Więcej niż Java

Autor: Bruce Tate
Tłumaczenie: Paweł Koronkiewicz
ISBN: 83-246-0329-8
Tytuł oryginału: Beyond Java
Format: B5, stron: 180

Java jest obecna na rynku od ponad 10 lat. Przez ten czas zyskała ogromną popularność i znacznie zmieniła swoje oblicze. Przestała być językiem wykorzystywanym do tworzenia mniej lub bardziej przydatnych apletów na strony WWW. Jest teraz potężną, uniwersalną platformą programistyczną używaną do budowania aplikacji korporacyjnych i finansowych. Jednocześnie istnieją obszary, w których Java nie jest wystarczająco elastyczna, a jej złożoność staje się problemem. Na rynku pojawiły się narzędzia mające uprościć tworzenie rozbudowanych systemów za pomocą Javy, jak na przykład Spring czy Hibernate. Jednak czy opracowanie zupełnie nowych platform programistycznych nie spowolni triumfalnego pochodu Javy?

W książce „Beyond Java" Bruce Tate zastanawia się nad przyszłością języków programowania. Przedstawia źródła popularności Javy i ogromne korzyści, jakie wniosła do współczesnej informatyki. Wskazuje problemy, z jakimi borykają się programiści korzystający z Javy. Wreszcie dokonuje przeglądu konkurencyjnych języków programowania, zastanawiając się, w czym są lepsze, a w czym gorsze od Javy i który z nich ma szansę zagrozić jej pozycji na rynku.

- Powody malejącej popularności C++
- Rozwój technologii open source
- Prawdy i mity związane z Javą
- Wady Javy
- Potencjalni konkurenci Javy
- Środowisko Ruby on Rails
- Przyszłość serwerów kontynuacyjnych

Jeśli chcesz wiedzieć, jakiego języka programowania będziesz używał za 5 lat, przeczytają tę książkę

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

ROZDZIAŁ 1.

Sowy i strusie

Znam kajakarzy, których postawa nie pozostawia wątpliwości — chcą zginąć. Lecą w dół odcinkami klasy V, ignorując wszelkie niebezpieczeństwa[1]. Wydaje się, że tylko kwestią czasu jest nadejście momentu, w którym wpadną na wodospad, gdzie utknęła kłoda drzewa. Przeszkoda zatrzyma kajak, a ten — przyciskany opadającymi zwałami wody — już nigdy się nie wynurzy. Są oni jak strusie, które szybko zapominają o niebezpieczeństwie po schowaniu głowy w piasek.

Jest też inny rodzaj kajakarza. Kiedy ja zająłem się tym sportem, każdy odcinek musiałem najpierw dokładnie zbadać, przejść się wzdłuż niego, poznać zagrożenia. Zatrzymywałem się przy każdym, najzwyklejszym przejściu klasy II+ (poziom „dla początkujących"), aby popatrzeć jak wygląda i nierzadko decydowałem się zainstalować zabezpieczenia. Często skutek był taki, że kończył mi się czas i byłem zmuszony z duszą na ramieniu płynąć ostatnim odcinkiem rzeki znacznie szybciej, aby zdążyć przed zmrokiem. Teraz, mając pewne doświadczenie, nie oglądam już tak szczegółowo każdego bardziej spienionego przesmyku. W wielu miejscach nie ma to po prostu sensu. Zacząłem natomiast stosować techniki „chase boating", wypracowane w wąskich i rwących strumieniach południowo-wschodnich stanów USA. Nie oznacza to wcale, że bardziej polubiłem ryzyko. Można powiedzieć, że koncentruję się bardziej na identyfikacji obszarów faktycznego zagrożenia. Dzięki temu zawsze mam czas na przeprowadzenie rozpoznania odcinków, których znajomość jest najbardziej potrzebna. Taki styl kajakarstwa można określić stylem sowy.

Podobnie jest w wielu innych dziedzinach. Często decyduję się na zignorowanie zagrożeń wywołujących mało znaczące konsekwencje lub rzadko występujących dlatego, że rozwiązywanie związanych z nimi problemów nie jest opłacalne. Szukanie i wdrażanie stuprocentowo poprawnych rozwiązań wymaga zbyt wiele wysiłku, pieniędzy lub czasu i niejednokrotnie prowadzi do powstania zupełnie nowych problemów. Na tym właśnie polega przewaga sowy nad strusiem. Na ogół te dwie postawy skrajnie różnią się od siebie. Należy jednak przyznać, że od czasu do czasu sowy również stają się zbyt pewne siebie albo popełniają drobne błędy w ocenie ryzyka. W efekcie kajakarz pokonuje niebezpieczny odcinek bez wstępnego rozpoznania. Mam za sobą takie pamiętne doświadczenie. Pływałem tą samą rzeką setki razy i widziałem zaskakujące zmiany, czy to głębokości, czy układu koryta po powodzi. Różnica między sową a strusiem nie jest tak wielka, jak mogłoby się wydawać. Często wręcz trudno ją wskazać. Jako kajakarz, skoro już zdecydowałem się ignorować pewne rodzaje niebezpieczeństw na pewnych rzekach i pod pewnymi warunkami, od czasu do czasu muszę wrócić do punktu wyjścia i przeprowadzić ponowną ocenę ryzyka. I o tym jest ta książka.

