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. Stosowanie wzorców projektowych

Autorzy: William Crawford, Jonathan Kaplan
Tłumaczenie: Jaromir Senczyk
ISBN: 83-7361-428-1
Tytuł oryginału: J2EE Design Patterns
Format: B5, stron: 392
Przykłady na ftp: 208 kB

Wzorce projektowe to opisy poprawnych rozwiązań problemów, na które napotkali programiści w swojej pracy. Pozwalają uniknąć pracy nad rozwiązaniem zagadnienia, które już dawno zostało rozwiązane. Jednak nawet największy zestaw wzorców projektowych jest nieprzydatny, jeśli nie wiadomo, jak zastosować je w określonym zadaniu. Wiedza o tym, że wzorzec istnieje bez umiejętności zaimplementowania go jest bezużyteczna.

Książka „J2EE. Stosowanie wzorców projektowych" zawiera nie tylko opisy wzorców, ale również sposoby ich implementacji w aplikacjach J2EE. Czytelnik znajdzie tu omówienie wzorców dotyczących wydajności, skalowalności i elastyczności aplikacji oraz wzorców ściśle związanych z tworzeniem aplikacji biznesowych.
Książka przedstawia również nowe wzorce dla mechanizmów dystrybucji komunikatów i trwałości.

W książce omówiono:

- Podstawowe zasady tworzenia aplikacji biznesowych w Javie.
- Język UML jako uniwersalne narzędzie do modelowania aplikacji.
- Wzorce dla warstwy prezentacji.
- Wzorce dla warstwy logiki biznesowej.
- Wzorce komunikacji pomiędzy warstwami.
- Wzorce dystrybucji komunikatów.
- Przykłady błędnych wzorców.

Największą zaletą książki jest to, że przedstawia zastosowanie wzorców projektowych do tworzenia aplikacji biznesowych. Jeśli zajmujesz się tworzeniem aplikacji J2EE, to ta książka jest dla Ciebie lekturą obowiązkową.

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

Spis treści

Przedmowa 9

Rozdział 1. Projektowanie aplikacji biznesowych na platformie Java 15

Wzorce projektowe 15

J2EE 18

Warstwy aplikacji 22

Podstawowe koncepcje procesu wytwarzania oprogramowania 25

W perspektywie 32

Rozdział 2. Język UML 33

Geneza języka UML 34

Siedmiu Wspaniałych 35

UML i cykl rozwoju oprogramowania 36

Diagramy przypadków użycia 37

Diagramy klas 41

Diagramy interakcji 47

Diagramy aktywności 50

Diagramy wdrożenia 52

Rozdział 3. Architektura warstwy prezentacji 53

Warstwa prezentacji po stronie serwera 54

Struktura aplikacji 55

Zastosowanie centralnego kontrolera 65

Rozdział 4. Zaawansowane rozwiązania warstwy prezentacji 83
Wielokrotne użycie i aplikacje internetowe 84
Rozbudowa kontrolera 84
Zaawansowane widoki 95

Rozdział 5. Skalowalność warstwy prezentacji 109
Skalowalność i wąskie gardła 110
Buforowanie zawartości 111
Pule zasobów 125

Rozdział 6. Warstwa biznesowa 133
Warstwa biznesowa 135
Obiekty dziedziny 139

Rozdział 7. Komunikacja między warstwami 151
Wzorce transferu danych 151

Rozdział 8. Bazy danych i wzorce 163
Wzorce dostępu do danych 164
Wzorce klucza głównego 175
Odwzorowania obiektowo-relacyjne 180

Rozdział 9. Interfejsy warstwy biznesowej 193
Abstrakcje logiki biznesowej 194
Dostęp do usług zdalnych 204
Wyszukiwanie zasobów 214

Rozdział 10. Współbieżność biznesowa 221
Zarządzanie transakcjami 222
Ogólne wzorce współbieżności 236
Implementacja współbieżności 240

Rozdział 11. Dystrybucja komunikatów 251
Komunikaty i problematyka integracji 254
Wzorce dystrybucji komunikatów 258
Typy komunikatów 262
Korelacja komunikatów 265
Wzorce klienckie 267
Komunikaty i integracja 276
Dalsze lektury 282

Rozdział 12. Antywzorce J2EE 283
Przyczyny powstawania antywzorców 284
Antywzorce architektury 285
Antywzorce warstwy prezentacji 291
Antywzorce EJB 299

