenTask wygeneruje kolekcje zgodne z Java 1.1, jeżeli chcielibyśmy, z jakichkolwiek względów wymusić zgodność z Java 1.2 to należy dodać atrybut `types` a jego wartość ustawić na `j2`.

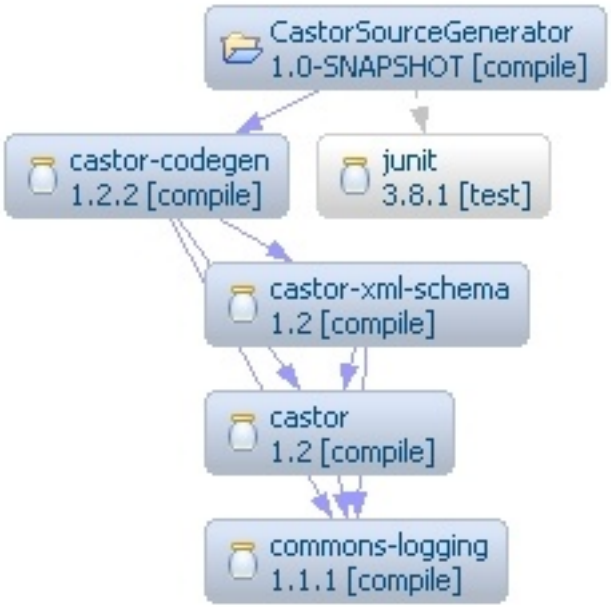
Atrybut `nomarshal` [10] ustawiliśmy na true aby poinformować generator, aby nie tworzył metod do marshalingu/unmarshalingu.

Na koniec przedstawię jeszcze Dependency Graph w obu projektach (rys. 7 i rys. 8):

Artefakty `enough-j2mepolish-client.jar`, `midp.jar` oraz `cldc.jar` znajdują się w katalogu `/j2me-polish2.0.7/lib/`

Przykład instalacji artefaktów to lokalnego repozytorium:

```
mvn install:install-file -Dgro-
upId=javax.midp       -Dartifac-
tId=midp          -Dversion=2.0
-Dpackaging=jar           -Dfi-
le=/path/to/file
```



Linki

1. *http://maven.apache.org/*
2. *http://ant.apache.org/*
3. *http://m2eclipse.codehaus.org/*
4. *http://dist.codehaus.org/castor/1.2/castor-1.2-src.zip*
5. *http://java.sun.com/javame/reference/apis/jsr139/*
6. *http://java.sun.com/products/archive/jdk/1.1/index.html*
7. *http://maven.apache.org/plugins/maven-compiler-plugin/*
8. *http://wiki.eclipse.org/index.php/Introduction_to_the_XSD_Editor*
9. *http://maven.apache.org/plugins/maven-antrun-plugin/plugin-info.html*
10. *http://www.castor.org/srcgen-anttask.html*

Obsługa XML w Javie - biblioteka XStream

Marek Kapowicki

Format XML

XML (Extensible Markup Language, w wolnym tłumaczeniu *Rozszerzalny Język Znaczników*) to uniwersalny język formalny przeznaczony do reprezentowania różnych danych w ustrukturalizowany sposób. i znacząco przyczyniło się do popularności tego języka. XML wyrósł na niekwestionowany standard wymiany danych w sieci WWW. Język i dokumenty XHTML pozwalają skutecznie opisywać zawartość stron WWW; tymczasem XML pozwala na opisywanie dowolnych danych, byle miały jakąś zdefiniowaną strukturę. Do tego zamiast udostępniać ograniczony zestaw znaczników do oznaczenia elementów dokumentu, XML pozwala na definiowanie własnych słowników, dla każdej dziedziny z osobna. Dokładny opis standardu XML nie jest celem tego artykułu. Chętnych do zapoznania się z specyfikacją standardu odsyłam do literatury i sieci.

Przykłady zastosowania

W praktyce dokumenty XML są bardzo często stosowane i mają szeroki zakres zastosowań. Jako programiście często zdarza mi się otrzymać plik XML z danymi, które trzeba zapisać w bazie. Mogą to być wszelkiego rodzaju informacje o ludziach, książkach, instytucjach itp.. Dokument trzeba umieć sparsować i otrzymane obiekty zapisać w bazie. Dane zawarte w dokumencie XML mogą być przekształcone na dokument HTML, PDF , czy dowolny inny format tekstowy przy użyciu mechanizmu XSLT. Warto więc umieć tworzyć dokumenty XML z posiadanych danych, chociażby do wygenerowania pliku PDF. XML jest wykorzystywany do definiowania plików konfiguracyjnych – z zastosowaniem tym spotkało się na pewno wielu programistów JAVA.

Obsługa XML w Javie

Popularne mechanizmy

Myślę, że dwa najpopularniejsze API do obsługi XML z poziomu Javy to SAX i DOM. Służą one do analizy składniowej dokumentów. Pierwszy przetwarza dokument i za każdym razem, gdy napotka jakiś znacznik, komentarz, fragment tekstu, lub jakiś inny element XML odwołuje się do kodu programu w celu zasygnalizowaniu zdarzenia. Wtedy nasz kod może wykonać odpowiednie zadanie. Używając tego API mamy dostęp tylko do aktualnie przetwarzanego elementu. Interface SAX jest szczególnie przydatny przy analizowaniu dużych plików. DOM po sparsowaniu pliku XML udostępnia w pamięci kompletną reprezentację dokumentu w postaci drzewa. Mimo że opisywane API nie są tematem artykułu to zachęcam do zapoznania się z nimi.

Alternatywne podejścia

Odmiennym podejściem jest mapowanie (wiązanie danych). Zamiast pracować z elementami i atrybutami, posługujemy się obiektami Javy co znacznie ułatwia pracę z XML. Podejście powinno wydać się znajome czytelnikom, którzy spotkali się z Hibernatem (wiążącym tabele z bazy danych z obiektami Javy).

Częściowo, aby zaznaczyć różnicę w wiązaniu danych zastosowano nieco odmienną terminologię – zamiast terminów „analiza składniowa” (parsing) czy „serializacja” (serializing) stosuje się termin szeregowanie (marshaling) - konwersja obiektów XML na obiekty Javy. Proces odwrotny to rozszeregowywanie (unmarshaling) tj. zamiana klas Javy na obiekty XML. Narzędziem opartym na opisanym podejściu jest biblioteka XStream, którą postaram się przybliżyć w niniejszym artykule. Biblioteka wykorzystuje mechanizm refleksji do w celu zidentyfikowania pól, które mają być zapisane. Jest bardzo prosta (nie wymaga określania XML Schema) i znacznie ułatwia obsługę XML z poziomu Javy. Więcej informacji o bibliotece można znaleźć na stronie http://xstream.codehaus.org/.

Użycie biblioteki XStream

Dodanie biblioteki do projektu

Jeśli chcemy korzystać z XStream, musimy dodać ją do swojego projektu. W tym celu należy pobrać ze strony projektu najnowszego jara zawierającego bibliotekę – w czasie pisania arty-

> *odwzorowanie odbywa się po nazwach*

kułu była to wersja 1.3.1. Jeśli nie będziemy rozwijać XStreama, a tylko z niego korzystać radzę pobrać wersję binarną. Pobrany plik należy rozpakować - interesujące nas archiwum znajduje się w podkatalogu lib (plik xstream-1.3.1.jar). Następnie należy dodać ścieżkę do pobranego jara do Java Build Path naszego projektu. Jeśli ktoś, podobnie jak ja używa Mavena do tworzenia aplikacji, powinien dodać do pliku pom.xml następującą zależność :

```
<dependency>
  <groupId>
    com.thoughtworks.xstream
  </groupId>
  <artifactId>xstream</artifactId>
  <version>1.3.1</version>
</dependency>
```

Biblioteka XStream obsługuje adnotacje i w związku z tym do korzystania z niej potrzebna jest Java 1.5 lub nowsza.