[1] Mowa o rwących górskich potokach — *przyp. tłum.*

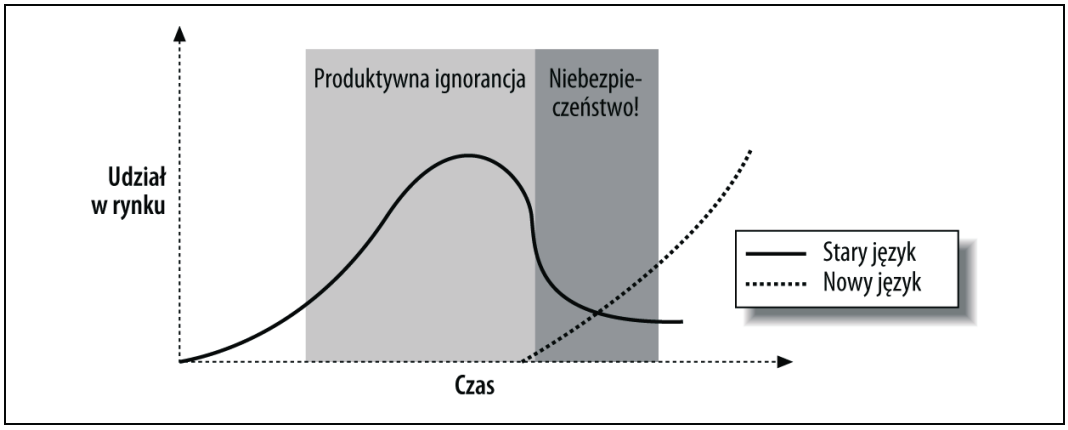
Ignorancja jako cnota

Można snuć wiele paraleli między kajakarstwem a programowaniem. W którymś momencie nauczyłem się wspaniałej sztuczki — ignorowanie większości problemów pozwala uzyskać niezwykłą produktywność. Przy odrobinie szczęścia wiele z nich z czasem po prostu przestaje istnieć. Oczywiście, taka postawa może być korzystna i skuteczna lub nie. Skuteczność tej techniki odkryli już dawno pracownicy urzędów pocztowych i słabo opłacani ekspedienci fast foodów. Sprawdza się ona w odniesieniu do ich problemów, zwanych najczęściej klientami. Mówimy tu oczywiście o strusiach. Jeżeli jednak przyjrzymy się temu dokładniej, niejednokrotnie znajdziemy w takim postępowaniu selektywne, mądre zastosowanie ignorancji — znak rozpoznawczy sowy. Praktyka przekonała mnie, że większość „problemów" procesu programowania to w istocie problemy *potencjalne*. Czytelnik, który zna chociaż jedną z moich wcześniejszych książek, pamięta zapewne, że uwielbiam pisać o zagrożeniach wynikających z przedwczesnej optymalizacji i powtarzać popularną zasadę Agile Programming — *YAGNI*: „You ain't gonna need it", czyli „Nie wykorzystasz tego". Zazwyczaj też trzymam się z dala od rozbudowanych systemów, które obiecują wielkie oszczędności czasu i ułatwione programowanie, a ufam swojemu instynktowi skłaniającemu mnie ku prostszym rozwiązaniom.

Wracając do tematu, pewnego dnia stwierdziłem, że Java ma wszystko, czego potrzebuję, i przestałem interesować się światem poza nią. Niewiedza jest słodka. Wiem, że istnieją języki bardziej dynamiczne, czasem też pozwalające na większą produktywność. Jednak, gdy rozważam różne aspekty i fragmenty pełnego obrazu, Java okazuje się na dłuższą metę najlepsza. W tym języku napisano już dziesiątki tysięcy najróżniejszych bibliotek i systemów umożliwiających stworzenie każdego rozwiązania, począwszy od obsługi reaktora atomowego, na programowaniu mikrosterownika wbudowanego w elektryczne szczypce do paznokci kończąc. Są one najczęściej określane nazwą *framework*, co można tłumaczyć jako „platforma" lub po prostu „system". Wiele z nich można wykorzystywać bezpłatnie. Nie ma też nigdy problemu ze znalezieniem programisty języka Java, który napisze nowy fragment programu. Wiem, że jest to język sprawdzony, a klienci ufają jego mechanizmom. Krótko mówiąc, środowisko i zakres możliwości Javy od wielu lat przewyższają to, co oferuje konkurencja. W końcu przestałem szukać innych dróg. Cieszę się, że taki moment nastąpił, ponieważ zyskałem dzięki temu czas na stworzenie prężnie działającej firmy i dopracowanie oferowanych rozwiązań. Zamiast prowadzić szeroko zakrojone „badania", mogłem spokojnie i z należytą uwagą zająć się problemami klientów.

Kiedy dominujący język lub technologia osiągają szczyt swojego rozwoju, następuje moment słodkiej ignorancji, prawdziwa sielanka, okres spokojnej pracy, w którym całkowite odrzucenie potencjalnych możliwości działa na naszą korzyść. Zilustrowałem to na rysunku 1.1. Kiedy pojawia się nowy język o takich możliwościach i zasięgu, jak Java czy C++, można na pewien czas pozwolić sobie na ignorancję. Problem pojawia się wówczas, gdy przeoczymy koniec takiego okresu. Pewnego dnia konkurencja okazuje się lepsza, osiąga większą wydajność pracy, a co z tym związane wyższą jakość, produktywność i liczbę klientów. Kiedy rozpoczyna się okres przejściowy, niewiedza nie popłaca.

