helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

J2EE. Podstawy programowania aplikacji korporacyjnych

Autor: Alan Monnox
Tłumaczenie: Mikołaj Szczepaniak
ISBN: 83-246-0211-9
Tytuł oryginału: Rapid J2EE(TM) Development: An Adaptive Foundation for Enterprise Applications
Format: B5, stron: 480

Kompendium wiedzy dla każdego programisty, projektanta i kierownika projektu

- Nowoczesne metodyki wytwarzania oprogramowania
- Narzędzia do modelowania aplikacji i automatycznego generowania kodu
- Koncepcja architektury sterowanej modelami
- Sposoby zapewnienia jakości aplikacji

Tworzenie aplikacji korporacyjnych to wyścig z czasem. Organizacje zmieniają się podobnie jak otoczenie biznesowe, w którym działają. Zbyt długi okres przygotowania aplikacji może sprawić, że po wdrożeniu okaże się ona bezużyteczna. Z drugiej jednak strony, zbyt duży pośpiech przy tworzeniu aplikacji powoduje, że pomija się fazę modelowania i testowania, pisząc kod źródłowy bez jakiejkolwiek koncepcji i planu. Efektem takiego pośpiechu są aplikacje niedostosowane do wymagań użytkowników i pracujące niestabilnie. Sposobem na stworzenie odpowiedniego systemu informatycznego dla korporacji jest wykorzystywanie odpowiednich metodyk projektowych i nowoczesnych narzędzi ułatwiających zarówno pisanie, jak i testowanie aplikacji.

Książka „J2EE. Podstawy programowania aplikacji korporacyjnych" przedstawia najlepsze praktyki projektowe stosowane przy tworzeniu systemów informatycznych z wykorzystaniem platformy J2EE. Opisano w niej kolejne etapy projektu oraz narzędzia i metodyki, dzięki którym przeprowadzenie każdego z nich będzie szybsze i efektywniejsze. Czytając ją, poznasz metodyki RUP i XP, typy architektur systemów oraz sposoby modelowania aplikacji i narzędzia do automatycznego generowania szkieletu kodu źródłowego. Dowiesz się, jak optymalnie skonfigurować środowiska programistyczne i jak testować kolejne moduły aplikacji. Nauczysz się korzystać z nowoczesnych metodyk i narzędzi.

- Podstawowe wiadomości o błyskawicznym wytwarzaniu aplikacji (RAD)
- Metodyki projektowe Rational Unified Process (RUP) oraz Extreme Programming (XP)
- Wielowarstwowe architektury systemów
- Modelowanie systemów za pomocą języka UML
- Automatyczne generowanie kodu
- Stosowanie narzędzi XDoclet i Hibernate
- Komunikacja z bazami danych
- Zasady programowania aspektowego
- Testowanie aplikacji

Wiadomości zawarte w tej książce sprawią, że będziesz w stanie szybciej projektować i tworzyć aplikacje korporacyjne.

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

Spis treści

Rozdział 12.

Optymalne kompilowanie systemów

Jednym z pierwszych zadań, które będziesz musiał zrealizować w fazie konstruowania projektu, będzie właściwe zaplanowanie i przygotowanie środowiska pracy dla zespołu projektowego. Dobrze zaprojektowane środowisko jest podstawowym warunkiem zapewnienia produktywności i efektywności całego zespołu projektowego.

W niniejszym rozdziale skupimy się tylko na jednym aspekcie środowiska wytwarzania — przedmiotem analizy będzie proces kompilacji, który jest kluczowym czynnikiem decydującym o możliwości błyskawicznego wytwarzania aplikacji. Szczegółowo omówimy wagę szybkiego, precyzyjnego i dającego możliwość zarządzania procesu kompilacji i przeanalizujemy techniki korzystania z narzędzia Apache Ant w zakresie konstruowania optymalnych środowisk kompilacji dla aplikacji platformy J2EE.

W rozdziale omówimy szereg zagadnień związanych z procesem budowy w ramach projektów błyskawicznego wytwarzania oprogramowania:

- Znaczenie efektywnych mechanizmów kompilacji w procesie błyskawicznego wytwarzania aplikacji.
- Zalety programu Ant jako narzędzia obsługującego kompilację.
- Sposoby wykorzystywania zależności kompilacji do ograniczania czasu potrzebnego do kompilacji.
- Zastosowanie narzędzia **Antgraph** (oferowanego z otwartym dostępem do kodu źródłowego), które umożliwia graficzne przeglądanie zależności zadeklarowanych w plikach Anta.
- Wskazówki dotyczące organizowania artefaktów kodu składających się na projekty J2EE.
- Najczęściej spotykane dobre praktyki korzystania z narzędzia Ant.

Na końcu tego rozdziału ponownie przeanalizujemy język skryptowy Jython i zademonstrujemy sposób, w jaki skrypty tego języka mogą być wykorzystywane do rozszerzania funkcjonalności Anta.

Czas i ruch

Każdy, kto kiedykolwiek miał okazję pracować przy linii produkcyjnej w zakładzie przemysłowym, doskonale zna pojęcie **czasu i ruchu** (ang. *time and motion*). Analiza czasu i ruchu obejmuje między innymi przegląd operacji potrzebnych do wytworzenia produktu w fabryce, a jej celem jest ograniczenie czasu produkcji i — tym samym — poprawa wydajności linii produkcyjnej. Taka analiza musi dotyczyć wszystkich kroków procesu wytwarzania, ponieważ tylko takie rozwiązanie pozwoli zidentyfikować wszystkie czynniki obniżające wydajność zakładu.

Linia produkcyjna musi być wzorem efektywności, natomiast inżynierowie odpowiedzialni za przebieg produkcji powinni wykorzystywać analizy czasu i ruchu podczas optymalizowania procesu produkcji. Wyobraź sobie linię produkcji samochodów, gdzie ogromna liczba budowanych aut przesuwa się na gigantycznej taśmie. Jeśli choć jeden krok tego procesu nie będzie realizowany w sposób optymalny, może się okazać, że nie tylko wzrasta koszt wyprodukowania każdego z samochodów, ale też spada liczba pojazdów wprowadzanych przez firmę na rynek.

Czas produkcji nie jest jedynym kryterium, którym powinni się kierować inżynierowie odpowiedzialni za funkcjonowanie takiej linii. Innym ważnym elementem jest jakość — proces musi gwarantować satysfakcjonujący i w miarę stały poziom jakości każdego z towarów opuszczających linię produkcyjną.

Chociaż analiza czasu i ruchu jest techniką wykorzystywaną przede wszystkim w zakładach produkcyjnych, podobne podejście (przede wszystkim skupianie uwagi inżyniera produkcji na efektywności) równie dobrze mogłoby znaleźć zastosowanie w przypadku takich zadań jak projektowanie procesu kompilacji oprogramowania.

Linia produkcji oprogramowania

Niewiele osób dostrzega podobieństwa pomiędzy procesami wytwarzania oprogramowania a funkcjonowaniem linii produkcyjnych w zakładach przemysłowych. Wytwarzanie oprogramowania jest zadaniem wymagającym nie tylko twórczego myślenia, ale też nieco innego podejścia do każdego budowanego systemu. Okazuje się jednak, że pewne działania składające się na projekt informatyczny są często powtarzane (nawet codziennie) niemal przez wszystkich członków zespołu.

Do zadań inżyniera oprogramowania należą tak naturalne czynności jak pobieranie najnowszej wersji aplikacji z systemu kontroli wersji kodu źródłowego, kompilowanie, wdrażanie i testowanie nowej funkcjonalności aplikacji oraz przeglądanie dziennika zdarzeń w poszukiwaniu błędów, które ujawniły się w systemie od ostatniego cyklu. Te i wiele innych zadań składają się na typowy dzień pracy niemal każdego programisty.

Podobne, wielokrotnie powtarzane zadania są realizowane także przez inne osoby zaangażowane. Zespół odpowiedzialny za zapewnianie jakości regularnie wykonuje proces instalowania najnowszej wersji przekazanej przez zespół programistów i przygotowywania odpowiednich środowisk dla przyszłych testów. Taki zespół może realizować te zadania ręcznie lub z wykorzystaniem skryptów tworzących nowe, „czyste" środowisko dla każdego cyklu testowego.

Aby było możliwe błyskawiczne wytwarzanie aplikacji, wszystkie te działania muszą być wykonywane efektywnie i bez zakłóceń.

Czas i ruch w kontekście wytwarzania oprogramowania

Środowisko wytwarzania oprogramowania oraz stosowane w nim procesy i procedury muszą gwarantować wszystkim członkom zespołu projektowego warunki do wygodnej i efektywnej pracy. Aby osiągnąć ten optymalny model realizacji projektu, zaangażowane zespoły powinny stosować praktyki bardzo podobne do tych właściwych dla inżynierów w zakładach produkcyjnych — powinny wykonywać własne analizy czasu i ruchu w zakresie tych czynności, które są wykonywane odpowiednio często.

Nie chodzi oczywiście o zatrudnianie inżynierów z doświadczeniem w przemyśle, którzy z zegarkiem w ręku będą pilnowali należytego zaangażowania i produktywności programistów. Za zapewnianie właściwej efektywności powinni odpowiadać wszyscy członkowie zespołu projektowego — każdy z nich musi stale poszukiwać sposobów na udoskonalanie procesów.

Uwzględnianie w praktykach wytwarzania oprogramowania wniosków zgłaszanych przez projektantów i programistów ma kluczowe znaczenie dla utrzymania właściwego fundamentu adaptacyjnego pod błyskawiczne wytwarzanie oprogramowania, ponieważ tylko przez udoskonalanie tego procesu z projektu na projekt można osiągnąć produktywne środowiska pracy dla przyszłych aplikacji.

Skoro mamy już świadomość znaczenia czasu i ruchu, przejdźmy do analizy niełatwych zagadnień związanych z procesem kompilacji.

Proces kompilacji

Dobry proces kompilacji powinien być czymś więcej niż tylko pojedynczym skryptem automatyzującym kompilowanie oprogramowania. Proces kompilacji powinien automatyzować wiele typowych i powtarzalnych czynności wykonywanych w ramach projektu. Do przykładów takich czynności należą:

- pobieranie kodu z systemu kontroli wersji,
- instalacja odpowiednich wersji bibliotek i oprogramowania,

- kompilacja bibliotek i komponentów aplikacji,
- uruchamianie zautomatyzowanych pakietów testowych,
- generowanie dokumentacji w standardzie JavaDoc,
- upakowanie komponentów i bibliotek w skompresowanych pakietach,
- konfiguracja środowisk wytwarzania i testowania systemu, włącznie z aktualizacją schematów wykorzystywanych baz danych i przygotowaniem danych testowych,
- wdrażanie aplikacji w środowisku wytwarzania,
- tworzenie kolejnych wersji komponentów programowych,
- wdrażanie budowanych wersji w środowisku testowym.