Dodatek A Wzorce warstwy prezentacji 311

Dodatek B Wzorce warstwy biznesowej 325

Dodatek C Wzorce komunikatów 353

Dodatek D Antywzorce J2EE 365

Skorowidz 371

5

Skalowalność warstwy prezentacji

Wielu projektantów uważa, że wzorce projektowe nie sprzyjają skalowalności aplikacji. Argumentują, że wzorce powodują powstawanie dodatkowych warstw w architekturze aplikacji i w związku z tym serwer musi wykonywać dodatkowe operacje oraz używać więcej pamięci podczas obsługi każdego żądania. Dodatkowe operacje powodują wydłużenie czasu odpowiedzi, a wzrost zapotrzebowania na pamięć sprawia, że serwer może równocześnie obsłużyć mniejszą liczbę klientów. Stwierdzenie takie samo w sobie jest jak najbardziej poprawne i gdyby tylko każde dwa żądania były różne, to moglibyśmy na tym zakończyć dyskusję.

Jednak w przypadku aplikacji biznesowych wielu klientów korzysta z tych samych danych. Na przykład witryna podająca informacje giełdowe może obsługiwać tysiące zapytań o wartość tej samej akcji na minutę. Jeśli cena akcji zmienia się co pięć minut, to rozwiązanie polegające na wymianie danych z systemem giełdowym podczas obsługi każdego żądania będzie zupełnie nieefektywne. Nawet w przypadku banku dostępnego przez internet, którego klienci przeglądają oczywiście wyłącznie własne konta, zasoby takie jak na przykład połączenia z bazą danych nie muszą być tworzone osobno dla każdego żądania.

Często możemy założyć pewną utratę szybkości działania aplikacji, jeśli tylko będzie ona oznaczać statystyczną poprawę wydajności. Pierwsze żądanie ceny konkretnej akcji lub pierwsze połączenie do bazy danych może wymagać wtedy skomplikowanych operacji, ale już kolejne żądania zostaną obsłużone dużo mniejszym nakładem i w efekcie szybciej. Skalowalność aplikacji wzrośnie — w tym samym przedziale czasu będzie ona mogła obsłużyć więcej żądań.

W niniejszym rozdziale omówimy trzy wzorce projektowe, które zwiększają skalowalność warstwy prezentacji, bazując na przedstawionej powyżej ogólnej koncepcji:

Wzorzec strony asynchronicznej

Pokazuje sposób buforowania danych takich jak na przykład ceny akcji i wykorzystania ich do generowanie dynamicznych stron.

Wzorzec filtra buforującego

Może być użyty do buforowania całych stron dynamicznych po ich wygenerowaniu.

Wzorzec puli zasobów

Opisuje zasadę tworzenia puli skomplikowanych obiektów lub obiektów kosztownych z innych powodów, które mogą być wynajmowane z puli zamiast tworzone za każdym razem od nowa.

Skalowalność i wąskie gardła

Zanim przejdziemy do omówienia kolejnych wzorców, zastanówmy się przez chwilę, co rozumiemy pod pojęciem *systemu skalowalnego*. Wyobraźmy sobie w tym celu system internetowy jako procesor żądań. Napływają do niego żądania klientów, którzy oczekują następnie odpowiedzi. Wszystko co ma miejsce pomiędzy odebraniem żądania a wysłaniem odpowiedzi, możemy uważać za przetwarzanie, niezależnie od tego, czy prowadzi ono do zwrócenia zawartości statycznego pliku czy też do wygenerowania w pełni dynamicznej strony.

Skalowalność procesora żądań związana jest z liczbą żądań, które mogą być przetworzone równocześnie. W najprostszym przypadku przez skalowalność można rozumieć zdolność systemu do „przetrwania" napływu określonej liczby żądań w tym samym czasie i dostarczenia w końcu odpowiedzi na każde z nich. Jednak z praktyki wiemy, że definicja taka nie jest zgodna z prawdą. Jeśli, na przykład, system udostępniający najnowsze wiadomości otrzyma równocześnie 10 000 żądań i odpowie na każde z nich w ciągu 3 sekund, to możemy powiedzieć, że system ten skaluje się poprawnie, a nawet wyjątkowo dobrze. Natomiast gdy ten sam system otrzyma 100 000 równoczesnych żądań i odpowie na każde z nich w ciągu 3 minut, to sytuacja taka będzie trudna do zaakceptowania[*].