Pierwsze użycie

W celu wykonania pierwszej prostej konwersji stwórzmy w Javie Beana:

```
public class Person  {
      private String name;
      private String surname;
      private Date birthday;
      //gettery i settery
}
```

Teraz użyjemy opisywanej biblioteki do stworzenia XML

Stworzymy i wypełnijmy obiekt Person (wypełniłem swoimi danymi) i uruchomiamy napisaną metodę. W jej wyniku ujrzymy na ekranie:

```
public String person2Xml(Person person) {
      XStream mapping=new XStream(new DomDriver());
      String xml=mapping.toXML(person);
      return xml;
}
```

Jak widać odwzorowanie odbywa się po nazwach. XStream mapuje całą nazwę klasy (wraz z pakietem, w którym się znajduje) czego nie zawsze chcemy. Również stworzona data jest w nietypowym formacie. Na szczęście możemy wpływać na odwzorowanie np. poprzez definiowanie aliasów (najlepiej przy użyciu adnotacji). Do zamiany otrzymanego XML z powrotem na obiekt Javy używamy :

```
<pl.marek.Person>
  <name>Marek</name>
  <surname>Kapowicki</surname>
  <birthday>
    1983-09-28 00:00:00.0 CET
  </birthday>
</pl.marek.Person>
```

```
public Person xml2Person(String xml){
   XStream mapping =
     new XStream(new DomDriver());
   return (Person) mapping.
     fromXML(xml);
}
```

Zaawansowane wykorzystanie biblioteki

W celu pokazania możliwości biblioteki stworzyłem program służący do przechowywania informacji o filmach. Przechowywane dane to tytuł, opis, gatunek filmu, lista występujących aktorów, rok powstania, informację o reżyserze, okładka oraz link do strony WWW opisującej wybrany film. Aplikacja umożliwia tworzenie pliku XML, na podstawie danych wprowadzonych w Javie, oraz wykonanie operacji przeciwnej – skonstruowanie obiektów z wybranego pliku.

Aplikacja stworzona jest przy użyciu Mavena.

Najlepiej korzystać z adnotacji

Uruchamiamy ją wykonując z linii poleceń *mvn install*. Składa się z kilku pakietów, które po krótce opisze (więcej informacji znajduje się w javadocach)

- `pl.marek.beans` – pakiet zawierający beany, które będą mapowane na dokumenty XML tj.
  - `Film` – do reprezentacji pojedynczego filmu
  - *`Person`* – do przechowywania informacji o osobach: reżyserach, aktorach
  - `Films` – klasa zawierająca pełną informację o filmach, właścicielu itp. Objekty tej klasy są przekształcane w pliki xml – które są właściwym wynikiem działania programu
  - Wszystkie beany dziedziczą po klasie abstrakcyjnej *`BaseXML`*, dzięki czemu możliwe jest mapowanie wszystkich plików przy pomocy tego samego mechanizmu(jeśli do sterowania mapowaniem używamy tylko adnotacji, a nie udostępnianych przez bibliotekę metod)
- `pl.marek.enums` – pakiet zawierający enuma z możliwymi kategoriami filmów
- `pl.marek.xstream` – zawiera właściwą część aplikacji odpowiedzialną za mapowanie danych
  - `XMLMappingInterface` – interface służacy do mapowania Beanów dziedziczących po klasie *`BaseXML`*. Udostępnia cztery metody:
  - **`public String`** `java2xml(BaseXML base)` – zamienia objekt na napis będący zawartością pliku XML
  - **`public void`** `java2xmlFile(BaseXML base,String fileName)` – zamienia objekt na plik XML.
    Metody tworzące XML są uniwersalne I mogą być używane do wszystkich klas dziedziczących po *`BeanXML`* . Podczas wykonywania konwersji wykrywany jest typ przetwarzanego obiektu i wczytywane są odpowiednie adnotacje.
  - `BaseXML xml2java(String xml,Class<?` **`extends`** `BaseXML> className)` – zamienia napis (zawartość pliku XML) na objekt
  - **`public`** `BaseXML xmlFile2Java(String fileName,Class<?` **`extends`** `BaseXML> className)` - zamienia plik XML na objekt
    Wywołując metody konwerujące plik xml na objekty Javy należy podać typ tego objektu
  - `XMLMapping` - klasa implementująca powyższy interface
  - `PersonXMLMapping` – służy do mapowania objektów klasy Person. Stworzona w celu pokazania, że używając udostępnianych przez XStream metod zamiast adnotacji narażamy się na problemy i nie możemy korzystać z jednego uniwersalnego mechanizmu mapowania
  - `PersonConventer` – konwerter służący do ręcznego odwzorowania klas na pliki XML

Wywołania stworzonych metod znajdują się w katalogu z testami JUnit. Po uruchomieniu testów powinniśmy zobaczyć na dysku dwa nowe pliki author.xml i films_list.xml.

Patrząc w kod beanów zauważymy, że mapować możemy proste typy Javy, obiekty własnych klas np. składowa director w klasie Film oraz kolekcje – lista filmów w klasie Films

Zarządzanie mapowaniem

Korzystając z opisywanej biblioteki możemy wpływać na mapowanie. Najlepiej korzystać z adnotacji, umieszczając je przy dowolnych polach w beanach. Należy poinformować konwerter (obiekt klasy XStream) o tym, że powinien przetwarzać adnotacje. W tym celu wywołujemy metodę

`xStream.processAnnotations(Class className)` – adnotacje z klasy className będą wczytane przez konwerter. Jeśli używamy adnotacji przy tworzeniu pli-

> *Jest to proste narzędzie ułatwiające obsługę plików xml z poziomu Javy.*

ku XML to przy jego ponownej zamianie na obiekty musimy również poinformować konwerter o adnotacjach. W przeciwnym razie XStream będzie używał odwzorowania po nazwie i nie będzie w stanie wykonać zamiany.

Przydatne adnotacje:

- `@XStreamAlias(„nazwa”)` może być stosowana zarówno przy nazwach klasy, jak i nazwach pól składowych. Określa na jaką nazwę ma być mapowane wskazane pole. Przykłady użycia znajdują się w klasie *Film*
- `@XStreamImplicit(itemFieldName="nazwa")` używane przy mapowaniu kolekcji. Określa na jaką nazwę mają być mapowane elementy kolekcji. Przykład użycia w klasie *Films*
- `@XStreamOmitField` adnotacja umieszczane przy polach, które nie będą ulegały mapowaniu

Jeśli nie chcemy używać adnotacji możemy korzystać z metod udostępnianych przez konwerter. Przy tworzeniu pliku author.xml użyłem np. xStream.alias("director", Person.class) w celu określenia aliasu (zastępowania nazwy klasy Person napisem direktor). Uważam, że nie jest to podejście ułatwiające używanie biblioteki i „zaciemnia” kod. Dlatego zachęcam do używania adnotacji, dzięki którym możemy w prosty sposób zarządzać mapowaniem.

Konwertery

Jeśli przy mapowaniu nie chcemy używać standardowych mechanizmów wbudowanych w bibliotekę XStream, możemy własnoręcznie napisać konwerter. W tym celu musimy stworzyć klasę rozszerzającą `com.thoughtworks.xstream.converters.Converter`, która implementuje dwie metody:

- **`public boolean`** `canConvert(Class clazz)` – sprawdza jakie klasy mogą być konwertowane
- **`public void`** `marshal(Object value, HierarchicalStreamWriter writer,MarshallingContext context)` – metoda, która jest wywoływana przy zamianie obiektu Java na xml:
  - *value* – konwerowany obiekt
  - *writer* – obiekt używany do zapisywania danych
  - *context* – aktualny kontekst

  Metoda użyta jest w napisanym przeze mnie konwerterze PersonConverter – służącym do ręcznej zamiany obiektów klasy Person

  W metodzie sprawdzamy, jakie pole jest

```
DateFormat df=DateFormat.
  getDateInstance(DateFormat.MEDIUM);
if(author.getBirthday() != null) {
  writer.startNode("birthday");
  writer.setValue(df.format(
    author.getBirthday()));
  writer.endNode();
}
```

  aktualnie przetwarzane i określamy co należy zrobić przy przetwarzaniu tego pola. W przytoczonym przykładzie zamieniamy datę na format rrrr-mm-dd

- **`public`** `Object unmarshal(HierarchicalStreamReader reader,UnmarshallingContext context)` metoda wykonywana przy zamianie pliku xml na objekt Javy.