Przygotowanie takiego procesu kompilacji jeszcze przed przystąpieniem do właściwej realizacji projektu może się przyczynić do znacznych oszczędności czasowych. Procesy kompilacji zwykle mają postać wyrafinowanych narzędzi, które wymagają uważnego zaprojektowania, skonstruowania i przetestowania. Oznacza to, że inwestycja w procesy wytwarzania i kompilacji (które będą później wykorzystywane w wielu projektach) jest kluczowym czynnikiem decydującym o kształcie fundamentu adaptacyjnego pod błyskawiczne wytwarzanie oprogramowania.

W następnym punkcie przedstawimy kilka wskazówek odnośnie tworzenia procesu kompilacji, który ułatwi realizację z najlepszych praktyk błyskawicznego wytwarzania oprogramowania.

Projektowanie procesu kompilacji

Tak jak budowane systemy informatyczne, także systemy kompilacji wymagają przeprowadzenia fazy projektowania. Niezależnie od rodzaju wytwarzanego oprogramowania, znaczenie systemu kompilacji jest cechą wspólną wielu projektów. Poniżej wymieniono i krótko opisano kilka najważniejszych wymagań w tym zakresie:

Dokładność

Proces kompilacji musi w sposób logiczny i spójny łączyć kompilacje — prowadzić do generowania takich samych wyników na żądanie wszystkich programistów zaangażowanych w prace nad projektem dla tego samego zbioru plików źródłowych.

Błyskawiczność

Skoro kompilacje są wykonywane często i w regularnych odstępach czasu, proces musi być na tyle zoptymalizowany, aby wyeliminować ryzyko niepotrzebnego tracenia czasu na realizację tego procesu.

Automatyzacja

Wszystkie kroki procesu kompilacji muszą być kontrolowane przez odpowiednie narzędzia, które zapewniają jego całkowitą automatyzację. Jeśli programiści będą musieli wykonywać pewne kroki (np. kopiowania

plików lub kompilowania poszczególnych modułów) samodzielnie, ryzyko popełnienia błędów w tym procesie będzie znacznie wyższe. Co więcej, brak pełnej automatyzacji wyklucza możliwość kompilacji oprogramowania poza godzinami pracy, np. w nocy.

Standaryzacja

Sposób stosowania procesu kompilacji w kolejnych projektach realizowanych przez dany zespół powinien być identyczny lub bardzo podobny.

Parametryczność

Kompilacja przeprowadzana na potrzeby programisty najprawdopodobniej będzie się nieznacznie różnić od tej realizowanej z myślą o środowisku formalnych testów. Tego typu rozbieżności mogą się sprowadzać do odpowiednich opcji kompilatora lub pomijania pewnych kroków kompilacji. Proces kompilacji musi umożliwiać generowanie systemów w wersjach właściwych dla każdego z typów środowisk.

Możliwość konserwacji

We współczesnych środowiskach kompilacji można zaobserwować tendencję do zwiększania rozmiaru i poziomu wyrafinowania do punktu, w którym ich konserwacja staje się poważnym problemem (nie wspominając już o wydłużonym czasie konfiguracji tych środowisk). System kompilacji musi z jednej strony zapewniać prostotę konserwacji, z drugiej zaś powinien obsługiwać nawet najbardziej skomplikowane zadania kompilacji. Okazuje się niestety, że te dwa wymagania bywają sprzeczne.

Opisane powyżej wymagania dotyczą większości projektów polegających na wytwarzaniu oprogramowania. Wytwarzanie oprogramowania korporacyjnego wprowadza dodatkowy zbiór wymagań specyficznych dla platformy J2EE.

Wymagania platformy J2EE w zakresie kompilacji

Proces kompilacji w przypadku konwencjonalnej aplikacji Javy dotyczy zwykle prostej struktury budowanej z myślą o pojedynczym komputerze klienta, zatem nie wymaga skomplikowanego środowiska kompilacji. Tego samego nie można niestety powiedzieć o aplikacjach platformy J2EE, w przypadku których proces kompilacji składa się z wielu zawiłych kroków prowadzących do generowania modułów dla wielu docelowych komputerów.

W przeciwieństwie do programów platformy J2SE aplikacja J2EE składa się ze zbioru komponentów, które dopiero razem tworzą system informatyczny. Każdy z tych komponentów może wprowadzać własne, nieraz bardzo zróżnicowane wymagania w zakresie kompilacji. Komponenty EJB wymagają takich wyspecjalizowanych zadań kompilacji jak generowanie implementacji obiektów pośredniczących i szkieletów. Z uwagi na oferowane oszczędności czasowe stale rośnie popularność automatycznych generatorów kodu (np. popularnego XDocleta), warto jednak pamiętać, że ich stosowanie oznacza dodatkowe kroki w procesie kompilacji i — tym samym — podnosi poziom złożoności tego procesu.

Narzędzie XDoclet omówiono w rozdziale 6.

Poza tymi wyspecjalizowanymi zadaniami, komponenty J2EE wymagają tzw. **upakowania** (ang. *packaging*). Komponenty EJB należy umieszczać w archiwach Javy (ang. *Java archive — JAR*), natomiast aplikacje internetowe są upakowywane w archiwach internetowych (ang. *Web archive — WAR*). Na koniec wszystkie komponenty można jeszcze zebrać w jednym pliku zasobów korporacyjnych (ang. *Enterprise Resource — EAR*), czyli pliku zgodnym z formatem obsługiwanym przez serwery aplikacji (takie rozwiązanie bardzo ułatwia wdrażanie aplikacji J2EE).

Podsumowując, kompilacja oprogramowania dla platformy J2EE zwykle obejmuje następujące zadania (których wykonywanie nie jest konieczne w przypadku tradycyjnych aplikacji Javy):

- uruchomienie generatorów kodu,
- przeprowadzenie wyspecjalizowanej kompilacji stosowanych komponentów (np. Enterprise JavaBeans),
- upakowanie (w plikach JAR, WAR i EAR),
- wdrożenie.

Wszystkie te zadania wymagają czasu. Procesem szczególnie kosztownym czasowo jest upakowywanie, które wymaga przeniesienia wszystkich plików systemu do struktury, w której będzie je można zapisać w formacie gotowym do ostatecznego wdrożenia. Może się okazać, że równie czasochłonnym zadaniem będzie wdrożenie komponentów na serwerze i ich przygotowanie do prawidłowego funkcjonowania.

Skracanie czasu poświęcanego na kompilację i wdrażanie systemu musi polegać na ograniczaniu ilości pracy potrzebnej do realizacji tych procesów. W ten sposób dochodzimy do dwóch ważnych pojęć związanych z pracą systemów kompilujących: **minimalnych kompilacji** (ang. *minimal builds*) i **minimalnych wdrożeń** (ang. *minimal deployments*).

Minimalne kompilacje

Generowanie kodu źródłowego, kompilowanie tego kodu i upakowywanie plików binarnych w plikach JAR to zadania, które wymagają wielu zasobów. Jeśli istnieje możliwość ograniczenia częstotliwości wykonywania któregoś z tych zadań, może to oznaczać znaczne oszczędności czasowe.

Tego rodzaju oszczędności w czasie pracy systemu kompilacji zwykle polegają na zastępowaniu pełnego procesu kompilacji odpowiednimi zadaniami przyrostowymi, czyli kompilowaniem tylko tych komponentów, które uległy zmianie w ostatnim cyklu. Oznacza to, że jeśli zmieniono jakiś plik źródłowy, system kompilujący nie powinien ponownie generować całego systemu — powinien oferować mechanizmy wykrywania zależności pomiędzy komponentami i obszarem aplikacji, na który dana zmiana rzeczywiście miała wpływ, aby na tej podstawie skompilować, upakować i wdrożyć tylko zmodyfikowane moduły.

Wdrożenia minimalne

Kompilacje minimalne to tylko jeden z kroków zaradczych mających na celu skrócenie cykli kompilacji. Warto się również zastanowić, jak skompilowana aplikacja może osiągać stan, w którym będzie ją można uruchamiać i testować. W przypadku rozwiązań dla platformy J2EE naturalnym rozwiązaniem jest wdrażanie komponentów na serwerze aplikacji.

Podejście polegające na minimalnych wdrożeniach sprowadza się do redukcji łącznej liczby kroków wykonywanych przez inżynierów oprogramowania podczas wdrażania zmienionej aplikacji na serwerze. Najgorszym przypadkiem jest oczywiście konieczność zatrzymania serwera, ponownego wdrożenia całej aplikacji i przywrócenia pracy serwera. Opóźnienia związane z taką procedurą w wielu sytuacjach są nie do przyjęcia. Większość producentów serwerów aplikacji J2EE na szczęście zrozumiało, jak duże znaczenie ma szybkie i efektywne wdrażanie zmodyfikowanego oprogramowania, i w związku z tym ich systemy oferują funkcje **wdrażania aplikacji „na gorąco”**, **w ruchu** (ang. *hot deployment*). Z uwagi na swoje znaczenie dla efektywności procesu kompilacji praktyka wdrażania „na gorąco” wymaga dalszego wyjaśnienia.

Czym jest wdrażanie „na gorąco”?

Wdrażanie „na gorąco” jest rozwiązaniem stosowanym przez większość producentów serwerów aplikacji i sprowadzającym się do możliwości wdrażania zaktualizowanej aplikacji J2EE w docelowym, działającym środowisku bez konieczności wstrzymywania pracy serwera lub samej aplikacji.

W przypadku działających systemów technika wdrażania „na gorąco” w sposób oczywisty eliminuje problem dostępności oprogramowania. Inne oczekiwania w zakresie wdrażania „na gorąco” mają programiści, którzy bardziej skłaniają się ku odmianie tej koncepcji nazywanej **wdrażaniem automatycznym**.

Idea wdrażania automatycznego przewiduje, że serwer aplikacji stale „dopytuje się” o nowe pliki. W przypadku ich wykrycia, natychmiast wczytuje odnalezione zmiany i integruje nową wersję z bieżącą aplikacją.

Automatyczne wdrażanie nie ma zastosowania w systemach pracujących w swoim docelowym środowisku, ponieważ konieczność nieustannego sprawdzania katalogów wdrożenia w poszukiwaniu nowych wersji plików stanowiłoby nadmierne obciążenie i — tym samym — obniżało wydajność tych systemów.

W środowisku wytwarzania, w którym szczególny nacisk kładzie się efektywność pracy, tego typu funkcjonalność może znacznie odciążyć inżyniera oprogramowania, który nie będzie już musiał ręcznie wykonywać kroków związanych z wprowadzaniem nowych wersji budowanego oprogramowania do serwera aplikacji.