Lepsza definicja skalowalności systemu polega na zdefiniowaniu jej jako zdolności systemu do obsługi wzrastających wymagań. Oczywiście od żadnego pojedynczego serwera nie oczekuje się w praktyce możliwości obsługi nieskończonej liczby żądań. Osiągnięcie przez serwer skalowalnego systemu granic jego możliwości powoduje, że mamy do wyboru jedną z dwóch opcji:

- zakup szybszego serwera,
- zakup kolejnych serwerów.

Chociaż może wydawać się, że szybszy serwer będzie mógł obsłużyć więcej żądań, to nie zawsze jest to prawdą. Wyobraźmy sobie na przykład bank, który przechowuje informacje o swoich globalnych aktywach w pojedynczym rekordzie. Rekord ten musi być modyfikowany za każdym razem, gdy do banku napływają kolejne kwoty lub są z niego wypłacane. Jeśli rekord ten może być równocześnie aktualizowany tylko przez jedno żądanie,

[*] Z punktu widzenia projektanta skalowalność może być powiązana ze sposobem działania systemu — wygenerowanie dynamicznej strony zajmie zwykle więcej czasu niż zwrócenie zawartości strony statycznej. Jednak użytkownicy systemu nie są świadomi takich niuansów.

to maksymalna liczba transakcji będzie ograniczona czasem zapisu tego rekordu. Zwiększenie szybkości procesora nowego serwera może przynieść pewną poprawę poprzez przyspieszenie operacji aktualizacji tego rekordu, jednak dodatkowy koszt komunikacji pomiędzy wieloma procesorami — na skutek ubiegania się wielu procesów o dostęp do pojedynczego zasobu — może sprawić, że dodanie kolejnych procesorów spowolni działanie systemu zamiast je przyspieszyć! Pojedynczy punkt, który ogranicza skalowalność całej aplikacji (taki jak rekord zawierający wartość globalnych aktywów) nazywany jest *wąskim gardłem*.

Zastosowanie wielu serwerów powoduje zwielokrotnienie liczby potencjalnych wąskich gardeł. Wyobraźmy sobie rozproszoną wersję naszego systemu bankowego posiadającą rekord globalnych aktywów umieszczony na dedykowanym serwerze. Każdy procesor żądań będzie przesyłać temu serwerowi komunikat za każdym razem, gdy do banku trafiają pieniądze lub są z niego pobierane. Oczywiście skalowalność takiego systemu będzie nadal ograniczona czasem potrzebnym do aktualizacji rekordu aktywów, ale skalowalność systemu może tym razem ograniczać również sieć łącząca serwery, a także czas potrzebny na przetłumaczenie danych na format przesyłany siecią.

W praktyce zagadnienie skalowalności zawsze wiąże się z kompromisami. Jeden z nich polega na wyborze pomiędzy kilkoma dużymi systemami a dużą liczbą małych systemów. Drugi z tych przypadków jest zwykle tańszy, ale komunikacja pomiędzy wieloma systemami może dodatkowo ograniczać jego skalowalność. Kompromis konieczny jest również pomiędzy wielkością buforów systemu a wielkością pamięci używanej w tradycyjny sposób. Wydzielenie z tej ostatniej zbyt dużych buforów spowoduje, że zmieści się w nich więcej informacji, a system będzie szybciej obsługiwał powtarzające się żądania. Równocześnie jednak zmniejszy się obszar pozostałej pamięci, wobec czego spowolniona może zostać obsługa żądań, dla których odpowiedź nie znajduje się w buforach. Sztuka tworzenia skalowalnych aplikacji polega na eliminowaniu wąskich gardeł i zachowaniu kompromisów związanych z poziomem komplikacji kodu oraz sposobami przydziału zasobów.