Bez wstydu przyznaję, że lubiłem trzymać głowę w piasku. Było to łatwe, produktywne i, w owym czasie, poprawne politycznie. Jestem przekonany, że identycznie postąpiło wielu innych programistów języka Java, choć może nie zawsze z tego samego powodu. Życie pod kloszem jest niewątpliwie łatwiejsze — niepodejmowanie żadnego działania przynosi więcej korzyści niż praca po godzinach. Człowiek ma poczucie bezpieczeństwa — nie zwalnia się za wybranie marki IBM (choć może Component Broker w OS/2 nie był najlepszym pomysłem…).



Rysunek 1.1. Przez pewien czas ignorancja jest produktywna, potem jednak jej skutki mogą okazać się dotkliwe

Jeżeli zainwestowaliśmy wiele w swoje umiejętności, czerpiemy zyski pełnymi garściami. Jeżeli zainwestowaliśmy mniej, i tak możemy mieć rację. O trwałości związku z Javą mógł zdecydować duży, wciąż rozwijany projekt albo to, że pracujemy w pewnej, związanej z językiem, grupie. Powodów mogło być wiele i, podobnie jak moje, doskonale uzasadniały one przyjętą drogę postępowania.

Zachwianie w posadach

Po ponad pięciu latach słodkiej ignorancji nadszedł wielki wstrząs. Prowadziłem nowy projekt ścieżką, która wymagała zastosowania trzech, moim zdaniem najefektywniejszych, prostych systemów wspomagających tworzenie aplikacji operujących trwałymi danymi w sieci WWW: Hibernate, Spring i Web Work. Wiedziałem, że istnieją też środowiska mogące zaoferować nieco wyższą wydajność pracy programisty, ale te albo trudno skalować (w kategoriach złożoności lub wydajności), albo są zbyt mało popularne, aby podejmować ryzyko korzystania z nich.

Wraz z moim partnerem podjęliśmy jednak decyzję o zaimplementowaniu niewielkiego fragmentu aplikacji w Ruby on Rails, bardzo efektywnym systemie programowania WWW. Nie wynikało to z życzenia klienta czy szczególnych potrzeb, a jedynie z naszej ciekawości. Rezultaty były zaskakujące:

- Przy przepisywaniu naszego fragmentu w *Ruby on Rails* pracowaliśmy znacznie szybciej. Justin, główny programista, napisał w cztery dni to, co wcześniej tworzyliśmy przez cztery miesiące w Javie. Wzrost produktywności oceniliśmy w przedziale od 5 do 10 razy.
- Kod był czterokrotnie krótszy; pięciokrotnie, jeżeli doliczymy pliki konfiguracyjne.
- Wysoka wydajność pracy została zachowana nawet po wyjściu poza etap przepisywania wcześniejszego kodu.
- Wersja w Ruby on Rails pracowała szybciej. Trudno oczekiwać, że będzie tak w każdym przypadku, ale w naszym eksperymencie mechanizm aktywnych rekordów RoR pobił na głowę *Object Relational Mapping* (ORM, odwzorowanie obiektowo-relacyjne), który zapewnia Hibernate, nawet jeżeli wymagało to minimalnego dostrojenia parametrów roboczych.
- Klient był zdecydowanie bardziej zainteresowany naszą wydajnością niż tym, żeby korzystać ze sprawdzonej platformy języka Java.

Jak łatwo sobie wyobrazić, zachwiało to moim widzeniem świata. Od tego czasu „bycie na bieżąco" stało się dla mnie niemal obsesją. Było to tak, jakby sytuacja na rzece uległa nagłej, nieoczekiwanej zmianie. Musiałem udać się na długi spacer wzdłuż jej brzegów.

Gotowanie żab

Spójrzmy na to jeszcze inaczej. Czytelnik może słyszał opowieści o tym, że żaba włożona do gorącej wody natychmiast z niej wyskoczy, ale włożona do chłodnej pokornie pozwoli się ugotować. Na tego rodzaju zagadnienie chcę zwrócić uwagę w tej książce. Czy woda dookoła nas robi się już gorąca? Przypominam, że gdy pisałem o sowach i strusiach, konsekwencje ich postępowania oceniłem jako bardzo podobne. Sami zainteresowani mogą nie zdawać sobie z tego sprawy, ale ich motywacje mają niewielkie znaczenie. Kiedy woda zacznie się gotować bądź warunki na rzece ulegną zmianie, obaj zginą.

Przez ostatni rok wiele czasu poświęciłem badaniu temperatury otaczającej nas wody. Poznałem w tym czasie między innymi Ruby i programowanie aspektowe (ang. *aspect-oriented programming*, AOP). Te szczegółowe badania doprowadziły do konkluzji, że woda faktycznie robi się coraz gorętsza. Jeszcze się nie gotuje i nie jestem w stanie stwierdzić, że taki moment z pewnością nastąpi. Wiem jednak, że wciąż będę czujnie obserwował temperaturę. Mam też nadzieję, że przekonam do tego Czytelnika. Już wyjaśniam dlaczego.