Koncepcja automatycznego wdrażania nie jest częścią specyfikacji platformy J2EE, która wspomina tylko o wdrażaniu plików EAR, WAR, JAR i RAR. Okazuje się jednak, że serwery aplikacji (np. WebLogic firmy BEA) w tym zakresie wykraczają poza

oficjalną specyfikację J2EE i oferują inżynierom oprogramowania obsługę znacznie bardziej wyrafinowanych mechanizmów wdrażania aplikacji niż tylko funkcji kopiowania skompresowanych plików. Zalecane przez firmę BEA podejście do wdrażania jest całkowicie zgodne z ideą minimalnych wdrożeń.

Serwer WebLogic obsługuje automatyczne wdrożenia pod warunkiem, że pracuje w trybie wytwarzania. Aby uruchomić serwer w tym trybie, wystarczy ustawić wartość `false` dla właściwości systemowej `-Dweblogic.ProductionModeEnabled`.

Aby sprawdzić, jakie mechanizmy w zakresie minimalnych wdrożeń i wdrożeń „na gorąco" oferuje Twój serwer aplikacji, musisz się dokładnie zapoznać z jego dokumentacją.

Do tej pory omówiliśmy kilka najważniejszych wymagań stawianych procesom kompilacji systemów informatycznych, zatem możemy przystąpić do tworzenia takiego procesu. Aby było to możliwe, w pierwszej kolejności zapoznamy się z wygodnym narzędziem kompilującym.

Wprowadzenie do narzędzia Ant

Ant jest rozszerzalnym narzędziem opracowanym przez Apache Software Foundation. W programie Ant skrypty kompilacji tworzy się, stosując składnię zbliżoną do języka XML. Narzędzie jest oferowane z otwartym dostępem do kodu źródłowego i można je pobrać za darmo z witryny internetowej Apache'a (patrz *http://ant.apache.org*).

Ant nie wymaga chyba wprowadzenia, ponieważ już od jakiegoś czasu jest de facto standardowym narzędziem wykorzystywanym do kompilowania programów Javy. We wcześniejszych rozdziałach wielokrotnie wspominano, że takie narzędzia jak **XDoclet**, **Middlegen** czy **AndroMDA** do swojego działania wymagają właśnie Anta.

Narzędzie Middlegen omówiono w rozdziale 7., natomiast program AndroMDA opisano w rozdziale 8.

Sukces programu Ant wynika przede wszystkim z jego bogatej funkcjonalności, która czyni z niego narzędzie idealnie pasujące do wytwarzania aplikacji Javy. Wykorzystywanie dokumentów XML (z jasną i czytelną składnią) w roli plików definiujących przebieg kompilacji powoduje, że programiści mogą bardzo szybko opanować techniki używania Anta. Trudności w tym zakresie były podstawowym problemem wcześniejszych narzędzi Make, które bazowały na trudnej do nauki semantyce deklaratywnej.

Co więcej, Ant został zaimplementowany w Javie i pracuje pod kontrolą wirtualnej maszyny Javy. Oznacza to, że Ant jest narzędziem niezależnym od platformy, przenośnym, co ma niemałe znaczenie dla programistów pracujących nad rozwiązaniami dla platformy J2EE. To także ogromna zaleta w porównaniu z wcześniejszymi narzędziami Make, które w procesie kompilowania oprogramowania korzystały z polecenia powłoki właściwej dla konkretnej platformy.

Ant stał się dojrzałym narzędziem kompilującym, który oferuje bardzo bogatą funkcjonalność w zakresie niemal wszystkich zadań kompilacji, jakie można sobie tylko wyobrazić. Pliki kompilacji tego programu wywołują operacje kompilacji za pośrednictwem tzw. **zadań Anta** (ang. *Ant tasks*).

Ant oferuje zestaw kluczowych, wbudowanych zadań, za pomocą których można wykonywać wiele typowych operacji kompilacji — oto najważniejsze z nich:

- kompilacja kodu źródłowego Javy,
- definiowanie ścieżek do klas,
- generowanie dokumentacji JavaDoc,
- kopiowanie i usuwanie plików,
- zmiana uprawnień dostępu do plików,
- tworzenie plików JAR,
- uruchamianie aplikacji zewnętrznych,
- wywoływanie kroków kompilacji zdefiniowanych w pozostałych plikach kompilacji Anta,
- przetwarzanie formatów kompresji (w tym plików ZIP i TAR),
- wysyłanie wiadomości poczty elektronicznej,
- obsługa dostępu do repozytoriów oprogramowania kontroli wersji kodu źródłowego.

W sytuacji, gdy wbudowane zadania Anta nie obsługują niezbędnych operacji kompilacji, programista może zaimplementować własne zadania, do których będzie się później odwoływał z poziomu pliku kompilacji.

Kluczowe cechy programu Ant

Oto kilka powodów popularności Anta w społeczności programistów Javy:

- **łatwość użycia** dzięki prostemu, opartemu na składni XML-a językowi skryptowemu,
- **wieloplatformowość** — Ant działa we wszystkich systemach, w których można uruchomić wirtualną maszynę Javy,
- **bogata funkcjonalność** dzięki szerokiemu zakresowi wbudowanych zadań,
- **rozszerzalny model** oferujący programiście możliwość definiowania własnych zadań Anta w Javie,
- **standardowość** — producenci oprogramowania bardzo często wykorzystują zadania Anta,
- **ogromna baza użytkowników**, co oznacza, że większość inżynierów oprogramowania ma doświadczenie w pracy z plikami kompilacji tego programu.

Większość producentów oprogramowania pisanego w Javie z myślą o programistach tego języka obsługuje zadania Anta jako sposób kontroli procesów kompilacji podczas

korzystania z ich narzędzi. Tego typu rozwiązania zastosowano np. w takich generatorach kodu jak XDoclet, Middlegen czy AndroMDA, których funkcjonowanie jest w ogromnym stopniu uzależnione właśnie od zadań programu Ant.

Niesamowita popularność Anta w świecie Javy sprawia, że w wielu sytuacjach trudno tego użycia narzędzia uniknąć w procesach kompilacji. W kolejnych podrozdziałach skupimy się na zastosowaniach programu Ant podczas projektowania i implementacji optymalnych rozwiązań w zakresie kompilacji.

Kompilacje minimalne w programie Ant

Aby narzędzie obsługujące kompilację mogło realizować koncepcję kompilacji minimalnej (przyrostowej), musi oferować mechanizm identyfikowania zależności pomiędzy rozmaitymi artefaktami projektu, które składają się na modyfikowaną i kompilowaną aplikację.

Przed pojawieniem się Anta większość programistów Javy korzystało z narzędzi Make, które także wykorzystywały przygotowywane wcześniej pliki kompilacji. Narzędzia Make bazowały na semantyce deklaratywnego języka programowania, która dawała możliwość definiowania reguł zależności pomiędzy artefaktami systemu. Takie podejście oznaczało, że narzędzia make wnioskowały na temat niezbędnych zadań kompilacji w zależności od tego, który plik źródłowy lub komponent został zmieniony.

Programowanie deklaratywne omówiono w rozdziale 10.

Zaimplementowany w narzędziach Make mechanizm wnioskowania umożliwiał co prawda minimalne kompilacje oprogramowania, jednak kosztem tej opcji była większa złożoność plików kompilacji. Narzędzie Ant wykorzystuje w swoich plikach kompilacji składnię znacznie prostszą niż program Make, ale sam nie obsługuje deklaratywnego podejścia do procesu kompilacji.

Znaczenie zależności kompilacji

Aby docenić rzeczywiste znaczenie zależności kompilacji, przeanalizujmy przykładowy deskryptor *build.xml* (patrz listing 12.1). W tym przypadku plik kompilacji sygnalizuje programowi Ant właściwą kolejność kompilowania każdego ze składników.

Listing 12.1. *Przykładowy plik kompilacji Anta (build.xml)*

```
<project name="ant-build" default="compile">

  <target name="compile"
          description="Kompiluje wszystkie źródła Javy">
    <javac scrDir="."/>
  </target>
```

```
  <target name="clean"
          description="Usuwa wszystkie pliki klas">
    <delete>
      <fileset dir="." include="*.class"/>
    </delete>
  </target>

  <target name="build"
          depends="clean, compile"
          description="Ponownie kompiluje wszystkie źródła"/>

</project>
```

Przykładowy plik kompilacji przedstawiony na listingu 12.1 zawiera trzy zadania kompilacji: `compile`, `clean` i `build`. Zadanie `compile` ustawiono jako domyślne dla danego projektu, zatem właśnie ono będzie realizowane za każdym razem, gdy uruchomimy Anta (chyba że wprost wskażemy zadanie alternatywne, np. `build`).

Pierwsze wykonanie tego pliku kompilacji z domyślnym zadaniem `compile` (bez określenia zadania alternatywnego w poleceniu wywołującym) spowoduje kompilację wszystkich plików źródłowych Javy w katalogu bazowym. Załóżmy, że nasz katalog zawiera tylko jeden taki plik: *HelloWorld.java*.

Wykonywanie tego pliku kompilacji po raz drugi będzie trwało znacznie krócej, ponieważ Ant sprawdzi tylko, czy wszystkie pliki klas są aktualne, i tak naprawdę nie wykona żadnej kompilacji. Jak to możliwe, skoro przedstawiony plik kompilacji nie zawiera żadnych informacji o zależnościach pomiędzy plikami *HelloWorld.java* i *HelloWorld.class*?

Sekret tkwi w zadaniu `<javac>` wykorzystywanym do kompilowania plików źródłowych Javy. Otóż zadanie `<javac>` zawiera wbudowane reguły zależności i „wie", że pliki **.java* i **.class* należy ze sobą wiązać. Oznacza to, że przynajmniej w przypadku plików z kodem źródłowym Javy Ant realizuje koncepcję minimalnych kompilacji, która jest podstawą szybkiego procesu kompilacji. Ant nie oferuje niestety podobnych mechanizmów dla pozostałych typów danych. Problem jest o tyle istotny, że np. kompilacje aplikacji dla platformy J2EE wymagają wykonywania wielu ręcznie dostosowywanych kroków.

Na listingu 12.2 przedstawiono zmodyfikowaną wersję kodu z listingu 12.1, która ilustruje ten problem.