Buforowanie zawartości

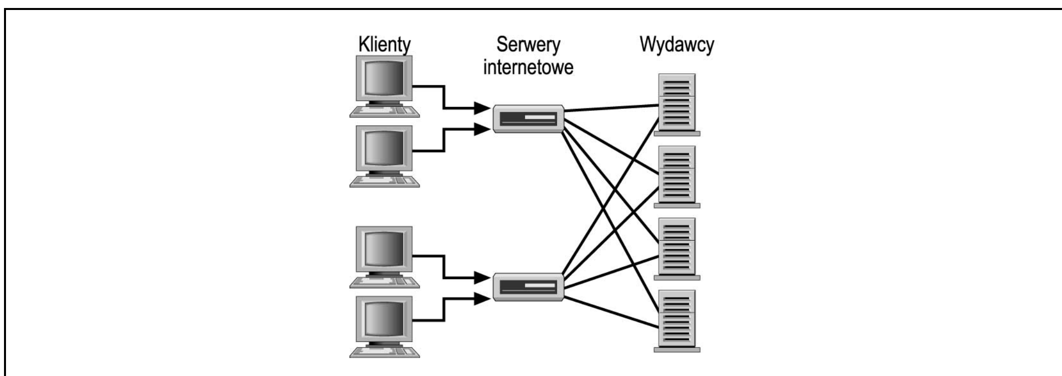
Jednym z lepszych sposobów poprawy skalowalności jest buforowanie. Buforowanie statycznych stron praktykowane jest szeroko w internecie, a dedykowane w tym celu urządzenia umieszczane są obok routerów i przełączników. Utrzymują one kopie najczęściej żądanych plików, co pozwala przyspieszyć obsługę żądań bez konieczności kontaktowania się z właściwym serwerem witryny. Większość dostępnych obecnie urządzeń buforujących ignoruje strony generowane dynamicznie, ponieważ efektywne rozróżnienie, czy dwa żądania powinny otrzymać tę samą odpowiedź (lub ustalenie, czy żądanie powinno zostać z innych powodów obsłużone przez właściwy serwer witryny) jest co najmniej kłopotliwe.

Aby zwiększyć wydajność środowiska J2EE, możemy zastosować buforowanie dynamicznych komponentów warstwy prezentacji. Przyjrzymy się teraz dwóm rozwiązaniom tego problemu: buforowaniem danych wejściowych używanych podczas dynamicznego generowania zawartości stron oraz buforowaniu samej zawartości.

Buforowanie komponentów zawartości

Tradycyjny model komunikacji używany przez aplikacje pajęczyny jest w swej naturze *synchroniczny*. Innymi słowy klienty żądają określonych adresów URL i czekają na odpowiedź w postaci strony. Biorąc pod uwagę ten model komunikacji, możemy łatwo zrozumieć, dlaczego logika większości aplikacji również implementowana jest w synchroniczny sposób. Po odebraniu żądania tworzone są i korelowane poszczególne części odpowiedzi, a następnie na ich podstawie generowana jest ostateczna odpowiedź. Rozwiązanie takie może okazać się zupełnie nieefektywne.

W rozdziale 4. przestawiliśmy metodę dodawania zdalnych źródeł wiadomości do witryny za pomocą prostego mechanizmu parsowania formatu *RSS*. Zastosowane rozwiązanie było synchroniczne — po odebraniu żądania dotyczącego strony zawierającej odnośnik do źródła wiadomości nasz własny znacznik JSP wczytywał i parsował zdalne dane, a następnie formatował je w celu prezentacji. Gdybyśmy zdecydowali się skalować takie rozwiązanie na wiele serwerów, z których każdy komunikowałby się z wieloma źródłami wiadomości, to otrzymalibyśmy sytuację podobną do przedstawionej na rysunku 5.1.