Sygnały ostrzegawcze

W wielu pisanych przez nas aplikacjach łączymy oparty na WWW fronton z bazą danych, czasem uzupełniając ten podstawowy schemat o dodatkowe reguły biznesowe. Mimo to, po ponad pięciu latach rozwiązywania tego problemu w kolejnych systemach, wciąż nie ukształtowała się technika pozwalająca zrobić to w przestrzeni Javy naprawdę szybko. Co więcej, większość twórców środowisk Java ogranicza się do usprawnień stopniowych, które nie prowadzą do prawdziwej rewolucji w programowaniu WWW. Stworzenie całkiem nowego zespołu, który rozwiąże ten problem, nie jest proste. Powołanie zespołu z programistów, na przykład języka COBOL, okazuje się niemalże niemożliwe. Język jest już zbyt mało znany, środowiska rozbudowane, a krajobraz niestabilny. Nawet w gronie doświadczonych programistów zbudowanie nawet najprostszej aplikacji wymaga zaskakująco dużej ilości kodu.

Jason Hunter „Następca Javy"

Autor *Java Servlet Programming*

Jason Hunter pracuje jako główny programista aplikacji w Mark Logic. Jest autorem książki *Java Servlet Programming* (wyd. O'Reilly). Jako reprezentant projektu Apache w Komisji Wykonawczej organizacji Java Community Process (Społeczność Języka Java), doprowadził do popisania kluczowej umowy prowadzącej do udostępnienia Javy jako Open Source. Jest właścicielem Servlets.com i XQuery.com, jednym z twórców serwera Apache Tomcat i członkiem grup odpowiedzialnych za rozwój API Servlet, JSP, JAXP i XQJ, należał też do XQuery Working Group (Grupy Roboczej XQuery) w organizacji W3C. Jest również współtwórcą opensource'owej biblioteki JDOM wspomagającej integrację Javy i XML.

Czy Javie grozi utrata obecnej, wiodącej pozycji?

JH: Okres królowania Javy już się zakończył. Proces rozpoczął się mniej więcej dwa lata temu, gdy najwybitniejsi specjaliści branży zaczęli odchodzić od technologii związanych z tym językiem i skierowali swoją uwagę ku innym obszarom. Teraz jest to już zjawisko bardzo wyraźne. Odejście Josha Blocha i Neala Gaftnera do Google to tylko jeden z wielu przykładów. Ktokolwiek robi dzisiaj coś nowego, nie opiera tego na znaczących innowacjach w Javie. Java może być narzędziem, ale nie jest już rozwijaną platformą.

Nie oznacza to, że Java stała się językiem martwym. Nie jest po prostu najnowszą, wiodącą technologią. Jest dobrze znana, bardzo stabilna i doskonale sprawdza się w outsourcingu.

Co będzie kolejnym przebojem?

JH: Co będzie kolejnym przebojem? Myślę, że nie można mówić o jednym przeboju. Java wciąż jeszcze jest wszędzie. Obszarem innowacji są teraz wyższe poziomy. Przykładami ciekawych tematów mogą być: zdalny dostęp do obiektów w sieci WWW (czyli Ajax), wyszukiwanie (czyli Google i XQuery) i folksonomie (czyli znaczniki *flickr.com*).

Mam własny, praktyczny sposób oceniania, która technologia jest w danym momencie najbardziej „gorąca" — jest to ta, dzięki której znający ją wykładowca może zarobić najwięcej. Java była taką technologią przez wiele lat. Zarabiałem dwa razy więcej niż wykładowcy C++. Nie wynikało to z poziomu trudności, ale z wielkości popytu.

Ten, kto uczy obsługi narzędzia, które zdążyło już spowszednieć (jak obecnie C++ i Java), zarabia przeciętnie. Podobnie jest, gdy uczymy czegoś zupełnie nowego. Wtedy klientów może być bardzo mało.

Nie dostrzegam teraz żadnego ruchu, który mógłby powtórzyć scenariusz Javy. Co robią *alpha geeks*[2], jak nazywa ich Tim O'Reilly? Na przykład James Davidson wszedł po uszy w Macintosha. Nie jest to jednak rynek o wielkim zapotrzebowaniu na specjalistów. Nie ma na nim wielu pieniędzy. Ja poszedłem w stronę XQuery, które zdobyło moją sympatię jako bardzo praktyczny sposób przeniesienia wyszukiwania „do siebie" i uzyskania pełnej kontroli nad uzyskanymi wynikami i ich dalszą obróbką. Mike Clark został specjalistą od automatyzacji. Osobom, którym brakuje konkretnego celu, polecałbym dzisiaj zajęcie się nauką pracy z Subversion i przeprowadzenie migracji CVS do SVN.

Drogi programistów muszą się rozejść. Zgodziliśmy się co do języka Java jako pewnej podstawy. Czas na zróżnicowanie, głównie w obszarze jego zastosowań.

W zadawanych pytaniach kurczowo trzymasz się Javy i „alternatywy dla Javy". Sieć WWW nie była alternatywą dla Windows. Była czym innym. Nie jesteśmy teraz w stadium zastępowania wszędobylskiego języka. Zajmujemy się czymś zupełnie innym. Java może stać się takim standardem jak procesor komputera; elementem niezbędnym. Java nie jest już magnesem dla pieniądza. Jest naszym chlebem powszednim.

[2] W wolnym tłumaczeniu „samce alfa informatyki".

Złożoność

Język Java wydaje się coraz bardziej odchodzić od swoich korzeni. Rozwiązanie najtrudniejszego nawet problemu jest znacznie uproszczone, ale wyzwaniem staje się napisanie prostej aplikacji WWW. James Duncan Davidson określa ten problem mianem **przystępności** (ang. *approachability*). W czasach młodości Javy zbudowanie niewielkiego apletu nie wymagało wielkiej wiedzy. Obecnie, aby zbudować prostą aplikację WWW na podstawie jednego z popularnych środowisk, musimy dysponować znacznie większym zasobem informacji i doświadczeniem.