Listing 12.2. *Plik kompilacji Anta ze zdefiniowaną zależnością*

```
<project name="ant-build" default="compile">

  <target name="generate"
          description="Czasochłonne zadanie kompilacji">
    <ejbdoclet>
      .
      .
      .
    </ejbdoclet>
```

```
  </target>

  <target name="compile"
          depends="generate"
          description="Kompiluje wszystkie źródła Javy">
    <javac scrDir="."/>
  </target>

  <target name="clean"
          description="Usuwa wszystkie pliki klas">
    <delete>
      <fileset dir="." include="*.class"/>
    </delete>
  </target>

  <target name="build"
          depends="clean, compile"
          description="Ponownie kompiluje wszystkie źródła"/>

</project>
```

Do poprzedniej wersji pliku kompilacji dodano nowe zadanie, <generate>, które wywołuje generator kodu (np. narzędzie XDoclet) przetwarzający z kolei pliki zawierające odpowiednie przypisy metadanych.

Aby było możliwe przetworzenie zadania <compile>, musi się zakończyć przetwarzanie wspomnianego zadania <generate>. Taka zależność pomiędzy dwoma zadaniami jest wyrażana za pomocą specjalnego atrybutu narzędzia Ant: depends. Umieszczenie tej zależności w pliku kompilacji powoduje, że Ant zawsze będzie przetwarzał zadanie <generate> przed zadaniem <compile>.

Zdefiniowana w ten sposób relacja pomiędzy zadaniami <generate> i <compile> ma charakter proceduralny — Ant nie określa, czy należy wykonać zadanie <generate>, na podstawie znaczników czasowych odpowiednich plików.

Przyjmijmy, że nasz przykładowy projekt składa się z dziesięciu tysięcy plików z kodem źródłowym, z których pięć tysięcy oznaczono przypisami lub atrybutami metadanych programu XDoclet. Uwzględnienie modyfikacji w pliku zawierającym takie przypisy wymaga uruchomienia XDocleta. Jeśli jednak przedmiotem zmiany był plik bez przypisów, można bezpiecznie pominąć zadanie <generate> i zrealizować wyłącznie zadanie <compile>. Ponieważ zadanie <compile> wykorzystuje podzadanie <javac>, Ant i tak ponownie skompiluje tylko zmodyfikowane pliki źródłowe.

Idealnym rozwiązaniem byłoby oczywiście stosowanie generatora kodu (XDocleta lub jakiegoś innego narzędzia), który oferowałby taką samą funkcjonalność jak opisane wcześniej zadanie <javac>. Jeśli zostanie zmodyfikowany tylko jeden z naszych pięciu tysięcy plików z przypisami metadanych, generator kodu powinien przetworzyć tylko ten jeden plik.

Proces kompilacji jest spowalniany przez programy antywirusowe. Oznacza to, że wyłączenie takiego programu na czas kompilacji znacznie przyspieszy (skróci) ten proces. Niebezpieczeństwo infekcji niezabezpieczonych komputerów jest jednak na tyle poważne, że wiele firm stosuje politykę, zgodnie z którą oprogramowanie antywirusowe ma pracować bez przerwy.

Jeśli nie możesz wyłączyć programu antywirusowego, powinieneś przynajmniej tak zmienić jego ustawienia, aby ignorował wszystkie pliki zapisane w katalogach kompilacji. Większość narzędzi tego typu oferuje możliwość definiowania plików i katalogów wyłączonych z procesu skanowania. Więcej informacji na ten temat znajdziesz w instrukcji użytkownika wykorzystywanego oprogramowania antywirusowego.

W takim przypadku proces kompilacji obejmowałby następujące kroki:

1. Programista modyfikuje pojedynczy plik źródłowy z przypisami metadanych.
2. XDoclet generuje nowy kod źródłowy na podstawie jedynego zmienionego pliku.
3. Zadanie `<javac>` kompiluje tylko pliki wygenerowane przez program XDoclet w ostatnim cyklu.

Okazuje się jednak, że zadanie `<generate>` przetworzy wszystkie pięć tysięcy plików z przypisami, co oznacza, że zadanie `<javac>` będzie zmuszone do przeprowadzenia bardzo czasochłonnego procesu powtórnej kompilacji znacznej części plików źródłowych.

Problem jest dość poważny. Nasz proces kompilacji nie realizuje założeń kompilacji minimalnych, więc nie powinniśmy oczekiwać, że zatrudniony w firmie ekspert w dziedzinie czasu i ruchu będzie usatysfakcjonowany tym, co zobaczy.

Gdybyśmy korzystali z narzędzia Make, moglibyśmy zdefiniować reguły kompilacji wymuszające ponowne uruchamianie generatora kodu tylko dla zmodyfikowanych plików. Program Ant niestety nie oferuje tak bezpośredniego mechanizmu sterowania procesem kompilacji i — co ważniejsze — takiej możliwości nie obsługuje też odpowiednie zadanie generatora XDoclet.

Tak czy inaczej autorzy Anta zdali sobie sprawę z wagi zależności kompilacji i dodali odpowiedni mechanizm w formie nowego elementu zestawu kluczowych (wbudowanych) zadań. Ponieważ jednak nie jest to domyślny tryb pracy tego narzędzia, korzystanie z tej funkcjonalności wymaga pewnych zabiegów ze strony programisty. Aby jak najlepiej zrozumieć zaimplementowany w narzędziu Ant mechanizm wymuszania zależności kompilacji, zapomnijmy na chwilę o programie XDoclet i przeanalizujmy zupełnie inny przykład.

Definiowanie zależności kompilacji w plikach Anta

Na listingu 12.3 przedstawiono plik kompilacji (*build.xml*) demonstrujący zastosowanie zadania `<uptodate>` do zdefiniowania warunkowej zależności kompilacji pomiędzy dwoma innymi zadaniami. W tym przypadku zależność definiuje upakowywania plików binarnych Javy w pojedynczym pliku JAR.

Listing 12.3. *Plik kompilacji Anta z zależnością warunkową*

```
<project name="depend-example" default="make">

<!-- Kompiluje plik z wykorzystaniem zależności zdefiniowanej
     pomiędzy zadaniami package i compile -->

<property name="src.dir"  location="src"/>
<property name="bin.dir"  location="bin"/>
<property name="dist.dir" location="dist"/>

<target name="compile"
        description="Kompiluje wszystkie źródła Javy">
  <javac srcdir="${src.dir}"
         destdir=" ${bin.dir}" />

  <!-- Sprawdza, czy pliki zostały zaktualizowane -->
  <uptodate property="package.notRequired"
            targetfile="${dist.dir}/app.jar">
    <srcfiles dir="${bin.dir}" includes="**/*.class"/>
  </uptodate>
</target>

<target name="package"
        depends="compile"
        unless="package.notRequired"
        description="Generuje plik JAR">
  <jar destfile="${dist.dir}/app.jar"
       basedir="${bin.dir}"/>
</target>

<target name="clean"
        description="Usuwa wszystkie pliki klas">
  <delete>
    <fileset dir="${bin.dir}"/>
    <fileset dir="${dist.dir}"/>
  </delete>
</target>

<target name="make"
        depends="package"
        description="Kompilacja przyrostowa"/>

<target name="build"
        depends="clean, make"
        description="Ponownie kompiluje wszystkie źródła"/>

</project>
```

Proces kompilacji przedstawiony na listingu 12.3 składa się z dwóch kroków. Po pierwsze, zadanie `<compile>` kompiluje (za pomocą podzadania `<javac>`) cały kod źródłowy znajdujący się w katalogu *src* i kieruje wszystkie skompilowane pliki binarne do katalogu *bin*. Po drugie, zadanie `<package>` kopiuje całą zawartość katalogu *bin* do pojedynczego pliku JAR, który z kolei jest umieszczany w katalogu *dist* (wyznaczonym jako miejsce składowania wersji gotowych do wdrożenia).

Dwa zadania kompilacji na szczycie struktury zadań odpowiadają za wykonywanie następujących kroków:

- Zadanie `<target>` usuwa wszystkie skompilowane pliki, po czym ponownie kompiluje i upakowuje system w pliku JAR.
- Zadanie `<make>` odpowiada za przyrostową kompilację systemu, ale tworzy nową, gotową do wdrożenia wersję tylko w sytuacji, gdy którykolwiek spośród plików źródłowych został zaktualizowany.

Upakowywanie ogromnej liczby plików źródłowych w skompresowanych plikach JAR jest procesem czasochłonnym, zatem rozwiązanie polegające na realizacji tego zadania tylko wtedy, gdy jest to konieczne, jest jak najbardziej uzasadnione. Wykonywanie zadania `<package>` zostało uzależnione od spełnienia warunku w formie atrybutu `unless`.

Atrybut `unless` wymusza na narzędziu Ant pominięcie wykonywania zadania `package` w sytuacji, gdy do wskazanej właściwości przypisano jakąkolwiek wartość. I odwrotnie, atrybut `if` w zadaniu `<target>` nakazuje programowi Ant wykonanie tego zadania, jeśli wartość wskazanej właściwości została ustawiona.

Za pomocą tych właściwości oraz atrybutów `if` i `unless` zadania `<target>` możemy kontrolować wykonywane zadania już na etapie kompilacji. W analizowanym przykładzie (patrz listing 12.3) zadanie `<package>` odwołuje się do właściwości `package.notRequired` i na podstawie jej wartości określa, czy należy wygenerować nowy plik JAR. Wartość tej właściwości jest ustawiana na końcu zadania `<compile>`, od którego zadanie `<package>` jest uzależnione.

W analizowanym pliku kompilacji za ustawianie właściwości `package.notRequired` odpowiada warunkowe zadanie `<uptodate>`. Właściwość identyfikowana przez atrybut `property` jest zmieniana, jeśli plik wejściowy jest nowszy od pliku docelowego — jeśli którykolwiek z plików klas został wygenerowany od czasu ostatniej aktualizacji pliku JAR. W takim przypadku zadanie `<uptodate>` pozostawia właściwość `package.notRequired` niezmienioną, co oznacza, że można przetworzyć zadanie `<package>`.

Oferowana przez program Ant obsługa zadań warunkowych daje programiście możliwość konstruowania wyrafinowanych skryptów, które będą właściwie sterowały procesem kompilacji przyrostowej. Dodawanie zachowań warunkowych do procesu kompilacji może się jednak przyczynić do jego niepotrzebnego komplikowania. Przykład z listingu 12.3 demonstrował jedno z najprostszych rozwiązań w tym zakresie. Typowe procesy kompilacji oprogramowania dla platformy J2EE obejmują znacznie więcej zadań, co oznacza, że wszelkie dodatkowe informacje o zależnościach mogą podnosić poziom złożoności tych procesów.

Aby zapobiec sytuacji, w której dodatkowa logika kompilacji sprawia, że pliki skryptów stają się zbyt skomplikowane i nieczytelne, dobry projekt powinien wymuszać rozbijanie procesu kompilacji na rozłączne bloki (moduły). Wówczas będzie można kompilować poszczególne moduły systemu osobno, zatem definiowanie zależności pomiędzy zadaniami w mniejszych plikach kompilacji będzie dużo prostsze.