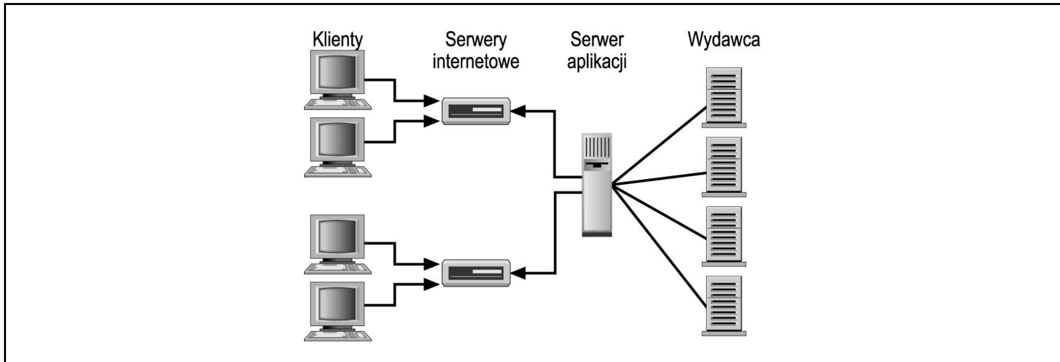


Rysunek 5.1. Odczyt wielu źródeł wiadomości

Rozwiązanie takie jest nieefektywne z wielu powodów. Kontaktowanie się z każdym źródłem dla każdego żądania powoduje niepotrzebne obciążenie sieci i samych źródeł. Sama obsługa żądań będzie wtedy kosztowna w sensie użytych zasobów, ponieważ dla każdego żądania konieczne będzie sparsowanie danych pochodzących ze źródła i ich przetłumaczenie na format *HTML*. Buforowanie danych na każdym ze serwerów powinno znacząco zwiększyć liczbę klientów, których mogą one obsłużyć.

Chociaż buforowanie ma doskonały wpływ na skalowalność systemu, to jednak nie uwzględnia ono faktu, że zawartość źródeł zmienia się okresowo. Buforowanie wymaga zastosowania odpowiedniej polityki, na przykład dane znajdujące się w buforach mogą być aktualizowane podczas obsługi co dziesiątego żądania lub co dziesięć minut. Zapewnienie absolutnej aktualności danych wymagać będzie zbyt częstej komunikacji serwerów ze źródłami i powodować duże obciążenie procesorów i sieci. Oznaczać będzie również konieczność oddzielnego parsowania i konwersji formatu danych na każdym serwerze. Lepszym rozwiązaniem byłoby przesyłanie danych tylko wtedy, gdy ulegną one zmianie.

Na rysunku 5.2 przedstawiony został ten sam system używający tym razem modelu wydawca-subskrybent. Pojedyncza maszyna, serwer aplikacji, rejestruje się w wielu źródłach wiadomości. Gdy maszyna ta otrzymuje nową wiadomość z jakiegoś źródła, parsuje ją i wysyła w odpowiednim formacie do wszystkich serwerów*. Ponieważ wydawca przesyła dane subskrybentowi tylko wtedy, gdy jest to konieczne, to komunikacja taka odbywa się *asynchronicznie*. Często rozwiązania asynchroniczne wymagają mniejszej przepustowości sieci niż rozwiązania synchroniczne, ponieważ dane przesyłane są do wielu serwerów tylko wtedy, gdy jest to konieczne.



Rysunek 5.2. Model wydawca-subskrybent

Wzorzec strony asynchronicznej

Zalety komunikacji asynchronicznej nie są nowym odkryciem. Przesyłanie komunikatów jest już od dawna jednym z ważniejszych komponentów systemów biznesowych. Interfejs programowy Java Message oraz wprowadzenie ostatnio komponentów JavaBeans sterowanych komunikatami umocniło pozycję asynchronicznej komunikacji na platformie Java. Również w internecie zaczynają się upowszechniać usługi o asynchronicznym charakterze. Z wyjątkiem jednak kilku dostawców zawartości działających według modelu wydawca-subskrybent, komunikacja asynchroniczna nie pojawiła się dotąd na warstwie klienckiej, ponieważ standardowe klienty w postaci przeglądarek internetowych nie obsługują modelu wydawca-subskrybent.

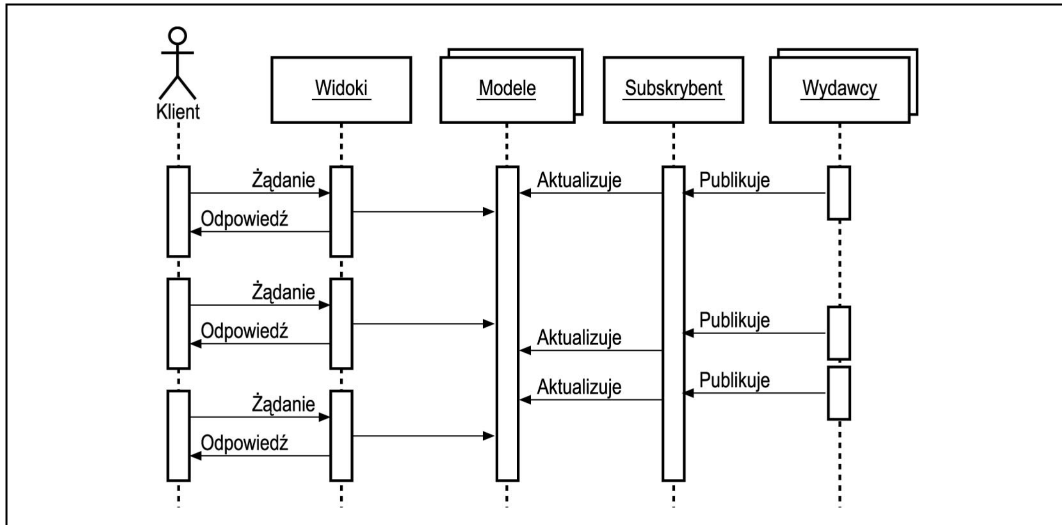
Brak obsługi modelu wydawca-subskrybent przez przeglądarki nie oznacza jednak, że komunikacja asynchroniczna w ogóle nie występuje w internetowej pajęczynie. Nadal może być ona potężnym narzędziem umożliwiającym poprawę skalowalności poprzez redukcję kosztów obsługi każdej transakcji.