Podsumowując

Mam nadzieję, że udało mi się zachęcić do zapoznania z biblioteką XStream. Jest to proste narzędzie ułatwiające obsługę plików xml z poziomu Javy. Biblioteki jest łatwa w użyciu i nie powinna przysporzyć nikomu problemów. Dzięki obsłudze adnotacji i możliwości definiowania konwerterów, mamy możliwość zarządzania mapowaniem.

TeamCity: pre-tested commit.

Paweł Zubkiewicz

Czyli w jaki sposób, prosto i skutecznie, rozwiązać problem commit'owania niedziałającego kodu do repozytorium, a tym samym zwiększyć szybkość pracy całego zespołu.

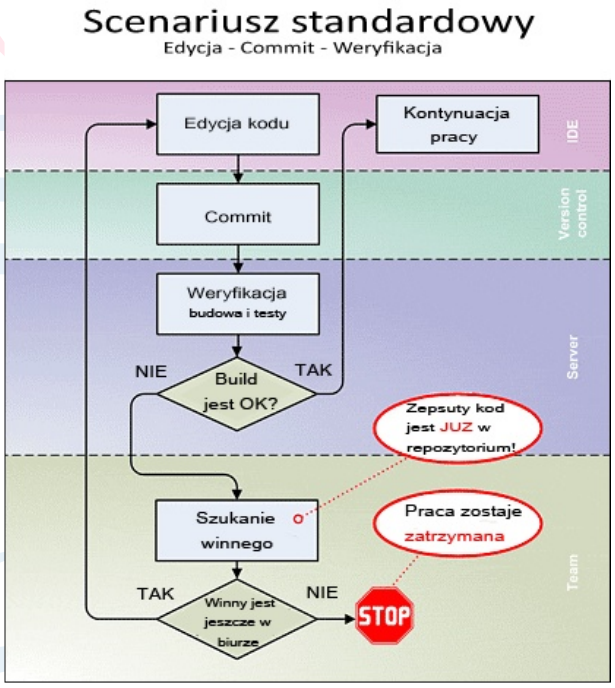
W obecnych czasach trudno jest, tworząc komercyjne rozwiązania, wyobrazić sobie projekty, które nie stosują się (czasem nawet nieświadomie) do zbioru praktyk Fowlera, powszechnie znanych pod nazwą *Continuous Integration*. Najważniejszym celem tych praktyk jest redukcja czasu (a co za tym idzie także i kosztów) wdrażania zmian do projektu poprzez wczesną i częstą (ich) integrację. Zmiany te są efektem pracy wielu programistów w zespole. Idea *Continuous Integration* liczy sobie blisko 10 lat. Przez ten czas powstało wiele narzędzi wspomagających nas – programistów - w ciągłym wdrażaniu tej idei w życie (we własnych projektach).

Wśród rozmaitych aplikacji wspierających idee *Continuous Integration* bardzo ważne miejsce zajmują serwery integracji, w skrócie *serwery CI*. Pozwalają one na automatyczne budowanie projektów, uruchamianie testów oraz bardzo szybkie informowanie o napotkanych błędach i awariach. Są to oczywiście tylko ich podstawowe możliwości. Najbardziej zaawansowane systemy oferują dużo więcej interesujących i użytecznych funkcji.

Jednym z takich zaawansowanych narzędzi, które na przestrzeni ostatniego roku zyskało sobie rzesze użytkowników, jest TeamCity. Niewątpliwym tego powodem jest unikatowa funkcjonalność, jaką oferuje swoim użytkownikom: ***pre-tested commit***.

Aby w pełni docenić jej zalety, na początku przypomnę pokrótce standardowy scenariusz codziennej pracy programisty:

1. Update kodu projektu z repozytorium;
2. Zmiany w paru klasach (programowanie właściwe :-)
3. [opcjonalnie] Weryfikacja zmian – uruchomienie testów jednostkowych;
4. Commit zmian do repozytorium kodu (poprzedzony update'em – o to dba samo repozytorium);
5. Automatyczne pobranie przez serwer CI najnowszej wersji źródeł z repozytorium i rozpoczęcie budowy projektu;
6. [opcjonalnie] Sprawdzenie czy serwer CI poprawnie zbudował kod z wprowadzonymi zmianami



Wszyscy tak pracujemy. Można więc zapytać: cóż złego jest w takim trybie pracy, tym bardziej, że jest on tak powszechnie stosowany? Głównym problemem (*smell'em*) jest to, że nawet jeśli programista uruchomił testy jednostkowe (które zakończyły się sukcesem) przed commit'em swojego kodu, to nie mamy żadnej gwarancji, że ten sam kod na serwerze CI też się skompiluje i przejdzie testy. Zapewne każdemu z nas przydarzyła się kiedyś taka sytuacja. Przyczyn jej może być wiele. Najczęstszą są różnice pomiędzy środowiskiem developerskim a tym, gdzie znajduje się serwer

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

CI. Mogą to być „zahardkodowane" ścieżki, różne wersje bibliotek czy inne niejawne założenia, jak np. dostęp do zasobów zdalnego dysku. Bywa, że do jednego build'a trafią zmiany z paru commit'ów, które wprawdzie osobno działały, ale razem już nie chcą. Czasem powód jest bardziej trywialny: nie wszystkie zmienione klasy zostały wysłane do repozytorium, co powoduje, że kod najzwyczajniej w świecie się nie kompiluje. W tym momencie zaczyna się zwykle szukanie winnego w zespole: tego, który umieścił *crap* w repozytorium. O ile znalezienie go nie stwarza problemu, to gorzej bywa jednak z poprawieniem samego błędu. W najgorszym przypadku praca całego zespołu zostaje zablokowana i to wszystko z powodu jednego niefortunnego commit'u.

Pomysłów na rozwiązanie tego problemu jest co najmniej kilka: od wprowadzenia wysokiej dyscypliny wśród programistów po skomplikowane rozwiązania SCM jak *stable trunk* (które wymagają od programistów nie mniej dyscypliny, a przy tym dużego nakładu pracy).

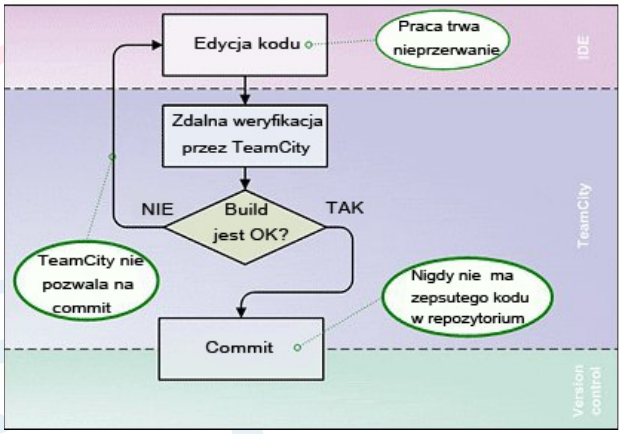
Inżynierowie z JetBrains wymyślili unikatowe rozwiązanie tego problemu. Zmodyfikowali oni powyższy scenariusz pracy w taki sposób, aby niedziałający kod nie mógł znaleźć się w repozytorium, a przy tym nie był uciążliwy dla jego użytkowników. Do tego właśnie służy funkcja *pre-tested commit,* którą między innymi oferuje TeamCity.

Scenariusz pracy programisty z TeamCity wygląda następująco:

1. Update kodu projektu z repozytorium;
2. Dokonanie zmian w paru klasach (programowanie właściwe :-)
3. Wysłanie zmienionych plików do TeamCity (za pomocą plugin'a w IDE);
4. Rozpoczęcie procesu budowania projektu przez TeamCity z wykorzystaniem plików przesłanych przez programistę oraz plików z repozytorium;
   a) umieszczenie w repozytorium wysłanych plików, jeśli build zakończył się sukcesem;
   b) brak zmian w stanie repozytorium, jeśli build się nie udał;
5. Poinformowanie programisty o stanie jego build'a.