Praca z podprojektami

Dobry projekt oprogramowania dzieli wielką aplikację na mniejsze moduły, które charakteryzują się względnie luźnymi związkami z pozostałymi modułami i jednocześnie wysokim poziomem wewnętrznej spójności, integracji. Skrypt kompilacji sam stanowi jeden z artefaktów oprogramowania, zatem podczas projektowania procesu kompilacji mają zastosowanie dokładnie te same reguły — powinniśmy podejmować próby rozbijania wielkich procesów kompilacji na mniejsze, autonomiczne jednostki znane jako podprojekty.

Narzędzie Ant oferuje szereg mechanizmów w zakresie rozbijania wielkich plików kompilacji na mniejsze struktury, których budowa i konserwacja są znacznie prostsze. Jednym z takich rozwiązań jest skonstruowanie centralnego, sterującego pliku kompilacji, który będzie odpowiadał za delegowanie zadań kompilacji do odpowiednich podprojektów. Takie podejście jest możliwe, ponieważ Ant oferuje mechanizm wywoływania zadań znajdujących się w innych plikach kompilacji — do tego celu należy używać zadania `<ant>`.

Przedstawiony poniżej fragment pliku kompilacji ilustruje użycie zadania `<ant>` do wywołania celu zdefiniowanego w zewnętrznym pliku kompilacji *build.xml*:

```
<ant antfile="build.xml"
     dir="${subproject.dir}"
     target="package"
     inheritAll="no">
  <property name="package.dir"
            value="${wls.url}"/>
</ant>
```

Atrybuty zadania `<ant>` określają nazwę zewnętrznego pliku kompilacji, położenie (katalog) tego pliku, nazwę wywoływanego zadania oraz to, czy dany podprojekt będzie miał dostęp do wszystkich właściwości pliku wywołującego. Zgodnie z domyślnymi ustawieniami wywoływany plik kompilacji przejmuje wszystkie właściwości pliku wywołującego (atrybut `inheritAll` domyślnie ma wartość `yes`). W przedstawionym przykładzie wyłączono tę opcję — zdecydowaliśmy się przekazać do wywoływanego pliku tylko wybrane właściwości zdefiniowane w zagnieżdżonym elemencie `<property>`.

Ten sposób rozbijania plików kompilacji większości programistów raczej nie przypadnie do gustu. Część wolałaby utrzymywać wszystkie mechanizmy kompilujące w jednym miejscu. Okazuje się, że Ant oferuje odpowiednie rozwiązanie — zamiast delegować zadania do poszczególnych plików kompilacji, można je włączać do głównego pliku (automatycznie w czasie kompilacji wydobywać definicje zadań z pozostałych plików i zawierać w głównym pliku).

Twórcy Anta zaimplementowali dwa mechanizmy dołączania funkcjonalności zdefiniowanej w zewnętrznych plikach. Pliki kompilacji Anta są przede wszystkim dokumentami XML, zatem można wykorzystać potencjał języka XML do wydobywania fragmentów plików kompilacji. Praktyczny przykład zastosowania takiego podejścia przedstawiono na listingu 12.4.

Listing 12.4. *Włączanie fragmentów definicji kompilacji do pliku kompilacji*

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

<!DOCTYPE project [
  <!ENTITY properties SYSTEM "file:./config/properties.xml">
  <!ENTITY libraries SYSTEM "file:./config/libraries.xml">
]>

<project name="include-example" default="make" basedir=".">

  &properties;
  &libraries;

  .
  .
  .