Prawdą jest, że narzędzia Open Source nieustannie zmieniają, i to w wielkim stopniu, efektywność pracy w Javie. Jest to kierunek ze wszech miar korzystny. Wspaniałe narzędzia, takie jak Hibernate i Spring, znacznie ułatwiają budowanie aplikacji, które mogą być stosowane w największych firmach. Jednak nauka właściwego i sprawnego korzystania z nich może trwać rok, a nawet dłużej. Programowanie aspektowe również się przydaje, ponieważ umożliwia pisanie zwykłych obiektów Java (ang. *plain old Java objects*, POJO) reprezentujących reguły biznesowe i izolowanie usług w takich „aspektach", jak bezpieczeństwo i transakcje. Tego rodzaju abstrakcje nieustannie podnoszą poprzeczkę, której muszą stawić czoło początkujący. Moje pytanie brzmi: „Jak wysoko można ją jeszcze podnieść?". Wydaje mi się, że już teraz jej wysokość zaczyna być dla większości świeżo upieczonych adeptów programowania bardzo kłopotliwa. Z coraz większą trudnością przychodzi mi mówienie klientom, że każdego programistę COBOL-a można przeszkolić i zrobić z niego programistę Javy. Nauki jest zbyt wiele i zajmuje ona zbyt wiele czasu.

W przeszłości złożone problemy prowadziły do tworzenia coraz wyższych poziomów abstrakcji. Kiedy komputery stały się zbyt rozbudowane, aby programować je bezpośrednio przy użyciu odpowiednich połączeń przewodów, specjaliści zaczęli stosować kod maszynowy. Kiedy pisane w nim programy stały się zbyt duże, aby efektywnie z nimi pracować, powstał język asemblera, w którym kody maszynowe i dane oznaczano już za pomocą pewnego systemu symboli. Rosnąca złożoność doprowadziła do powstania języków wysokiego poziomu, programowania strukturalnego i programowania obiektowego. W moim przeświadczeniu ta narastająca rzeka złożoności kiedyś wyleje, co zmusi nas do przyjęcia zupełnie nowego systemu abstrakcji. Co więcej, nastąpi to stosunkowo szybko.

Błyskawiczna rewolucja

W czasie ostatnich trzech lat byliśmy świadkami niezwykłej ilości otaczających Javę innowacji. Doświadczyliśmy przejścia od rozbudowanych kontenerów, takich jak EJB, do prostych i lekkich, takich jak Spring. Wielu Czytelników przeszło zapewne od stosowania EJB lub JDBC do iBATIS, JDO lub Hibernate. Wielu też zapewne dostrzega zalety zastąpienia platformy Struts takim rozwiązaniem, jak Tapestry. Moje doświadczenie potwierdza regułę, że matką większości wynalazków jest potrzeba. Zgodnie z moją teorią, koło zmian dostaje wielkiego przyspieszenia, gdy złożoność osiąga pewną granicę. Trudno tu o dowody, ale istnieje wiele wskazujących na to poszlak:

- przytłaczająca liczba nowości w dziedzinie systemów zapewniania trwałości danych,
- rozpowszechnianie się platform model-widok-kontroler (MVC),
- rozwój kontenerów,
- gwałtowny rozwój systemów wiązania XML.

Nowości towarzyszą faktycznym potrzebom. Jeśli dysponujemy produktem, który zaspokaja nasze potrzeby (lub zbliżonym), jak Ant czy Junit, pozostawiamy go w spokoju do czasu, gdy przestanie służyć naszym celom.

Doświadczeni programiści nie wykażą się zapewne zrozumieniem dla trudnego procesu uczenia się, niezbędnego do napisania najprostszej aplikacji WWW w Javie. Wielu może się nawet skarżyć, że przykładam do tego problemu zbyt wielką wagę. Jeżeli Czytelnik jest jednym z nich, proponuję poszukać inteligentnego, mało doświadczonego programisty języka Java uczącego się dziesiątek różnych aplikacji, których znajomość jest niezbędna do tworzenia aplikacji WWW dla przedsiębiorstw, i porozmawiać z nim. Problem jest dwojaki. Po pierwsze, trudno się tego wszystkiego nauczyć. Po drugie, konsekwencje porażki są dotkliwe. Jeżeli w którymś momencie wybierzemy złą drogę albo projekt realizowany przy użyciu starszej technologii uwiąże nas na trzy długie, pracowite lata, to przy kolejnym przedsięwzięciu znajdziemy się znowu w punkcie wyjścia. Skutki zmian mogą być przytłaczające. Dla mnie oznacza to, że programowanie musi odbywać się na wyższym poziomie abstrakcji, którego próżno szukać w Javie.

Nienaturalne rozciąganie

Prawdopodobnie już od pewnego czasu Czytelnik wychodzi w swoich programach poza pierwotne założenia języka Java. Trudno obecnie polemizować ze stwierdzeniem, że obiekty kodowane za pomocą podstawowych bibliotek języka praktycznie nigdy nie będą już wystarczające. W książce *Better, Faster, Lighter Java* dowodziłem, że dążenie do kodowania wszystkich usług i ich zachowań jako obiektów biznesowych jest błędem, a dziedziczenie jest niewystarczające. Musimy uciekać się do sztuczek, takich jak poprawianie kodu bajtowego w czasie kompilacji albo generowanie kodu w czasie wykonania. Dopiero wówczas obiekty stają się przezroczyste. Rozszerzamy więc możliwości Javy poza pierwotne założenia twórców języka. Nie ma w tym nic złego… ale do czasu. Jednym ze skutków jest narastanie bariery wejścia. Spytajmy o nią początkującego programistę, który próbował rozwiązywać problemy z opóźnionym ładowaniem Hibernate lub pośrednikami Spring.