Zmodyfikowany scenariusz TeamCity

Edycja - Weryfikacja - Commit



Jak widać, w takim trybie pracy w repozytorium znajdzie się tylko sprawdzony kod. Jak bardzo, zależy to już tylko od naszych testów. Mamy jednak gwarancję, że błąd jednego programisty nie zablokuje całego zespołu i nie spowoduje przestojów w pracy.

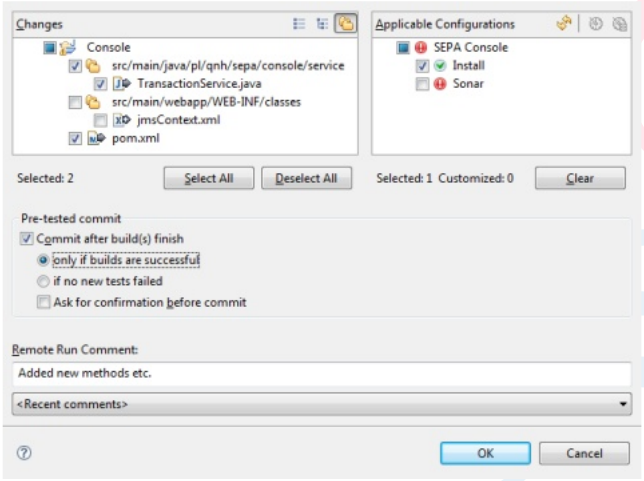
W praktyce korzystanie z funkcji *pre-tested commit* jest bardzo proste i w pewnym stopniu przypomina korzystanie z wtyczek repozytoriów kodu. Poza działającą instancją serwera TeamCity wymagana jest wtyczka do IDE. W chwili pisania tego artykułu wspierane były środowiska IntelliJ IDEA oraz Eclipse. W tym ostatnim plugin dostarcza cztery nowe zakładki, z których najważniejsza to *Remote Run*. To właśnie ona pozwala na wykorzystanie funkcji *pre-tested commit*. Można z niej zdalnie uruchomić build z „prywatnymi" zmianami programisty, który nazywany jest *Personal Build* (prywatny build).

Na zakładce *Remote Run* można wybrać:

> *Mapowanie jest gotowe...*

- pliki ze zmianami *wychodzącymi;*
- *konfiguracje builda* (uprzednio stworzoną na serwerze TeamCity), w której zmienione pliki mają być wykorzystane.

Zaznaczając opcje *Commit after build(s) finish*, pliki ze zmianami zostaną wysłane do repozytorium tylko i wyłącznie po pomyślnym „przejściu" build'a. I to jest właśnie sedno pomysłu *pre-tested commit* – do repozytorium trafią tylko te pliki, które już raz przeszły testy na serwerze CI.



Warto zwrócić uwagę, że Personal Build może być wykorzystany do zwiększenia produktywności programisty (czy wręcz jako alternatywa dla zakupu szybszego komputera dla niego). Korzystając z niej, programista może zdalnie, na serwerze, uruchamiać swoje testy (budować projekt) i w tym samym czasie nadal pracować nad dalszą częścią swojego rozwiązania. Tryb jego pracy nie jest przerywany przez oczekiwane na zakończenie lokalnego build'a, który często bywa czaso- i zasobochłonny.

Producentem TeamCity jest firma JetBrains, znana przede wszystkim jako producent IntelliJ IDEA. TeamCity to produkt komercyjny, pomimo to dostępny za darmo w wersji Professional. Wersja ta charakteryzuje się ograniczeniem ilości użytkowników i konfiguracji projektów (build'ów) do 20 oraz *Build Agents* do 3. Ilość *Build Agents* określa maksymalną możliwą ilość równolegle trwających procesów budowy projektów. Powyższe ograniczenia nie są jednak na tyle poważne, aby mogły przeszkodzić w pomyślnym wdrożeniu tego produktu w małych i średnich projektach zupełnie za darmo.

Podsumowując TeamCity wydaje się być nowatorskim produktem w świecie serwerów CI. Rozwiązanie to może w prosty sposób przyczynić się do ograniczenia problemów związanych z ciągłą integracją, na które napotykają się wręcz codziennie zespoły programistów tracąc czas i pieniądze. Wykorzystanie funkcji *pre-tested commit* jest łatwe i nie wymaga od programisty drastycznych zmian w jego codziennym trybie pracy, dzięki czemu jest tanie i realne do wprowadzenia w życie (nie napotkamy fali sprzeciwów w zespole).

VRILTEK.COM

Logotypy

Portale

Plakaty

Wizytówki

Strony WWW

DTP

Sklepy internetowe

Ulotki

Banery



Express killers, cz. III

Damian Szczepanik

W kolejnym odcinku zastanówmy się, jak maszyna wirtualna radzi sobie z drukowaniem referencji, która wskazuje na nic, czyli na null. Oto krótki kawałek kodu, który drukuje nazwę klasy obiektu pobranego z kolekcji:

```
import java.util.ArrayList;
import java.util.List;

public class PrintNull {

  private static PrintNull o;

  public static void main(String[] args) {
    List<PrintNull> list = new ArrayList<PrintNull>();
    list.add(o);
    for (PrintNull i : list) {
      System.out.println(i.toString());
    }
  }

  public String toString() {
    return (this == null) ? "<null>" : super.toString();
  }
}
```

Co zostanie wydrukowane na wyjście?
- null
- <null>
- za każdym razem inna wartość
- żadne z powyższych

Dziś bardzo krótki kawałek kodu i pytanie, co on robi – czy w ogóle coś robi, czy jest wynikiem poprawiania defektu poprzez usunięcie kawałka kodu, a nie całego bloku.

```
synchronized(obj)
{
}
```

Recenzja GroovyMag kwiecień 2009

Krzysztof Konwisarz

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

Get Rich Quick with Flex and Grails (Part I)

Autorzy: Jeremy Anderson i BJ Allmon

Artykuł demonstruje w jak prosty sposób można połączyć framework Adobe Flex z Grails otrzymując wydajne narzędzie do tworzenia bogatych aplikacji internetowych (RIA). Autorzy prowadzą nas za rękę przez proces tworzenia aplikacji pokazując kluczowe elementy niezbędne dla współpracy dwóch technologii: dostęp do danych poprzez usługę RemoteService i wiązanie danych po stronie serwera i klienta. Efektem jest stworzenie prostej aplikacji internetowej do zarządzania kontaktami, która zostanie rozbudowana w drugiej części artykułu.

Groovy under the Hood – How your Groovy Class Becomes Java Bytecode (Part II)

Autor: Kirsten Schwark

Druga część tekstu opisującego działanie Groovy i jego relację względem Javy. W tej odsłonie dowiadujemy się jak stworzono dynamiczny język programowania jakim jest Groovy. Poznamy podstawy jego implementacji: mechanizm Metaobject Protocol (MOP) odpowiadający za elastyczność języka oraz Call Site Caching w dużej mierze usprawniający jego działanie.

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

Sumatra: Testing JavaScript from Groovy

Autor: Scott Vlaminck

Sumatra to framework stworzony do testowania kodu JavaScript w Grails. Zamienia on kod JS na kod Groovy, umożliwiając jego proste testowanie. Pozwala na ładowanie zewnętrznych klas, uruchamianie kodu JS przekazywanego z Groovy w postaci stringów i testowanie go przy użyciu zwykłych asercji. Możliwe jest także testowanie kodu, który wykorzystuje frameworki JavaScript (w chwili obecnej obsługiwany jest jedynie Prototype).

What's New in Groovy 1.6?

Autor: Guillaume Laforge

Jak sama nazwa wskazuje, w artykule omówiona została długa lista zmian wprowadzonych w Groovy 1.6. Pełen tekst można znaleźć tutaj: *http://www.infoq.com/articles/groovy-1-6*.