</project>
```

Przykład przedstawiony na listingu 12.4 pokazuje, jak można wstawiać zawartość dwóch plików (w tym przypadku *properties.xml* i *libraries.xml*) bezpośrednio do pliku kompilacji (w tym przypadku *build.xml*). Docelowe położenie zawartości dołączanych plików oznaczono odpowiednio odwołaniami `&properties` i `&libraries`.

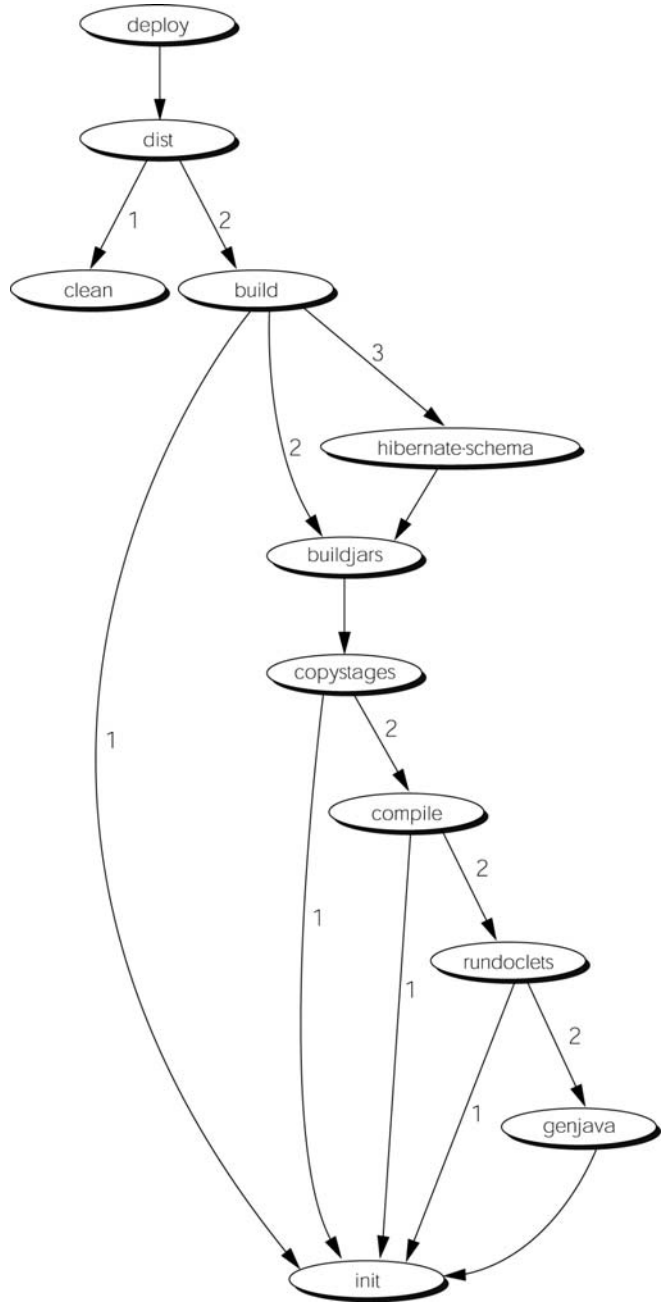
Ant 1.6 oferuje jeszcze drugą metodę wydobywania i wstawiania do pliku kompilacji zawartości plików zewnętrznych. Zadanie `<import>`, za pomocą którego można wstawiać (importować) całe pliki kompilacji, ma tę przewagę nad techniką przedstawioną w poprzednim przykładzie, że dodatkowo umożliwia dziedziczenie zadań importowanych plików kompilacji przez pliki docelowe. Oznacza to, że możemy — w razie konieczności — przykrywać zadania importowanych plików kompilacji.

Przeglądanie zależności kompilacji

Zależności wewnątrz plików kompilacji mogą być bardzo skomplikowane i zawiłe. Na szczęście znalazł się człowiek, Eric Burke, który opracował program **Antgraph** — bardzo wygodne narzędzie z otwartym dostępem do kodu źródłowego, które pozwala przeglądać pliki kompilacji Anta w czytelnej formie graficznej. Program Antgraph można pobrać za darmo z witryny internetowej Erica Burke'a (patrz *http://www.ericburke.com/downloads/antgraph/*). Oferowane oprogramowanie jest łatwe w instalacji i uruchamianiu; na wspomnianej witrynie znajdziesz wersje dla wielu platform.

Na rysunku 12.1 przedstawiono przykład diagramu (grafu) wygenerowanego i wyświetlonego przez program Antgraph.

Graf z rysunku 12.1 jest graficzną reprezentacją pliku kompilacji Anta dołączonego do narzędzia AndroMDA i przygotowanego z myślą o budowie projektu wykorzystującego generator kodu Hibernate. Przedstawiony diagram zawiera wszystkie zadania



Rysunek 12.1. *Informacje o zależnościach zawarte w pliku kompilacji Anta i wyświetlone przez narzędzie Antgraph*

zdefiniowane we wspomnianym pliku kompilacji (strzałki reprezentują hierarchię wywoływania tych zadań). Cyfry przy strzałkach reprezentują liczbę zależności dla każdego z celów. Przykładowo, zadanie `build` jest zależne od trzech innych zadań: `init`, `buildjars` i `hibernate-schema`.

Program Antgraph bazuje na darmowym module graficznym **Graphviz**, który doskonale radzi sobie z generowaniem grafów zależności. Graphviz obsługuje wiele różnych rodzajów grafów, zatem możesz samodzielnie poeksperymentować z jego rozmaitymi opcjami. W grafie przedstawionym na rysunku 12.1 użyto co prawda standardowego modułu układu **dot**, jednak równie dobrze mogliśmy wykorzystać moduły **neato** i **twopi**. Być może po wykonaniu kilku prób uznasz, że któryś z tych modułów odpowiada Ci bardziej niż standardowy **dot**.

Standardowe zadania kompilacji

Dobrym rozwiązaniem jest stosowanie wspólnych standardów dla wszystkich plików kompilacji wykorzystywanych w projektach. Używanie jednej konwencji nazewnictwa zadań kompilacji daje nam pewność, że wszystkie pliki kompilacji wykorzystywane w różnych projektach są logicznie spójne oraz prostsze w analizie i konserwacji dla członków zespołów projektowych. W tabeli 12.1 przedstawiono propozycję nazw dla najbardziej popularnych zadań kompilacji Anta.

Tabela 12.1. *Zalecenia w zakresie nazewnictwa zadań kompilacji programu Ant*

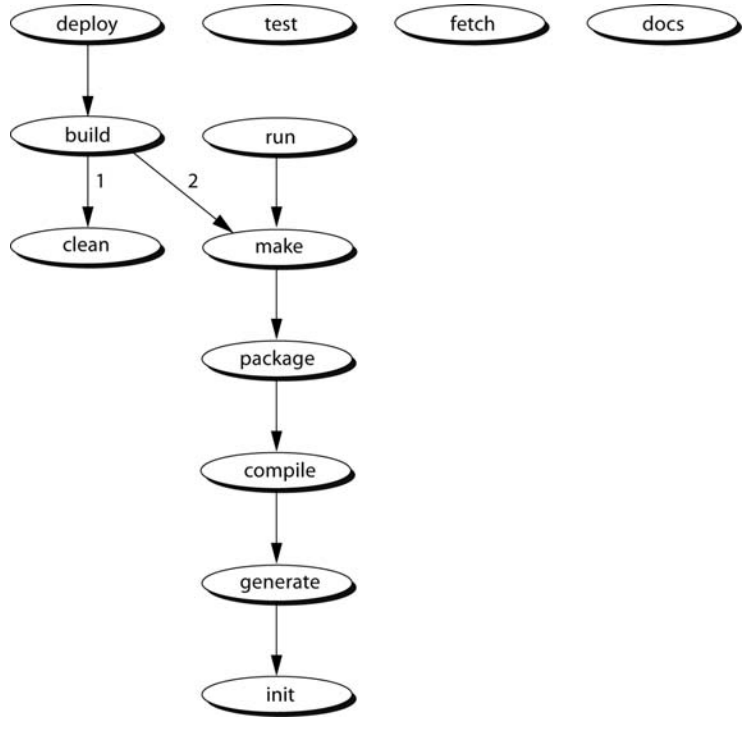
Nazwa	Opis
`init`	Uniwersalne zadanie pomocnicze wspomagające realizację rozmaitych czynności konfiguracyjnych, w tym tworzenie katalogów, kopiowanie plików czy weryfikacja poprawności zależności. Zadanie `init` rzadko jest wywoływane z poziomu wiersza poleceń — znacznie częściej wywołują go pozostałe zadania Anta.
`clean`	Zadanie `clean` usuwa wszystkie wygenerowane artefakty kompilacji i — tym samym — gwarantuje, że każdy kolejny proces kompilacji wygeneruje te artefakty ponownie.
`compile`	Zadanie `compile` kompiluje wszystkie pliki źródłowe. Wykonywanie tego zadania może być uzależnione od zadania `generate`, które wywołuje wszystkie generatory kodu będące częścią procesu kompilacji.
`package`	Zadanie `package` otrzymuje na wejściu wszystkie pliki wygenerowane w fazie kompilacji (w zadaniu `compile`) i upakowuje je w pliku gotowym do wdrożenia. Zadanie `package` jest czasem oznaczane nazwą `dist`.
`make`	Celem zadania `make` jest inicjowanie kompilacji przyrostowych. Zadanie `make` wymaga prawidłowego skonfigurowania warunkowych zależności pomiędzy różnymi zadaniami kompilacji.
`build`	Wykonuje kolejno zadania `clean` i `make`, zatem daje programiście pewność, że cała aplikacja została skompilowana od zera.
`deploy`	Wdraża skompilowaną aplikację na wskazanym serwerze aplikacji. Równie przydatne jest zadanie `undeploy`, które usuwa aplikację z serwera.
`test`	Zadanie `test` powinno być wykorzystywane do uruchamiania wszystkich testów jednostkowych. Oznacza to, że zadanie `test` należy wykonywać regularnie (najlepiej w ramach szerszego procesu conocnej kompilacji).
`docs`	Generuje dokumentację projektu w standardzie JavaDoc.
`fetch`	Wydobywa najnowszą wersję projektu z systemu kontroli wersji. Być może lepszym rozwiązaniem będzie zdefiniowanie zadania, które będzie od razu wydobywało najnowszą wersję kodu, wykonywało kompletną kompilację i przeprowadzało wszystkie testy jednostkowe.

Nazwy zasugerowane w tabeli 12.1 dotyczą tylko minimalnego zestawu zadań kompilacji; Twój projekt najprawdopodobniej będzie wykorzystywał znacznie bogatszy zbiór zadań. Przykładowo, mógłbyś użyć zadania uruchamiającego wszystkie aplikacje klienckie oraz zadań `start` i `stop` do kontrolowania pracy serwera aplikacji.

Stosowanie spójnej konwencji nazewnictwa zadań kompilacji okazuje się jeszcze ważniejsze w programie Ant 1.6, gdzie zadanie `<import>` umożliwia przykrywanie zadań zdefiniowanych w importowanych plikach kompilacji przez zadania pliku dziedziczącego.

Zawsze staraj się w miarę dokładnie opisywać przeznaczenie wszystkich zadań definiowanych w pliku kompilacji. Używaj do tego celu atrybutu `description` elementu `target`. Takie rozwiązanie umożliwi potem uruchamianie Anta z opcją `-projecthelp`, która spowoduje wyświetlenie listy wszystkich celów pliku kompilacji wraz z dołączonymi opisami.

Istnieje możliwość określania zależności pomiędzy każdym z zadań kompilacji. Na rysunku 12.2 przedstawiono graf wygenerowany przez program Antgraph dla celów zdefiniowanych w tabeli 12.1.



Rysunek 12.2. *Hierarchia zadań Anta w postaci grafu wygenerowanego przez program Antgraph*

W przypadku większych projektów złożonych z wielu modułów należy dokładnie rozważyć sposób, w jaki te projekty są zorganizowane jako całość. W następnym podrozdziale przedstawimy kilka wskazówek w tym zakresie.

Organizacja projektu

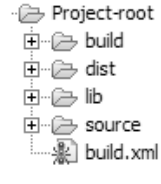
Dobrze zorganizowana struktura katalogowa projektu upraszcza nie tylko samo kodowanie, ale także zarządzanie procesem kompilacji. Istnieje wiele czynników decydujących o strukturze procesu kompilacji — należą do nich między innymi wymagania docelowego serwera aplikacji oraz narzędzi wytwarzania wykorzystywanych przez zespół projektowy. Mimo tych różnic rozwiązania prezentowane w tym podrozdziale stanowią uniwersalny zarys tego, do czego należy dążyć w naszych środowiskach projektowych.

Struktura projektu komplikuje się wraz ze wzrostem rozmiarów wytwarzanej aplikacji. Tylko zachowując maksymalną prostotę stosowanych struktur od samego początku możemy choć w części uniknąć niepotrzebnej złożoności podczas dalszego rozwoju projektu.

We wszystkich projektach platformy J2EE struktura katalogów jest dzielona na cztery osobne obszary: plików źródłowych, plików skompilowanych, bibliotek i plików gotowych do wdrożenia. Każdy z tych obszarów jest reprezentowany w systemie plików przez własny katalog, a cztery katalogi wymienionych obszarów znajdują się na szczycie hierarchii katalogowej projektu.

Na rysunku 12.3 przedstawiono strukturę katalogów na najwyższym poziomie hierarchii katalogowej projektu.

Rysunek 12.3. *Struktura katalogów na najwyższym poziomie projektu*



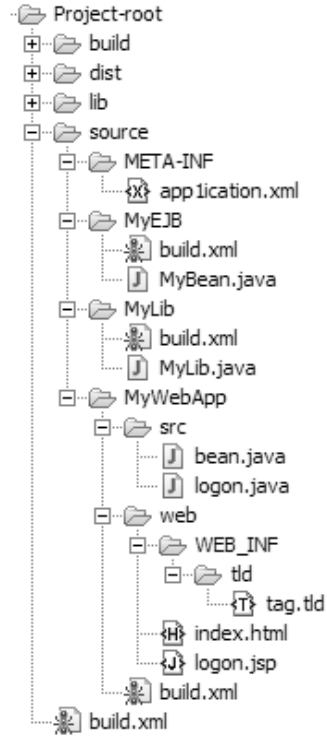
Naszą analizę rozpoczniemy od szczegółowego omówienia katalogu źródłowego (*source*).

Katalog źródłowy (source)

Katalog źródłowy zawiera wszystkie artefakty programowe, które są wykorzystywane w procesie konstruowania aplikacji. Nie chodzi tu wyłącznie o pliki źródłowe Javy — mogą to być takie artefakty jak skrypty kompilacji, pliki właściwości, deskryptory wdrożenia, skrypty baz danych czy schematy XML. Organizację katalogu źródłowego przedstawiono na rysunku 12.4.

Taka struktura projektu ma ścisły związek z wymaganiami w zakresie tworzenia pojedynczego pliku EAR, gdzie każdy gotowy do wdrożenia moduł jest utrzymywany w formie podprojektu.

Rysunek 12.4.
Struktura katalogu źródłowego



Cały kod źródłowy powinien się znajdować w odpowiednich pakietach w strukturze katalogowej. Warto też rozważyć dodanie równoległego katalogu źródłowego dla każdego z komponentów, co może bardzo ułatwić zarządzanie kodem dla testów jednostkowych.

Analiza struktury katalogowej przedstawionej na rysunku 12.4 pozwala zidentyfikować następujące moduły:

- Katalog *META-INF* zawiera wszystkie pliki potrzebne do zbudowania gotowego do wdrożenia pliku EAR.
- Moduły EJB zostały umieszczone w osobnym katalogu. Plik EAR może obejmować wiele modułów. Ze struktury przedstawionej na rysunku 12.4 wynika, że plik EAR będzie zawierał pojedynczy moduł (komponent) EJB: `MyEJB`.
- Wspólne biblioteki, które są kompilowane jako część projektu, powinny być składowane w ramach struktury katalogowej (patrz biblioteka `MyLib` na rysunku 12.4).
- Struktura katalogowa aplikacji internetowych jest najbardziej skomplikowana, ponieważ na każdą z tych aplikacji składa się wiele typów plików. W analizowanym przykładzie można wyróżnić aplikację internetową `MyWebApp`, która dobrze demonstruje układ i organizację poszczególnych typów plików.

Dla każdego modułu składającego się na wersję systemu gotową do wdrożenia na serwerze aplikacji J2EE należy zdefiniować osobny plik *build.xml*. Pliki kompilacji są autonomicznymi strukturami, za pomocą których można kompilować poszczególne moduły niezależnie od siebie. Pliki kompilacji zawierają oczywiście zapisy reprezentujące relacje łączące różne moduły systemu. Aplikacja internetowa najprawdopodobniej będzie się odwoływała do warstwy biznesowej, co oznacza, że będzie istniała zależność łącząca tę aplikację z modułem EJB. Wszystkie tego rodzaju związki powinny być zarządzane właśnie przez pliki kompilacji.

Wszystkie pliki kompilacji (dotyczące różnych modułów) powinny być spójne przynajmniej w obszarze nazewnictwa zadań. Każdy z tych plików musi wykorzystywać możliwie podobny zbiór zadań kompilacji. Na szczycie hierarchii (w głównym katalogu źródłowym) istnieje plik kompilacji, który odpowiada za sterowanie całym procesem. Jest to typowy przykład pliku Anta, który **deleguje** odpowiednie zadania do plików kompilacji poszczególnych podprojektów.

Wywołania zadań użyte w pliku kompilacji projektu są kolejno przekazywane w dół hierarchii do właściwych zadań poszczególnych podprojektów. Chociaż każdy plik kompilacji zawiera ten sam zbiór zadań, podejmowane działania są uzależnione od typów modułów, za których kompilowanie odpowiadają. Przykładowo, w przypadku modułu aplikacji internetowej zadanie `package` generuje plik WAR, natomiast w przypadku pliku kompilacji projektu na najwyższym poziomie hierarchii to samo zadanie generuje plik EAR.

Bardzo istotna jest kolejność kompilowania podprojektów — proces powinien się rozpoczynać od modułów na najniższym poziomie hierarchii i kończyć na jej szczycie (np. wygenerowaniem zbiorczego pliku EAR). Właśnie podejście „od dołu” i bezwzględne wymuszanie przestrzegania zależności prowadzi do osiągnięcia procesu kompilacji na poziomie gwarantującym łatwość konserwacji i logiczną spójność budowanego systemu.

Katalog bibliotek (lib)

Katalog bibliotek zawiera wszystkie zewnętrzne biblioteki. Skrypty kompilacji często odwołują się do tego katalogu podczas kompilowania modułów systemu. Podczas gromadzenia zasobów składających się na gotowy do wdrożenia plik EAR wszystkie biblioteki, których obecność w czasie wykonywania programu jest niezbędna, są kopiowane z katalogu *lib*.

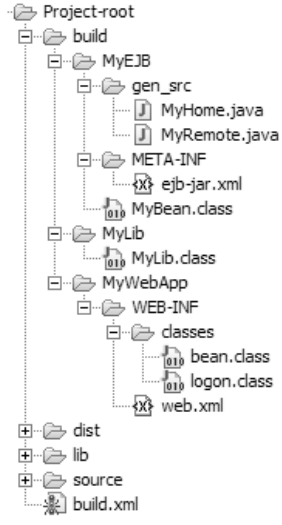
Katalog plików skompilowanych (build)

Wszystkie dane wyjściowe procesu kompilacji plików z katalogu źródłowego (*source*) trafiają do katalogu plików skompilowanych (*build*). Utrzymywanie plików źródłowych z dala od skompilowanych plików binarnych pozwala zachować należyty porządek w strukturze katalogowej projektu. Taki podział na dwa różne katalogi oznacza, że katalog plików skompilowanych może np. zostać w dowolnym momencie usunięty (co oczywiście znacznie upraszcza implementowanie zadania `clean`).

Struktura katalogu plików skompilowanych powinna odzwierciedlać strukturę katalogu źródłowego, ponieważ takie rozwiązanie znacznie uprości zarządzanie i odnajdywanie wygenerowanych plików. Uważa się, że dobrą praktyką jest kierowanie do katalogu *build* wszystkich wyników procesu kompilacji (włącznie z danymi wyjściowymi generatorów kodu, czyli np. deskryptorów wdrożenia czy zdalnych interfejsów). Najprostszym rozwiązaniem jest umieszczanie wszystkich plików wynikowych pełnego procesu kompilacji w katalogu plików skompilowanych, niezależnie od tego, czy są to pliki z kodem źródłowym czy pliki binarne.

Na rysunku 12.5 przedstawiono strukturę katalogu plików skompilowanych odpowiadającego strukturze projektu zdefiniowanej wcześniej dla plików źródłowych (patrz rysunek 12.4).

Rysunek 12.5. *Struktura katalogu plików skompilowanych*

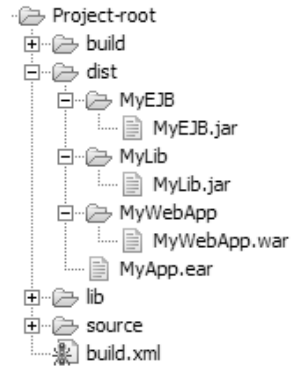


Po skompilowaniu wszystkich niezbędnych składników projektu kolejnym krokiem jest upakowanie komponentów gotowych do wdrożenia na serwerze aplikacji. Okazuje się, że najwygodniej jest użyć jeszcze jednego katalogu.

Katalog plików gotowych do wdrożenia

Katalog plików gotowych do wdrożenia jest bardzo pomocny w procesie upakowywania i wdrażania aplikacji. Istnieje co prawda możliwość realizowania tych zadań wewnątrz katalogu zawierającego skompilowane pliki, jednak stosowanie osobnego katalogu ze skompresowanymi (upakowanymi) plikami znacznie upraszcza procesy generowania formalnych wydań (wersji aplikacji) i przekazywania nowych wersji systemu do formalnego środowiska testowego.

Na rysunku 12.6 przedstawiono przykładową strukturę katalogu plików gotowych do wdrożenia. Także w tym przypadku prezentowana struktura modułów odpowiada przeanalizowanym w poprzednich punktach katalogom z plikami źródłowymi i plikami skompilowanymi.



Rysunek 12.6. *Katalog plików gotowych do wdrożenia*

Włączanie katalogu plików gotowych do wdrożenia do głównego katalogu projektu jest rozwiązaniem opcjonalnym. Może się okazać, że z perspektywy członków zespołu projektowego lepiej będzie umieszczać pliki generowane w procesie upakowywania bezpośrednio w strukturze katalogowej serwera aplikacji i — tym samym — w pełni wykorzystać oferowane przez ten serwer mechanizmy wdrażania „na gorąco". W niektórych przypadkach serwer aplikacji może sam żądać wdrożenia pewnych modułów w formacie nieskompresowanym, zwalniając inżynierów oprogramowania z odpowiedzialności za czasochłonny proces upakowywania gotowych plików.

Prezentowana struktura projektu jest tylko pewną ilustracją możliwych rozwiązań. Wymagania Twojego projektu mogą wymuszać stosowanie nieco innej techniki procesu kompilacji, zatem zawsze powinieneś projektować strukturę projektu z uwzględnieniem konkretnych uwarunkowań.

Integracja ze środowiskami IDE

Chociaż możliwość kompilowania całej aplikacji z poziomu wiersza poleceń jest bardzo ważna, stosowanie skryptów kompilacji nie powinno wykluczać korzystania z wygodnych funkcji Twojego ulubionego zintegrowanego środowiska programowania (ang. *Integrated Development Environment — IDE*) w tym zakresie. Mechanizmy współpracy środowisk IDE ze skryptami kompilacji są oczywiście ważne, jednak równie istotna jest relacja odwrotna, czyli możliwość wykorzystywania funkcjonalności tych środowisk w samych skryptach kompilacji.

Zalety korzystania ze zintegrowanych środowisk wytwarzania oprogramowania omówimy w rozdziale 13.

Zintegrowane środowiska wytwarzania oprogramowania nie są niestety dobrymi narzędziami kompilującymi. Zwykle obsługują tylko niewielkie podzbiory zadań kompilacji, które stosuje się w większości projektów. Tego typu narzędzia zazwyczaj mogą realizować takie zadania jak kompilacja czy upakowywanie skompilowanych plików, ale nie obsługują wszystkich zadań kompilacji stosowanych w procesie wytwarzania

oprogramowania korporacyjnego. Dla porównania, dobre skrypty kompilacji mogą realizować takie działania jak automatyczne przypisywanie plikom JAR odpowiednich numerów wersji, upakowywanie aplikacji w formacie umożliwiającym ich błyskawiczne wdrażanie, przekazywanie aplikacji bezpośrednio do środowiska testowego, wykonywanie testów jednostkowych itd.

Środowiska IDE nie wykonują wszystkich najważniejszych zadań składających się na szeroko rozumiany proces kompilacji projektu, okazuje się jednak, że z pewnymi działaniami radzą sobie znakomicie. Jednym z przykładów jest kompilacja wszystkich plików źródłowych. Inną przydatną i dość popularną funkcją zintegrowanych środowisk programowania jest stałe kompilowanie i sprawdzanie składni kodu już w czasie jego pisania. Tego typu mechanizmy pozwalają bardzo szybko wykrywać błędy i znacznie poprawiają produktywność programistów.

Aby obsługa tego typu mechanizmów była możliwa, zintegrowane środowisko wytwarzania oprogramowania należy odpowiednio skonfigurować — przekazać wszystkie informacje niezbędne do prawidłowego kompilowania aplikacji. Jeśli dodatkowo wykorzystujemy skrypt kompilacji, możemy przypadkowo powielić pewne czynności. Takie powtarzanie kompilacji wiąże się nie tylko z niepotrzebnymi opóźnieniami i nadmiarem pracy, ale także podnosi ryzyko wystąpienia niespójności kompilacji, ponieważ różne podejścia do tego procesu mogą prowadzić do niezgodnych charakterystyk budowanej aplikacji.

Jednym ze sposobów ominięcia tego problemu jest wymuszenie na narzędziu Ant korzystania z funkcjonalności środowiska IDE w zakresie kompilowania plików źródłowych. Taka metoda powoduje, że kompilacje realizowane przez samo zintegrowane środowisko programowania będą identyczne jak te realizowane przez skrypt kompilacji.

Aby takie rozwiązanie mogło prawidłowo funkcjonować, środowisko IDE musi oferować możliwość wywoływania procesu kompilacji z poziomu poleceń powłoki, najlepiej bez konieczności uruchamiania całego graficznego interfejsu użytkownika tego środowiska.

Poniżej przedstawiono sposób wykorzystania zadania `<exec>` skryptu narzędzia Ant do kompilowania plików źródłowych projektu budowanego w środowisku JBuilder firmy Borland:

```
<exec dir="."
      executable="${jbuilder_home.dir}/jbuilder.exe"
      failonerror="true">
  <arg lint="-build myproject.jpx make"/>
</exec>
```

Takie rozwiązanie jest możliwe dzięki zaimplementowaniu odpowiedniego mechanizmu w środowisku JBuilder i prawidłowej konfiguracji ścieżki do klas projektu tego środowiska. W ten sposób można wyeliminować konieczność powielania ustawień kompilacji w pliku projektu środowiska IDE i pliku kompilacji narzędzia Ant. Jeśli nie jesteś pewien, czy (i jak) Twoje ulubione środowisko programowania oferuje funkcję wymuszania procesu kompilacji z poziomu wiersza poleceń, zapoznaj się z jego dokumentacją.

Rozszerzanie Anta o obsługę języka Jython

Chociaż Ant oferuje bogaty zbiór wbudowanych zadań, które wystarczają do prawidłowego wykonywania większości operacji związanych z kompilowaniem systemów korporacyjnych, podczas projektowania swojego środowiska pracy być może miałeś do czynienia z przypadkami, w których istniejące zadania Anta okazały się niewystarczające. W takich sytuacjach najlepszym rozwiązaniem jest zdefiniowanie własnych zadań.

Zadania programu Ant są w zdecydowanej większości implementowane w Javie. Jeśli jednak któreś z zadań wymaga obsługi języków skryptowych, warto rozważyć użycie Jythona zamiast tego konwencjonalnego języka programowania. Zaletą takiego rozwiązania jest nie tylko możliwość wykorzystania zalet języka skryptowego Jython w zakresie błyskawicznego wytwarzania kodu, ale także dostęp do bogatej biblioteki funkcji Jythona.

Podstawy języka skryptowego Jython wprowadzono w rozdziale 9.

Tworzenie nowego zadania Anta

Proces implementacji prostego zadania narzędzia Ant składa się z następujących kroków:

1. Zdefiniowanie klasy Javy, która będzie rozszerzała klasę bazową `org.apache.tools.ant.Task`.
2. Zaimplementowanie publicznych metod ustawiających wszystkie atrybuty nowego zadania.
3. Zaimplementowanie publicznej metody `execute()`.

Na listingu 12.5 przedstawiono implementację zadania Anta napisaną w języku skryptowym Jython. Nowe zadanie ma tylko jeden atrybut: `message`. Wykonanie tego zadania z poziomu pliku kompilacji narzędzia Ant spowoduje wyświetlenie zarówno komunikatu zawartego w tym atrybucie, jak i informacji na temat zależności zadania macierzystego.

Listing 12.5. *Zadanie Anta napisane w języku Jython (RapidTask.py)*