Wzorzec strony asynchronicznej wykorzystuje zalety asynchronicznego dostępu do zdalnych zasobów w celu poprawienia wydajności i skalowalności aplikacji. Zamiast czekać na żądanie podania ceny wybranej akcji serwer może przechowywać w buforze wszystkie otrzymane dotąd ceny. Gdy odbierze żądanie podania ceny wybranej akcji, odpowie na nie danymi, które ma już w buforze.

---

* Model wydawca-subskrybent zostanie szczegółowo omówiony w innym kontekście w rozdziale 11.

Na rysunku 5.3 przedstawione zostały interakcje zachodzące we wzorcu strony asynchronicznej. W ogólnym przypadku występuje w nim jeden subskrybent, który zarejestrowany jest w szeregu wydawców. Gdy dane zostają zaktualizowane, to subskrybent aktualizuje model zależnych od niego aplikacji. Odpowiedzi na żądania zawierają w ten sposób zawsze ostatnie opublikowane dane.



Rysunek 5.3. Interakcje we wzorcu strony asynchronicznej

Warto zwrócić uwagę, że interfejs pomiędzy wydawcą a subskrybentem nie musi w tym przypadku ograniczać się do wysyłania powiadomień przez wydawcę, ale może również polegać na regularnym sprawdzaniu danych przez subskrybenta.

Koszt aktualizacji modelu może być różny. W niektórych przypadkach dane otrzymane od wydawcy nie wymagają przetworzenia i mogą być bezpośrednio umieszczone w modelu. Wzorzec jest jednak bardziej efektywny, gdy subskrybent przetwarza dane, redukując w ten sposób konieczność wykonywania tych operacji przez każdy model z osobna. Typowa strategia w takim przypadku polega na zastąpieniu dynamicznej strony przez stronę statyczną, która zostaje nadpisana za każdym razem, gdy pojawiają się nowe dane.

Implementacja wzorca strony asynchronicznej

Aby zilustrować sposób implementacji wzorca strony asynchronicznej, zastosujemy go do wcześniejszego przykładu związanego z parsowaniem formatu *RSS*. Przypomnijmy, że *RSS* jest standardem umożliwiającym wymianę wiadomości pomiędzy serwerami internetowymi. Wykorzystuje on język XML, a naszym zadaniem jest umieszczenie danych otrzymanych w tym formacie na stronie HTML.

W celu elastycznego parsowania formatu *RSS* stworzyliśmy dotąd jedną klasę i dwa własne znaczniki. Klasa `RSSInfo` odczytuje i parsuje dane dostępne pod podanym adresem URL. W oparciu o tę klasę utworzyliśmy dwa znaczniki. Parametrem pierwszego z nich,

RSSChannelTag, jest adres URL i wczytuje on zdalne dane. Znacznik RSSChannelTag przechowuje nazwę kanału i jego łącza za pomocą dwóch zmiennych skryptowych. Znacznik RSSItemsTag może być zagnieżdżany w dowolnym znaczniku RSSChannelTag. Znacznik RSSItemsTag przegląda wszystkie elementy kanału i umieszcza jego tytuł i łącze w zmiennych skryptowych.