New GORM Features in Grails 1.1 (Part 2 of 2)

Autor: Bashar Abdul-Jawad

Ta część przeglądu nowych funkcji Grails 1.1 zawiera omówienie: ładowania obiektów tylko do odczytu, dodawania domyślnego porządku sortowania do klas domenowych, zmian w dynamicznych finderach, nowych opcjach mapowania i ładowania obiektów z bazy danych partiami (*batch fetching*).

Plugin Corner – JavaScript validation plugin

Autor: Dave Klein

W cotygodniowym kąciku tym razem Javascript validation plugin – wtyczka dająca możliwość walidacji danych po stronie klienta. Ograniczenia zawartości pól formularza pobiera z klas domenowych, wspiera internacjonalizację, a do uruchomienia potrzebuje jedynie dodania kilku linijek kodu. Plugin jest w dość wczesnej fazie rozwoju, więc niektóre jego funkcjonalności są ograniczone.

Express killers, cz. III - odpowiedzi

Damian Szczepanik

Przykład pierwszy:

Do kolekcji został dodany obiekt null, zatem taki też zostanie z niej pobrany w pętli. Próba wywołania metody na referencji, która nie wskazuje na żaden, obiekt skończy się rzuceniem wyjątku. Bezpieczniejsze byłoby tutaj użycie instrukcji:
System.err.println(i);

co spowoduje wypisanie na wyjście „null" zamiast rzucenie wyjątku. Przesłonięcie metody toString() niczego w tym przypadku nie zmienia. Nie ma też znaczenia sprawdzenie, czy this nie jest null (wszakże kiedy jest to prawdą?).

Przykład drugi:

Powyższy kawałek kodu odnalazłem niedawno w jednym ze źródeł i zastanawialiśmy się razem, czy ten blok jest zamierzony, czy jego zawartość została usunięta podczas kolejnych zmian w kodzie.
Na pierwszy rzut oka kod może wydawać się bez znaczenia, gdyż w bloku nie ma nic, co można by synchronizować. To prawda, ale jeśli przyjrzeć się szerszemu kontekstowi, to można dostrzec jego potrzebę, jeśli inne wątki zakładają mutex na referencji do obiektu tej klasy. Oto przykład kodu, który zachowuje się inaczej, jeśli blok w metodzie printTrue() zostanie usunięty, a inaczej, gdy będzie pozostawiony.

```
public class Sync implements Runnable {

  private static final Object obj = new Object();
  private final boolean id;

  public Sync(boolean id) {
    this.id = id;
  }

  public static void main(String[] args) {
    new Thread(new Sync(false)).start();
  }

  public void run() {
    System.err.println("start:" + id);
    if (id) {
      printTrue();
    } else {
      printFalse();
    }
    System.err.println("end:" + id);
  }

  public void printTrue() {
    // zakomentuj te linijki i sprawdz, czy wynik będzie identyczny
    synchronized (obj) {
    }
  }
```

Rozjazd

> *Kod jest co najmniej dziwny i kwalifikuje się do wyrzucenia*

```
public synchronized void printFalse() {
  synchronized (obj) {
    try {
      new Thread(new Sync(true)).start();
      wait(2000);
      System.err.println("print");
    } catch (InterruptedException ex) {
      ex.printStackTrace();
    }
  }
}
```

Oczywiście wywołanie wait(2000) nie jest eleganckie i może zadziałać dla różnych implementacjach różnie (choć przy wartości 2 sekund efekt powinien być taki sam). Reasumując kod jest co najmniej dziwny i kwalifikuje się do wyrzucenia. Jednakże jego beztroskie usunięcie może mieć niepożądany skutek w działaniu programu.

Rozwarstwienie

Mariusz Sieraczkiewicz

Dlaczego o warstwach?

O warstwach, architekturze dwuwarstwowej, trójwarstwowej, wielowarstwowej słyszał prawie każdy programista. Jednak w wielu rozmowach na temat programowania odnoszę wrażenie, że jest to zagadnienie traktowane marginalnie, jak coś co ma niewielki wpływ na codzienną pracę programisty. Mimo że temat jest związany z architekturą systemu, to bez względu na rolę, jaką pełnisz w projekcie wpływa on lub może wpływać na to, co robisz. Ten artykuł mówi o tym, jak w praktyce wykorzystać ten koncept, jak jego zrozumienie może wpłynąć na polepszenia Twojego kodu i jak jednocześnie być pragmatycznym w tej kwestii.

O co właściwie chodzi?

Koncept warstw powstał, tak jak wiele różnych idei, po to by ułatwiać życie. W tym przypadku chodzi o ułatwienie tworzenia systemów informatycznych. Aby zorganizować strukturę systemu, warto wydzielić pewne logiczne części powiązanych ze sobą klas, które mają wspólną odpowiedzialność. I tak w dużej części systemów możemy wyróżnić m.in.:

- interfejs użytkownika - odpowiadający za interakcję z użytkownikiem, najczęściej poprzez odpowiednie widoki lub okienka,

- dziedzinę - główne dane systemu, przetwarzanie aplikacji, algorytmy, obliczenia, cykl życia systemu,

- komunikację ze światem zewnętrznym - dostęp do danych, sposób zapisu i odczytu danych w sposób trwały i komunikacja z systemami zewnętrznymi.

To tylko przykładowy podział warstwowy. Warstw może być więcej i mogą być inaczej zdefiniowane.

Pierwszym wyróżnikiem wynikającym ze stosowania warstw jest podzielenie systemu na logiczne części z jasno wydzieloną odpowiedzialnością. Części te są ortogonalne do funkcji systemu.

Drugim wyróżnikiem są jednoznacznie zdefiniowane relacje między warstwami. Tak jak cebule mają warstwy, tak i systemy informatyczne mają warstwy. Im bardziej zewnętrzne, tym bliższe końcowemu klientowi systemu. Warstwy mają zatem ustaloną kolejność i pełnią dla siebie funkcje usługowe.

Interfejs użytkownika

Dziedzina

Dostęp do danych

Z wymienionych powyżej warstw, najbardziej zewnętrzną warstwą jest interfejs użytkownika, który organizuje interakcje z użytkownikiem – odpowiada za pobieranie i wyświetlanie danych oraz za logiczną organizację widoków. Konkretne przetwarzanie w systemie jest delegowane do klas z warstwy dziedziny, gdyż to ona odpowiada za główne funkcje aplikacji (w oderwaniu od interfejsu). Zaś ostatecznie wszelkie operacje zapisu, odczytu danych lub komunikacji z systemami zewnętrznymi w warstwie dziedziny są delegowane do warstwy dostępu do danych. Tylko warstwy następujące po sobie bezpośrednio mogą się ze sobą komunikować, przy czym warstwa wyższa korzysta z warstwy niższej.

Jakie są główne korzyści wynikające z korzystania z warstw?

- Warstwy są sposobem na podzielenie systemu na wysokopoziomowe logiczne części – łatwiej nimi zarządzać i łatwiej je zrozumieć, gdyż każda z nich ma wyraźnie wydzieloną odpowiedzialność.

- Każda warstwa ma charakterystyczną dla siebie budowę i zestaw interfejsów, które należy zaimplementować.

- Warstwy można traktować jako niezależne całości w dużym stopniu niezależne od po-

Czy warto stosować warstwy?

zostałych.

- Komponenty z danej warstwy mogą być ponownie używane w innych aplikacjach o tej samej strukturze warstwowej, co sprzyja tworzeniu szkieletów aplikacyjnych.

- Niezależne zespoły mogą pracować nad rozwojem danej warstwy systemu.

- Komponenty z różnych warstw mogą być niezależnie tworzone, wdrażane, utrzymywane i aktualizowane.

Oczywiście są również i wady.

- Wiele warstw powoduje, że poważniejsze modyfikacje funkcji systemu wymuszają kaskadowe zmiany w wielu warstwach.

- Warstwy mogą spowodować spadek wydajności systemu.