```
from org.apache.tools.ant import Task
from org.apache.tools.ant import Target

class RapidTask(Task):

  # Przykrywa metodę Task.execute().
  #
  def execute(self):
    "@sig public void execute()"
```

```
    # Wyświetla właściwości tego zadania.
    #
    self.log('Nazwa zadania: ' + self.taskName)
    self.log('Opis: ' + self.description)

    # Wyświetla wartość właściwości message.
    #
    self.log('Komunikat: ' + self.message)

    # Odczytuje informacje o zadaniu macierzystym i wyświetla jego zależności.
    #
    target = self.owningTarget

    self.log('Nazwa zadania macierzystego: ' + target.name)

    for dependency in target.dependencies:
      self.log('\tZależy od: ' + dependency)

  # Metoda ustawiająca atrybut Anta.
  #
  def setMessage(self, message):
    "@sig public void setMessage(java.lang.String str)"
    self.message = message
```

Możesz się posługiwać tym przykładem jako swoistym szablonem dla zadań Anta tworzonych za pomocą języka skryptowego Jython.

Kompilowanie klas Jythona

Klasa Jythona, którą przedstawiono na listingu 12.5, różni się w kilku punktach od tradycyjnej klasy tego języka — różnice mają ścisły związek z wymaganiami Anta, który będzie z tej klasy korzystał. Kompilowanie kodu języka Jython na postać standardowego kodu bajtowego Javy wiąże się z konieczności przejścia ze świata Jythona, w którym nie istnieje pojęcie bezpieczeństwa typów, do bardziej wymagającego świata Javy, gdzie zgodność jest ściśle przestrzegana. Taka transformacja rodzi pewne problemy, szczególnie w kwestii definiowania metod Javy, które będą odpowiadały pozbawionym sygnatur metodom Jythona.

Skutecznym obejściem tego problemu jest umieszczenie w przestrzeni nazw `__doc__` każdej z metod Jythona łańcucha definiującego właściwą sygnaturę dla docelowego kodu bajtowego Javy. Poniżej przedstawiono osadzone w ten sposób sygnatury metod klasy `RapidTask` (odpowiednio `setMessage()` i `execuce()`):

```
"@sig public void setMessage(java.lang.String str)"
"@sig public void execute()"
```

Preambuła `@sig` umieszczona na początku każdego z tych łańcuchów sygnalizuje kompilatorowi `jythonc`, że ma do czynienia z docelową sygnaturą metody Javy. Kompilator Jythona dysponuje teraz wszystkimi informacjami niezbędnymi do kompilacji naszego skryptu — wygenerowania kodu bajtowego klasy `RapidTask`.

Przedstawioną klasę można skompilować za pomocą następującego polecenia:

```
jythonc -a -c -d -j rapidTask.jar RapidTask.py
```

Parametry `-a`, `-c` i `-d` instruują kompilator `jythonc`, że skompilowana klasa powinna zawierać wszystkie kluczowe biblioteki Jythona. Opcja `-j rapidTask.jar` wymusza umieszczenie skompilowanych klas we wskazanym pliku JAR (w tym przypadku *rapidTask.jar*).

Po pomyślnym skompilowaniu nowego zadania, warto sprawdzić jego działanie z poziomu pliku kompilacji Anta.

Testowanie nowego zadania

Nowe zadanie dodano do skryptu kompilacji za pomocą elementu `<taskdef>`, który definiuje zarówno nazwę tego zadania, jak i wczytywaną klasę. Na listingu 12.6 przedstawiono fragment przykładowego pliku kompilacji wykorzystującego nasze nowe zadanie Anta (napisane w Jythonie).

Listing 12.6. *Testowy plik kompilacji dla zadania Anta napisanego w języku Jython*

```
<target name="test"
        depends="clean, package"
        description="Demonstruje dostęp do zadania zaimplementowanego w Jythonie">

  <!-- Deklaruje nowe zadanie Jythona. -->
  <taskdef name="Rapid"
           classname="RapidTask">
    <classpath>
      <pathelement location="${task.jar}"/>
    </classpath>
  </taskdef>

  <!-- Ustawia właściwość tego zadania. -->
  <Rapid description="Przykładowe zadanie"
         message="To jest moje pierwsze zadanie Jythona." />

</target>
```

Wywołanie zadania `<test>` spowoduje wygenerowanie przez program Ant następujących danych wyjściowych:

```
test:
    [Rapid] Nazwa zadania: Rapid
    [Rapid] Opis: Przykładowe zadanie
    [Rapid] Komunikat: To jest moje pierwsze zadanie Jythona.
    [Rapid] Nazwa zadania macierzystego: test
    [Rapid]     Zależy od: clean
    [Rapid]     Zależy od: package

BUILD SUCCESSFUL
```

Nasze zadanie Anta wykorzystuje metody klasy `Task` do odczytywania takich informacji na temat jego środowiska jak nazwa zadania, nazwa zadania macierzystego oraz lista wszystkich zależności. Przedstawiony na listingu 12.6 fragment skryptu wyświetla też wartość atrybutu `message`, który jest częścią naszego zadania.

Tych kilka prostych operacji w zupełności wystarczy do zbudowania zadania Anta w języku skryptowym Jython. Połączenie Anta i Jythona pozwala jednocześnie korzystać ze struktury i możliwości kontroli tego pierwszego oraz z potencjału błyskawicznego wytwarzania kodu języka skryptowego — jest to doprawdy doskonała kombinacja prowadząca do szybkiego konstruowania procesów kompilacji.

Podsumowanie

Proces kompilacji jest tylko częścią szerszego procesu wytwarzania oprogramowania. Tak czy inaczej częstotliwość, z jaką kompilujemy budowany system, powoduje, że nawet niewielkie uchybienia w tym procesie mogą być źródłem znacznych opóźnień w realizacji całego projektu.

Zaprojektowanie uniwersalnych procesów kompilacji, które będą stosowane przez wszystkie zespoły projektowe, jest ważnym krokiem na drodze do budowy fundamentu adaptacyjnego pod błyskawiczne wytwarzanie oprogramowania. Jeśli już na początku projektu dysponujemy optymalnym procesem kompilacji, który dostosowano do potrzeb realizowanych zadań, możemy oczekiwać znacznych oszczędności czasowych.

Wysiłek związany z definiowaniem uniwersalnego procesu kompilacji należy traktować jak dobrą inwestycję w pomyślność przyszłych projektów, zatem nie powinieneś lekceważyć tego bardzo ważnego zadania. Pamiętaj, że proces kompilacji należy traktować jak analizę czasu i ruchu, która musi prowadzić do eliminowania wszelkich niepotrzebnych kroków.

W tym rozdziale szczegółowo omówiliśmy proces kompilacji, w następnym przeanalizujemy znaczenie narzędzi wspierających wytwarzanie oprogramowania znajdujących się w arsenale współczesnych inżynierów oprogramowania.

Informacje dodatkowe

Ant jest narzędziem zaprojektowanym z myślą o kompilowaniu oprogramowania. Członkowie projektu Apache wybrali Anta jako rozwiązanie bazowe dla swojego programu **Maven**, które ma być narzędziem wspierającym zarządzanie projektami, kompilacjami i wdrożeniami. Maven oferuje formalny szkielet dla wytwarzania artefaktów oprogramowania i bezwzględnie wymusza przestrzeganie zależności pomiędzy zadaniami kompilacji.

Więcej informacji na temat Mavena można znaleźć na witrynie internetowej *http://maven.apache.org*. Na witrynie tej udostępniono najnowszą wersję tego programu.

Ważną częścią środowiska wytwarzania systemów informatycznych, którą całkowicie pominięto w tym rozdziale, jest mechanizm integrujący fragmenty oprogramowania tworzone przez poszczególnych członków zespołu projektowego. Niewłaściwe łączenie opracowanych fragmentów funkcjonalności w wielu przypadkach może prowadzić do błędów procesu kompilacji, czyli jednego z najpoważniejszych źródeł opóźnień w realizacji projektów.

Dobrym rozwiązaniem problemu integracji oprogramowania jest wdrożenie polityki ustawicznej integracji. Idea ustawicznej integracji przewiduje możliwie częste (być może nawet kilka razy dziennie) scalenie modyfikowanych składników oprogramowania. Autorami tej koncepcji są Martin Fowler i Matthew Foemmel, którzy napisali ciekawy artykuł na ten temat (patrz strona internetowa *http://www.martinfowler.com/articles/continuousIntegration.html*).

Istnieje narzędzie implementujące koncepcję ustawicznej integracji — oferowany z otwartym dostępem do kodu źródłowego szkielet **CruiseControl**, który można zintegrować z istniejącym systemem kompilacji i który zawiera nie tylko mechanizmy kompilowania oprogramowania w regularnych odstępach czasu, ale także funkcje raportowania o wynikach kompilacji. Program CruiseControl można pobrać z witryny internetowej *http://cruisecontrol.sourceforge.net*.

I wreszcie wypada wspomnieć o czymś, co zainteresuje wszystkich programistów, którzy chcą poeksperymentować z **plikami make** — kopię narzędzia Make (udostępnianą zgodnie z licencją GNU) można pobrać ze strony *http://www.gnu.org/software/make*.