Zauważyłem też, że inne, bardziej dynamiczne języki rzadko odwołują się do takich metod, jak programowanie aspektowe lub wstrzykiwanie zależności. Pozwalają one rozwiązać bardzo ważne problemy w Javie i na tym tle języki bardziej dynamiczne, na przykład Smalltalk, Python i Ruby, wydają się znacznie wygodniejsze.

Nie uważam, że są to techniki złe. Górują one nad innymi rozwiązaniami prostotą i potencjalnymi możliwościami. Rozwiązują naprawdę trudne problemy. Jedyny kłopot stanowi to, że umysł ludzki ma ograniczone możliwości przyswajania nowej wiedzy i umiejętności. Java coraz szybciej staje się efektywnym zestawem narzędzi dla elity programistów. Cóż, może właśnie taki jest kierunek rozwoju dziedziny programowania. Sądzę tylko, że takie nienaturalne naciąganie możliwości stosowanej platformy jest kolejnym sygnałem, że warto zmierzyć temperaturę wody.

Ewolucja języka

Wielokrotnie zapowiadano, że Java 5 ma być największym krokiem w rozwoju języka na przestrzeni połowy dekady. Zgadzam się z tym, że znaczenie nowej wersji będzie ogromne. Zastanawia mnie jednak bardzo, czy jej wpływ będzie pozytywny. Na regularnej konferencji NoFluffJustStuff, której centralnym punktem jest dyskusja panelowa z grupą ekspertów, moje

ulubione pytanie dotyczy nowych elementów języka. Wszyscy bez wahania zgadzają się z tym, że generalia, przynajmniej w postaci, w jakiej są implementowane teraz, nie są wcale dobrym pomysłem. Stwierdzenie to jest zazwyczaj szokiem dla dużej części publiczności.

Związany z generaliami mechanizm Java Specification Request (JSR, żądanie specyfikacji) wprowadza stosunkowo dużą liczbę elementów składniowych tylko po to, by rozwiązać problem, którego znaczenie jest, ogólnie rzecz biorąc, marginalne. Brak natomiast zmiany w maszynie wirtualnej JVM. Wydaje mi się, że przeciętny programista języka Java rzadko ma do czynienia z wyjątkiem rzutowania klasy (ang. *class cast*), choć okazji nie brakuje. Większość obiektów w typowej aplikacji języka Java jest gromadzonych w kolekcje. Przy każdej operacji dostępu niezbędne jest rzutowanie z klasy `Object`. W tym momencie bezpieczeństwo typologiczne języka pomaga tyle co pas bezpieczeństwa w pikującym Boeingu. Składnia generaliów razi swoją inwazyjnością. Jeszcze mniej dobrego można powiedzieć o implementacji. W czasach, gdy coraz więcej ekspertów przyznaje, że typy dynamiczne pozwalają pisać prostsze aplikacje i skracają czas pracy nad nimi, programiści Javy uczą się, jak budować ściślejszą kontrolę typów statycznych.

Dodajmy do tego kontrowersyjne stosowanie skądinąd uzasadnionych mechanizmów, takich jak adnotacje mogące prowadzić do zupełnej zmiany semantyki programu bez użycia konwencjonalnego kodu. To kolejne źródło potencjalnych kłopotów. Czy uzyskane korzyści równoważą wzrost złożoności i zmniejszenie przejrzystości kodu? Adnotacje to zupełnie nowe narzędzie i niemalże nowy model programowania. Nie potrafię przewidzieć, czy nauczymy się jeszcze stosować adnotacje w sposób właściwy, ale jestem pewny, że zanim to nastąpi, popełnimy jeszcze liczne błędy.

Nie chciałbym w tym miejscu wyczerpywać tematu. O ograniczeniach Javy będę jeszcze pisał w rozdziałach od 3. do 5. Teraz chciałem tylko zwrócić uwagę Czytelnika na to, że Java nie jest już nowym, wspaniałym językiem. Stała się problemem i wyzwaniem. Czy to problemy dojrzewania? A może nieuleczalny reumatyzm starości? Nie wiem, ale temperatura rośnie szybko i lepiej trzymać rękę na pulsie.

Co dobre, to DOBRE

Nie chciałbym, żeby Czytelnik odniósł wrażenie, że próbuję obwieścić koniec języka Java. Próbuję raczej zwrócić uwagę tak sów, jak i strusi, że czas już wytężyć wzrok i nieco bardziej uważnie obserwować zmieniającą się sytuację. Moje stanowisko można podsumować stwierdzeniem, że **warunki sprzyjają powstawaniu nowych, wartościowych rozwiązań**. W chwili pisania tych słów pozycja Javy jest bardzo silna. W istocie, sprzyjają jej nowe czynniki o ogromnym znaczeniu:

- „Społeczność" Javy jest niezwykle aktywna. Łatwo znaleźć specjalistów rozwiązujących najtrudniejsze problemy. Łatwo też o nie-programistów, przede wszystkim sprzedawców i menedżerów, którym język Java nie jest obcy.
- Większość producentów docenia znaczenie Javy, a przynajmniej pochodzącego od niej języka C#. W efekcie dostępne są najróżniejsze aplikacje, serwery, komponenty, narzędzia, usługi, a nawet konsole zarządzania.
- Ruch Open Source jest kolejną, ogromną siłą i źródłem pojawiających się już codziennie nowatorskich, godnych uwagi rozwiązań.