Jeśli tworzysz jednoosobowo pewną aplikację lub masz wpływ na architekturę systemu, wtedy warstwy pomagają Ci łatwiej opanować projekt i uprościć jego tworzenie - odpowiedzialności w systemie są wyraźnie wydzielone. Jeśli napotykasz sytuacje, kiedy po kilku dniach rozwoju pewnej aplikacji staje się ona niespójna – jest wiele powtórzeń, nie wiesz, jak rozdzielić kod odpowiedzialny za zapytania do bazy danych od reszty systemu, wtedy z pomocą może przyjść podział warstwowy w systemie.

Jeśli jesteś członkiem większego zespołu, prawdopodobnie architektura jest już góry narzucona. Już wcześniej ktoś zdecydował, że w systemie, który tworzysz, obowiązuje architektura warstwowa. W różnych technologiach może być ona inaczej zdefiniowana, ale podstawowa idea pozostaje taka sama. Znajomość warstw pozwala Ci łatwiej odnaleźć się w tworzonym systemie, łatwiej go zrozumieć i wpasować się do niego. Wiesz za co powinny odpowiadać Twoje klasy, a czym nie powinny się zajmować. Warstwy są jak kontynenty na mapie świata – wiesz co i gdzie możesz znaleźć.

Prosty przykład bezwarstwowy

Przyjrzyjmy się prostemu przykładowi opartemu o konsolę. Już w takiej aplikacji można bez większego wysiłku wyodrębniać warstwy. Oczywiście ważnym pytaniem, które należy sobie postawić, to pytanie „Czy warto stosować warstwy”. Na potrzeby tego artykuły dla prostoty użyjemy przykładu konsolowego.

Zajmiemy się prostą aplikacją, służącą do zarządzania tłumaczeniami słów z języka angielskiego na polski. W aplikacji możemy:

- dodawać nowe słowa i ich tłumaczenia;

- usuwać zadane słowo razem z tłumaczeniem;

- znaleźć słowo z jego tłumaczeniem;

- wyświetlić wszystkie słowa z tłumaczeniami:

- bez sortowania,

- sortowane alfabetycznie,

- sortowane według daty dodania słowa;

- aplikacja ma przechowywać dane w sposób trwały pomiędzy uruchomieniami.

Jedna z prostych implementacji takiego systemu mogłaby wyglądać następująco:

```
package bnsit.layers.wordbook;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
```

Warstwy pomagają Ci łatwiej opanować projekt i uprościć jego tworzenie

```
import java.io.ObjectOutputStream;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.Date;
import java.util.List;
import java.util.Scanner;

public class Wordbook {

  private static String FILENAME = "wordbook.dat";

  public static void main(String[] args)
    throws FileNotFoundException, IOException, ClassNotFoundException {

    List<DictionaryWord> words = loadData();

    boolean ok = true;
    Scanner s = new Scanner(System.in);

    System.out.println("Welcome to Wordbook.");

    while (ok) {
      System.out.print("dictionary> ");
      String line = s.nextLine();
      String [] arguments = line.split(" ");

      if ( line.startsWith( "search" ) ) {
        if ( arguments.length != 2 ) {
          System.out.println( "Command syntax: search <english_word>" );
        } else {
          String englishWord = arguments[1];
          for (DictionaryWord word : words) {
            if ( word.getEnglishWord().equals(englishWord) ) {
              System.out.println( word );
            }
          }
        }
      } else if ( line.startsWith( "add" ) ) {
        if ( arguments.length != 3 ) {
          System.out.println(
          "Command syntax: add <english_word> <polish_word>" );
        } else {
          String englishWord = arguments[1];
          String polishWord = arguments[2];
          DictionaryWord dictionaryWord
            = new DictionaryWord(
              englishWord, polishWord, new Date());
          words.add( dictionaryWord );
          System.out.println( "Word added: " + dictionaryWord );
          writeData(words);
        }
      } else if ( line.startsWith( "delete" ) ) {
```

> *Prawdopodobnie architektura jest już góry narzucona*

```
      if ( arguments.length != 2 ) {
        System.out.println(
          "Command syntax:delete <word_natural_number>");
      } else {
        int wordNumber = Integer.valueOf( arguments[1] );
        words.remove( wordNumber - 1 );
        writeData(words);
      }
    } else if ( line.equals( "show" ) ) {
      showList(words);
    } else if ( line.equals( "show sorted by name" ) ) {
      showList(words, new Comparator<DictionaryWord>() {
        @Override
        public int compare(DictionaryWord o1, DictionaryWord o2) {
          return o1.getEnglishWord()
            .compareTo(o2.getEnglishWord());
        }
      });
    } else if ( line.equals( "show sorted by date" ) ) {
      showList(words, new Comparator<DictionaryWord>() {
        @Override
        public int compare(DictionaryWord o1, DictionaryWord o2) {
          return o1.getDate().compareTo(o2.getDate());
        }
      });
    } else if ( line.equals( "exit" ) ) {
      ok = false;
    } else {
      System.out.println( "Command not found: '" + line + "'" );
    }
    }
    s.close();
}

private static void writeData(List<DictionaryWord> words)
    throws IOException, FileNotFoundException {
    ObjectOutputStream objectOutputStream
      = new ObjectOutputStream( new FileOutputStream( FILENAME ) );
    objectOutputStream.writeObject( words );
}

private static List<DictionaryWord> loadData()
  throws FileNotFoundException, IOException, ClassNotFoundException {

    List<DictionaryWord> result = new ArrayList<DictionaryWord>();
    File file = new File( FILENAME );
    if ( file.exists() ) {
      ObjectInputStream objectInputStream
        = new ObjectInputStream( new FileInputStream( FILENAME ) );
      result = (List<DictionaryWord>) objectInputStream.readObject();
    }
    return result;
}
```

> *Gdy system będzie się rozwijał, tego typu rozwiązanie będzie coraz trudniejsze w utrzymaniu*

```
    private static void showList(List<DictionaryWord> words) {
        int counter = 0;
        for (DictionaryWord word : words) {
          System.out.println( ++counter + " " + word );
        }
    }

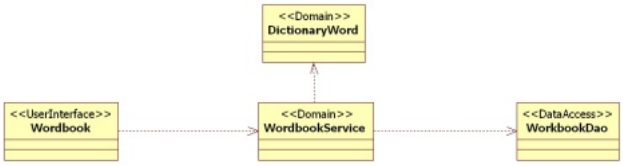
    private static void showList(List<DictionaryWord> words,
        Comparator<DictionaryWord> comparator) {

        List<DictionaryWord> wordsCopy = new ArrayList<DictionaryWord>(words);
        Collections.sort(wordsCopy, comparator);
        showList(wordsCopy);
    }
}
```

Jest to typowy przykład aplikacji o płaskiej architekturze. Oczywiście w tak prostym systemie tego typu rozwiązanie ma same zalety – jest proste, zwięzłe i dość łatwo poruszać się po kodzie. Jednak gdy tylko system będzie się rozwijał, tego typu rozwiązanie będzie coraz trudniejsze w utrzymaniu. Będzie występować coraz więcej powtórzeń, konstrukcje programistyczne będą coraz bardziej skomplikowane oraz elementy interfejsu użytkownika, dostępu do danych będą ze sobą wymieszane.

Wprowadzamy warstwy

Czy na podstawie tak prostego systemu możemy wyodrębnić warstwy? Oczywiście! Przyglądając się aplikacji możemy wydzielić elementy odpowiedzialne za interfejs użytkownika (pobieranie danych od użytkownika i wyświetlanie informacji na ekranie) – klasa *Wordbook*, za przetwarzanie w systemie (np. sortowanie, dodawanie nowych słów) – klasa *WordbookService* i dostęp do danych (zapis i odczyt z pliku) – klasa *WordbookDao*, co przedstawia poniższy rysunek.



Przyjrzyjmy się przykładom klasom, po to aby wyodrębnić główne cechy klas z danej warstwy. Na początek zajmiemy się klasą interfejsu użytkownika – *Wordbook* (kod źródłowy poniżej). Klasa ta, w porównaniu z kodem z poprzedniej wersji, ma konkretnie wydzieloną odpowiedzialność – interakcję z użytkownikiem. Pozostały w niej tylko instrukcje związane ze współpracą z konsolą oraz delegowanie konkretnych zadań do klasy *WordbookService*, która reprezentuję w tym przypadku warstwę dziedziny. Klasa *Wordbook*:

- pobiera dane z konsoli;

- waliduje i analizuje dane wpisywane przez użytkownika;

- wyświetla stosowne komunikaty;

- deleguje operacje konkretne.

Zauważmy, że nie ma tutaj żadnego konkretnego przetwarzania związanego z wewnętrzną logiką działania systemu. Tylko i wyłącznie interfejs użytkownika. Zatem odchudziliśmy klasę tak, aby pełniła jedną konkretną rolę w systemie.

Operacje konkretne są delegowane do klasy *WordbookService*, która zajmuje się głównymi

Konduktor

Czy na podstawie tak prostego systemu możemy wyodrębnić warstwy? Oczywiście!

```
public class Wordbook {
  private WordbookService wordbookService = new WordbookService();

  public static void main(String[] args) {
    Wordbook wordbook = new Wordbook();
    wordbook.run();
  }

  public void run() {

    boolean ok = true;
    Scanner s = new Scanner(System.in);

    System.out.println("Welcome to Wordbook.");

    while (ok) {
      System.out.print("dictionary> ");
      String line = s.nextLine();
      String [] arguments = line.split(" ");

      if ( line.startsWith( "search" ) ) {
        if ( arguments.length != 2 ) {
          System.out.println(
              "Command syntax: search <english_word>" );
        } else {
          String englishWord = arguments[1];

          List<DictionaryWord> words
            = wordbookService.find( englishWord );
          for (DictionaryWord word : words) {
              System.out.println( word );
          }
        }
      } else if ( line.startsWith( "add" ) ) {
        if ( arguments.length != 3 ) {
          System.out.println( "Command syntax: "
              + "add <english_word> <polish_word>" );
        } else {
          String englishWord = arguments[1];
          String polishWord = arguments[2];
          DictionaryWord dictionaryWord
            = wordbookService.createNewWord(
                englishWord, polishWord);
          System.out.println( "Word added: " + dictionaryWord );
        }
      } else if ( line.startsWith( "delete" ) ) {
        if ( arguments.length != 2 ) {
          System.out.println( "Command syntax: "
              + "delete <word_natural_number>" );
        } else {
          int wordNumber = Integer.valueOf( arguments[1] );
          wordbookService.remove( wordNumber );
        }
```

> *Odchudziliśmy klasę tak, aby pełniła jedną konkretną rolę w systemie.*

```
      } else if ( line.equals( "show" ) ) {
        List<DictionaryWord> words = wordbookService.findAll();
        showList(words);
      } else if ( line.equals( "show sorted by name" ) ) {
        List<DictionaryWord> words
          = wordbookService.findAllSortedByName();
        showList(words);
      } else if ( line.equals( "show sorted by date" ) ) {
        List<DictionaryWord> words
          = wordbookService.findAllSortedByDate();
        showList(words);
      } else if ( line.equals( "exit" ) ) {
        ok = false;
      } else {
        System.out.println( "Command not found: '" + line + "'" );
      }
    }
    s.close();
  }

  private void showList(List<DictionaryWord> words) {
    int counter = 0;
    for (DictionaryWord word : words) {
      System.out.println( ++counter + " " + word );
    }
  }
}
```

zadaniami związanymi z funkcjami systemu. Jednak operacje trwałego zapisu lub wyszukiwania danych są delegowane do innego obiektu – *WordbookDao*.

Przyjrzyjmy się klasie *WordbookService*. Co warto zauważyć?

1. Metody w tej klasie odpowiadają funkcjom systemu np. znajdź, usuń, znajdź wszystkie.

2. Metody są dość krótkie i czytelne.

3. Metody nie zależą w żaden sposób od interfejsu użytkownika, a więc można ich użyć z dowolnym interfejsem użytkownika!

4. Operacje zależne od źródła danych są delegowane do klasy *WordbookDao*.

```
public class WordbookService {

  private WordbookDao wordbookDao = new WordbookDao();

  public List<DictionaryWord> find(String englishWord) {
    return wordbookDao.find(englishWord);
  }

  public DictionaryWord createNewWord(String englishWord, String polish-
Word) {
    DictionaryWord dictionaryWord
      = new DictionaryWord( englishWord, polishWord, new Date());
    wordbookDao.save(dictionaryWord);
```

> *Można bez większych konsekwencji dla reszty aplikacji zmienić sposób zapisu danych*

```
    return dictionaryWord;
  }

  public void remove(int wordNumber) {
    DictionaryWord dictionaryWord
      = wordbookDao.findByIndex( wordNumber - 1  );
    wordbookDao.remove(dictionaryWord);
  }

  public List<DictionaryWord> findAll() {
    return wordbookDao.findAll();
  }

  public List<DictionaryWord> findAllSortedByName() {
    List<DictionaryWord> words = wordbookDao.findAll();
    Collections.sort(words, new Comparator<DictionaryWord>() {
      @Override
      public int compare(DictionaryWord o1, DictionaryWord o2) {
        return o1.getEnglishWord().compareTo(o2.getEnglishWord());
      }
    });
    return words;
  }

  public List<DictionaryWord> findAllSortedByDate() {
    List<DictionaryWord> words = wordbookDao.findAll();
    Collections.sort(words, new Comparator<DictionaryWord>() {
      @Override
      public int compare(DictionaryWord o1, DictionaryWord o2) {
        return o1.getDate().compareTo(o2.getDate());
      }
    });
    return words;
  }
}
```

Przyjrzyjmy się na końcu klasie *WordbookDao*. Kilka elementów, na które warto zwrócić uwagę:

1. Odpowiedzialność tej klasy to współpraca z danymi i źródłem danych (w tym przypadku jest to plik z zserializowanymi danymi).

2. Metody w tej klasie reprezentują podstawowe operacje związane z pracą na danych.

3. Metody są krótkie, czytelne i mają jednoznacznie zdefiniowaną odpowiedzialność.

4. Ze względu na enkapsulację dostępu do danych, można bez większych konsekwencji dla reszty aplikacji zmienić sposób zapisu danych (np. do pliku XML).

```
public class WordbookDao {

  final private String FILENAME = "wordbook.dat";
  private List<DictionaryWord> words = null;

  public WordbookDao() {
```

Jasno wydzielona odpowiedzialność, przygotowanie aplikacji na zmiany, łatwiejsze zarządzanie i organizacja kodu

```
    words = loadData();
  }

  public List<DictionaryWord> find(String englishWord) {
    List<DictionaryWord> result = new ArrayList<DictionaryWord>();
    for (DictionaryWord word : words) {
      if ( englishWord.equals(word.getEnglishWord()) ) {
        result.add(word);
      }
    }
    return result;
  }

  public DictionaryWord findByIndex(int i) {
    return words.get( i );
  }

  public List<DictionaryWord> findAll() {
    return new ArrayList<DictionaryWord>(words);
  }

  public void save(DictionaryWord dictionaryWord) {
    words.add(dictionaryWord);
    writeData(words);
  }

  public void remove(DictionaryWord dictionaryWord) {
    words.remove( dictionaryWord );
    writeData(words);
  }

  private void writeData(List<DictionaryWord> words) {
    ObjectOutputStream objectOutputStream;
    try {
      objectOutputStream = new ObjectOutputStream(
          new FileOutputStream(FILENAME));
      objectOutputStream.writeObject(words);
    } catch (Exception e) {
      throw new WordbookDaoException(e);
    }
  }

  private List<DictionaryWord> loadData() {
    List<DictionaryWord> result = new ArrayList<DictionaryWord>();
    File file = new File(FILENAME);
    if (file.exists()) {
      ObjectInputStream objectInputStream;
      try {
        objectInputStream = new ObjectInputStream(
            new FileInputStream(FILENAME));
        result = (List<DictionaryWord>) objectInputStream.readObject();
      } catch (Exception e) {
        throw new WordbookDaoException(e);
```