- Instytucje akademickie włączyły programowanie w języku Java do swoich programów nauczania i podjęły inwestycje związane z wieloma pokrewnymi problemami. Przykładem może być przedsięwzięcie, z którym zetknąłem się ostatnio, zainicjowane właśnie w uniwersyteckim laboratorium, a mające doprowadzić do stworzenia praktycznego narzędzia szacującego wydajność programu w języku Java na podstawie jego diagramu UML.
- Wynalazek JVM zapewnia niepowtarzalny poziom przenośności oprogramowania między różnymi systemami. Niektórzy wierzą, że to właśnie maszyna wirtualna JVM zapisze się w historii jako krok bardziej znaczący niż sam język.

Jeszcze do niedawna wierzyłem, że tak wielka prężność i różnorodność środowiska skupionego wokół Javy gwarantuje niezachwianą przewagę tego języka nad każdym innym w niemal każdym zastosowaniu. Zresztą, nawet jeżeli Java nie jest w danym przypadku najlepsza, cóż może być porównywalną alternatywą? Już samo skompletowanie większego zespołu programistów innego języka stanowiłoby problem. Czytelnik zapewne myśli teraz: „Spójrzmy prawdzie w oczy, mamy do wyboru .NET i Javę". Przejście od Javy do .NET przypominałoby poprawianie jakości diety przez zmianę restauracji z McDonald's na Burger King. A porównywalnych rozwiązań praktycznie nie ma.

Jest to w dużej mierze prawdą. Jeżeli nie istnieje godna zaufania alternatywa, kurs na Javę zapewni sukces. W takim przypadku ta książka już jest martwa, a Czytelnik może nie zawracać sobie nią głowy. Zanim jednak tak się stanie, niech jeszcze poświęci swój cenny czas na przeczytanie tylko kilku kolejnych stron.

Nowe horyzonty

Muszę zawczasu uprzedzić Czytelnika, że jestem z natury dość cyniczny. Kiedy mowa o technologiach, niełatwo wzbudzić moje uniesienie. Wciąż jeszcze nie napisałem bodaj jednej „usługi WWW" (przynajmniej nie przy użyciu wielkich stosów *web services* IBM i Microsoftu). Pierwszy Enterprise Java Bean zakodowałem dopiero w 2003 roku. Nigdy też nie napisałem i nie zamierzam pisać obiektów *entity bean* (pomijam oczywiście przykłady w wywodach przeciwko ich stosowaniu). Zawsze wolałem prostsze architektury, takie jak REST, programowanie POJO, przezroczystą trwałość i Spring. Samo przekonanie się do nich wymagało czasu.

Jeszcze trudniej skłonić mnie do zabawy z nowym językiem. Dave Thomas, znany i szanowany programista oraz utalentowany wykładowca, lubi powtarzać, że powinniśmy uczyć się nowego języka programowania co kilka miesięcy. Moja średnia bliższa jest pięciu lat, a w poznawaniu „nowego języka" rzadko wykraczałem poza zabawę. Dopiero ostatnio moje poszukiwania nowości doprowadziły do ciekawszych wyników. Nie tylko ciekawszych. Nagle poczułem się jak na terenie zagrożonym trzęsieniem ziemi.

Zdecydowałem się poświęcić związanym z nowymi językami programowania nowościom znacznie więcej czasu niż zwykle. W budowaniu stron WWW i serwerów aplikacji moją uwagę niezmiennie przykuwają dwie koncepcje: metaprogramowanie (jak w Ruby on Rails) i serwery ciągłości (jak Seaside w języku Smalltalk). Żadna z nich nie wpłynęła w dużym stopniu na rozwój Javy. Dokładnie zajmiemy się tym tematem w rozdziałach 7. i 8., ale dość powiedzieć, że obie platformy są kilkakrotnie bardziej wydajne niż ich odpowiedniki w Javie.

Języki dynamiczne

Java szczególnie zapracowała sobie na miano języka kompromisów. Wiele cech Javy ułatwia budowanie rozszerzeń systemu operacyjnego i oprogramowania middleware, a negatywnie odbija się na programowaniu aplikacji. Rozważmy prosty fragment w języku Ruby:

```
tekst = "Sowy i strusie"
4.times {puts tekst}
```

Powoduje on czterokrotne wyświetlenie tekstu `Sowy i strusie`. Zwróćmy uwagę na potencjał tego języka:

- Jeżeli nie chcemy, nie musimy zawracać sobie głowy takimi szczegółami, jak określanie typów. Jeżeli coś wygląda jak kaczka i do tego kwacze, Ruby potraktuje to jako kaczkę. Pozwala to zaoszczędzić więcej czasu, niż mogłoby się wydawać.
- `4` to obiekt. Wszystko jest obiektem. Można wywoływać metody obiektu `4` i metody ciągów znakowych.
- `{puts someting}` to blok kodu. Blok kodu można przekazać jako parametr, a metoda może traktować taki parametr jako blok kodu (a nie wartość bloku). Znakomicie ułatwia to takie operacje, jak iterowanie elementów i modyfikowanie wnętrza pętli w bibliotece.