> *Dzięki zastosowaniu interfejsów elementy systemu są ze sobą luźno powiązane*

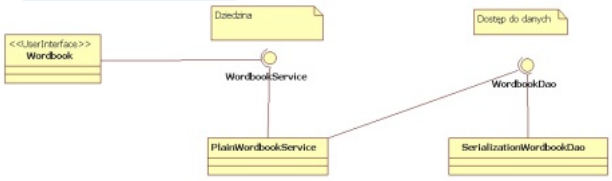
```
        }
      }
      return result;
    }
  }
```

W ten sposób udało nam się rozwarstwić aplikację. Jakie wynikają z tego konsekwencje? Jasno wydzielona odpowiedzialność, przygotowanie aplikacji na zmiany, łatwiejsze zarządzanie i organizacja kodu – wiadomo, gdzie szukać poszczególnych elementów. Z drugiej strony bardziej skomplikowana struktura – zamiast jednej klasy mamy trzy. Potencjalnie nasza aplikacja może również stracić na wydajności. Cóż, nie ma róży bez kolców. W prostych systemach – składających się z kilku, kilkunastu klas, takie podejście będzie zbyt pracochłonne. W większych systemach konkretyzuje strukturę i ułatwia nawigację.

Podmiana warstw

Jedną z głównych cech podziału warstwowego jest możliwość podmiany warstw i zmiany zastosowanych rozwiązań w danej warstwie przy względnie niewielkim wpływie na resztę systemu. To prawdziwa magia tego rozwiązania.

W tym celu powinniśmy uelastycznić budowę systemu. W chwili obecnej klasy systemu, są ze sobą ściśle powiązane. Zastosujemy dwie techniki, aby rozluźnić nieco projekt – zastosujemy interfejsy dla klas danej warstwy oraz zastosujemy wzorzec Dependency Injection, co umożliwi nam łatwe podmienianie zależności w aplikacji. System będzie miał taką postać:



Dzięki zastosowaniu interfejsów elementy systemu są ze sobą luźno powiązane i możemy podmieniać ich konkretne implementacje.

Klasa *Wordbook* korzysta z interfejsu *WordbookService*, co oznacza, że w jego miejsce możemy podstawić dowolną implementację (np. opartą o POJO lub EJB). Aby umożliwić wstrzykiwanie zależności dodaliśmy akcesory (metody set/get), zaś w metodzie *main* umieściliśmy budowanie powiązanych ze sobą klas.

```
public class Wordbook {
  private WordbookService wordbookService = null;

  public static void main(String[] args) {
    Wordbook wordbook = new Wordbook();

    PlainWordbookService plainWordbookService = new PlainWordbookService();
    plainWordbookService.setWordbookDao(new SerializationWordbookDao());
    wordbook.setWordbookService(plainWordbookService);

    wordbook.run();
  }
  // ...
  public WordbookService getWordbookService() {
    return wordbookService;
```

> *Jedną z głównych cech podziału warstwowego jest możliwość podmiany warstw i zmiany zastosowanych rozwiązań*


```
  }

  public void setWordbookService(WordbookService wordbookService) {
    this.wordbookService = wordbookService;
  }
}
```

Analogiczne zmiany wprowadziliśmy w klasie *PlainWordbookService*, która implementuje interfejs *WordbookService*.

```
public interface WordbookService {

  public abstract List<DictionaryWord> find(String englishWord);

  public abstract DictionaryWord createNewWord(String englishWord,
      String polishWord);

  public abstract void remove(int wordNumber);

  public abstract List<DictionaryWord> findAll();

  public abstract List<DictionaryWord> findAllSortedByName();

  public abstract List<DictionaryWord> findAllSortedByDate();

}

public class PlainWordbookService implements WordbookService {

  private WordbookDao wordbookDao = null;
 // ...*
  public WordbookDao getWordbookDao() {
    return wordbookDao;
  }

  public void setWordbookDao(WordbookDao wordbookDao) {
    this.wordbookDao = wordbookDao;
  }

}
```

Podobnie zmodyfikowaliśmy klasę *WordbookDao*. Poniżej zamieszczony został jej fragment. Pełną wersję kodów źródłowych do tego artykułu można pobrać ze strony http://www.bnsit.pl/rozwarstwienie/.

```
public interface WordbookDao {

  public abstract List<DictionaryWord> find(String englishWord);

  public abstract DictionaryWord findByIndex(int i);

  public abstract List<DictionaryWord> findAll();
```

> *Każda warstwa to dodatkowy poziom złożoności.*

```
  public abstract void save(DictionaryWord dictionaryWord);

  public abstract void remove(DictionaryWord dictionaryWord);

}
public class SerializationWordbookDao implements WordbookDao {

  final private String FILENAME = "wordbook.dat";
  private List<DictionaryWord> words = null;

  public SerializationWordbookDao() {
    words = loadData();
  }

  public List<DictionaryWord> find(String englishWord) {
    List<DictionaryWord> result = new ArrayList<DictionaryWord>();
    for (DictionaryWord word : words) {
      if ( englishWord.equals(word.getEnglishWord()) ) {
        result.add(word);
      }
    }
    return result;
  }* // ...

  private List<DictionaryWord> loadData() {
    List<DictionaryWord> result = new ArrayList<DictionaryWord>();
    File file = new File(FILENAME);
    if (file.exists()) {
      ObjectInputStream objectInputStream;
      try {
        objectInputStream = new ObjectInputStream(
            new FileInputStream(FILENAME));
        result = (List<DictionaryWord>) objectInputStream.readObject();
      } catch (Exception e) {
        throw new WordbookDaoException(e);
      }
    }
    return result;
  }
}
```

Obecnie aplikacja w warstwie dostępu do danych jest oparta o plik z zserializowanymi danymi. Równie dobrze możemy stworzyć inną implementację interfejsu *WordbookDao*, na przykład opartą o JDBC. Wtedy ta sama aplikacja, bez większych zmian w warstwie interfejsu użytkownika oraz dziedzinie, będzie działać z bazą danych! Potraktuj to jako ćwiczenie, a rozwiązanie znajdziesz na stronie http://www.bnsit.pl/rozwarstwienie/

Powyżej opisany sposób myślenia można przeskalować na bardziej złożone systemy.

Testowanie

Kolejna korzyść ze stosowania warstw ujawnia się w chwili, gdy testujemy system. Wystarczy porównać początkową i końcową wersję przykładowej aplikacji. Którą łatwiej przetestować? Czy monolityczny kod, który zawi-

> *Warstwy nie są złotym środkiem, który rozwiąże wszystkie kwestie architektoniczne*

era wiele alternatywnych ścieżek i przemieszany kod interfejsu użytkownika z kodem dziedziny i dostępem do danych? Czy może klasy z niewielkimi metodami, o dobrze określonej odpowiedzialności? Testowanie jednostkowe staje się przyjemnością.

Podsumowanie

Warstwy nie są złotym środkiem, który rozwiąże wszystkie kwestie architektoniczne. W złożonych systemach są nieodzowne, aby móc je efektywnie rozwijać. Jednak każda warstwa to dodatkowy poziom złożoności. W mniejszych aplikacjach jest to indywidualna decyzja, którą warto podjąć, jeśli w ustrukturyzowany sposób chcesz rozwijać swoją aplikację. Wtedy pozostaje pytanie, ile i jakie warstwy chcesz zastosować. Życzę powodzenia podczas eksperymentowania.

O autorze



Mariusz Sieraczkiewicz

Trener, konsultant, menedżer projektów IT, coach. Założyciel zespołu programistów Equilibrium. Współinicjator JUGa Łódź. Autor artykułów o inżynierii oprogramowania. Współwłaściciel firmy szkoleniowej BNS IT. Szczęśliwy mąż :-) http://www.linkedin.com/pub/2/a24/812