Już tych kilka prostych cech pozwala znakomicie zwiększyć wydajność pracy. Dodajmy do tego inne cechy języka dynamicznego, a okaże się, że mamy do dyspozycji ogromne możliwości i szansę na niezwykle efektywne programowanie. Już jakiś czas temu zauważono, że języki określane jako skryptowe mogą być bardzo wygodne dla programistów aplikacji.

Metaprogramowanie

W ostatnich latach okres intensywnego rozwoju przechodzą style programowania określane przymiotnikami dynamiczny, przezroczysty i refleksyjny. Ogólnie można je określić mianem **metaprogramowania** (ang. *metaprogramming*), ponieważ dotyczą nie tyle obiektów, co klas. Faktycznie, podejście takie umożliwia znaczne usprawnienie pracy. Platformy przezroczystej trwałości, takie jak Hibernate, zapewniają standardowym klasom i kolekcjom ciągłość egzystencji. Programowanie aspektowe (AOP) pozwala rozbudować grupę metod o dodatkowy kod, bez modyfikowania tych metod. Tego rodzaju zagadnienia to zagadnienia metaprogramowania.

Kiedy ekspert w dziedzinie Javy odkrywa metaprogramowanie, nowa przygoda kończy się często dla niego przejściem do korzystania z zupełnie innego języka. Łatwo o przykłady. David Geary, jeden z najpopularniejszych autorów książek o programowaniu w języku Java i członek grupy JSF, poświęcił się obecnie poznawaniu Ruby on Rails i pisaniu o tym języku. James Duncan Davidson, twórca serwera Tomcat i narzędzia Ant, opuścił społeczność Java, aby pisać programy w Objective C dla platformy Macintosh. Również Justin Gehtland, wraz ze mną, korzysta z Rails przy budowie nowych aplikacji WWW.

Metaprogramowanie można rozumieć jako budowanie wysokopoziomowego „kreatora". Ruby on Rails wykorzystuje zapisane w schemacie bazy danych kolumny i relacje do budowy modelu, widoku i kontrolera aplikacji WWW. Cechy środowiska robią wrażenie:

- Jest niesłychanie wydajne. Istnieje rozległa grupa problemów, w przypadku których można mówić o co najmniej pięciokrotnym wzroście efektywności pracy w porównaniu z podobnymi narzędziami języka Java.
- Jest elastyczne. Istnieją mechanizmy budujące standardową aplikację, którą można później w określony sposób rozbudowywać. Standardową aplikację wygenerowaną przez Rails można rozszerzać z równą swobodą, jak program napisany samodzielnie.
- Sprzyja ograniczeniu powtórzeń kodu i prowadzi do większej spójności.

Kiedy rozważam programowanie aplikacji korporacyjnych, najważniejszą cechą języka lub środowiska jest dla mnie efektywność wykonywanej pracy. Chcę, żeby każdy wiersz kodu pracował możliwie ciężko i żeby korelowało to z produktywnością. Przy tym nie kończę mierzenia wydajności pracy na etapie wdrożenia. Jeżeli pielęgnacja i rozwój malutkiej aplikacji okazują się niemożliwe, tracimy praktycznie wszystko. Takie właśnie podejście sprawiło, że zakochałem się w Ruby on Rails. Do tych zagadnień wrócimy jeszcze w rozdziale 7.

Serwery kontynuacyjne

Programiści WWW, którzy korzystają z języka Java, tracą niewiarygodną ilość czasu na zajmowanie się stanem aplikacji, wątkami i obsługą przycisku *Wstecz*. Tego rodzaju kwestie stają się naprawdę bardzo trudne, gdy witryna jest złożona i dynamiczna. Mała rewolucja nastąpiła ostatnio w języku Smalltalk, który teraz kojarzy się przede wszystkim z platformą o nazwie Seaside. Wykorzystuje się w niej cechę języka o nazwie *kontynuacja* (ang. *continuation*) umożliwiającą przechowywanie stanu w aplikacjach WWW. Kontynuacje dbają o przechowywanie danych stanu aplikacji, dlatego oparte na nich serwery są wolne od problemu zarządzania nimi. Kontynuacje pozwalają też w prosty sposób realizować obsługę przycisków *Wstecz* oraz wątków.

Perspektywy

Nie chcę powiedzieć, że Smalltalk czy Ruby zajmą jutro miejsce Javy. Nie jestem nawet przekonany, że sukces królującego obecnie języka może zostać powtórzony. Nie wierzę jednak, że Java jest wieczna. Przez pięć lat ignorowanie tego, co poza nią, było dobrą strategią. Niestety, żaden język nie utrzyma wiodącej pozycji po wsze czasy. W tym momencie przesłanki zawarte w niniejszej książce powinny być już dość jasne:

- Java oddala się od korzeni i swojej pierwotnej bazy użytkowników. Choć rozwiązywanie poważnych problemów aplikacji korporacyjnych jest często łatwiejsze niż kiedykolwiek, rozwiązanie problemów prostych staje się coraz trudniejsze. Co więcej…
- Java staje się „językiem z przeszłością". Ciekawe innowacje coraz częściej pojawiają się poza nią. Zatem…
- Czas przywrócić zwyczaj rozważania różnych języków implementacji.

Podnieś głowę, otwórz oczy. Rozpocznij od tej książki, a nie pożałujesz.