Problemem, który będziemy chcieli teraz rozwiązać, jest to, że znacznik RSSChannelTag odczytuje dane ze zdalnego źródła. Lepiej byłoby, gdyby dane te były przechowywane lokalnie i aktualizowane tylko wtedy, gdy jest to konieczne. Niestety, RSS nie oferuje mechanizmu subskrypcji, wobec czego będziemy zmuszeni okresowo sprawdzać dane zdalnego źródła. Zamiast wykonywać tę operację podczas każdego odczytu, stworzymy raczej dedykowany mechanizm, który będzie się zajmował kontrolowaniem danych źródła i aktualizacją ich lokalnej kopii. Dedykowany mechanizm pozwoli nam odczytywać dane z jednego, centralnego źródła, a następnie dystrybuować je (jeśli uległy zmianie) do właściwych procesorów żądań.

W naszym przykładzie dodamy jeszcze jedną klasę, RSSSubscriber. Klasa ta umożliwiać będzie dodawanie subskrypcji różnych źródeł RSS przez podanie ich adresu URL. Po dodaniu subskrypcji osobny wątek będzie sprawdzać dane źródła w określonych odstępach czasu. Jeśli dane te ulegną zmianie, zostaną dodane do lokalnego bufora. Wszystkie żądania z wyjątkiem pierwszego będą obsługiwane na podstawie zawartości tego bufora. Implementacja klasy RSSSubscriber została przedstawiona na listingu 5.1.

Listing 5.1. Klasa RSSSubscriber

```
import java.util.*;
import java.io.IOException;

public class RSSSubscriber extends Thread {
    private static final int UPDATE_FREQ = 30 * 1000;

    // wewnętrzna reprezentacja subskrypcji
    class RSSSubscription implements Comparable {
        private String url;
        private long nextUpdate;
        private long updateFreq;

        // sortowanie subskrypcji w oparciu o czas następnej aktualizacji
        public int compareTo(Object obj) {
            RSSSubscription rObj = (RSSSubscription) obj;
            if (rObj.nextUpdate > this.nextUpdate) {
                return -1;
            } else if (rObj.nextUpdate < this.nextUpdate) {
                return 1;
            } else {
                // jeśli czas aktualizacji jest taki sam,
                // porównuje adresy URL
                return url.compareToIgnoreCase(rObj.url);
            }
        }
    }

    // zbiór subskrypcji posortowany na podstawie czasu kolejnej aktualizacji
    private SortedSet subscriptions;
```

```
private Map cache;
private boolean quit = false;

// singleton subskrybenta
private static RSSSubscriber subscriber;

// zwraca referencję subskrybenta
public static RSSSubscriber getInstance() {
    if (subscriber == null) {
        subscriber = new RSSSubscriber();
        subscriber.start();
    }
    return subscriber;
}

RSSSubscriber() {
    subscriptions = new TreeSet();
    cache = Collections.synchronizedMap(new HashMap());
    setDaemon(true);
}

// zwraca obiekt RSSInfo z bufora lub tworzy nową subskrypcję
public RSSInfo getInfo(String url) throws Exception {
    if (cache.containsKey(url)) {
        return (RSSInfo)cache.get(url);
    }

    // dodaje do bufora
    RSSInfo rInfo = new RSSInfo();
    rInfo.parse(url);
    cache.put(url, rInfo);

    // tworzy nową subskrypcję
    RSSSubscription newSub = new RSSSubscription();
    newSub.url = url;
    newSub.updateFreq = UPDATE_FREQ;
    putSubscription(newSub);

    return rInfo;
}

// dodaje subskrypcję
private synchronized void putSubscription(RSSSubscription subs) {
    subs.nextUpdate = System.currentTimeMillis() + subs.updateFreq;
    subscriptions.add(subs);
    notify();
}

// oczekuje, aż kolejna subskrypcja będzie gotowa do aktualizacji
private synchronized RSSSubscription getSubscription() {
    while(true) {
        while(subscriptions.size() == 0) {
            try { wait(); } catch(InterruptedException ie) {}
        }

        //  pobiera pierwszą subskrypcję z kolejki
        RSSSubscription nextSub = (RSSSubscription)subscriptions.first();

        // sprawdza, czy nadszedł czas aktualizacji
        long curTime = System.currentTimeMillis();
        if(curTime >= nextSub.nextUpdate) {
            subscriptions.remove(nextSub);
```

```
                return nextSub;
            }

            // zawiesza wykonanie do czasu kolejnej aktualizacji
            // oczekiwanie to zostanie przerwane na skutek dodania subskrypcji
            try {
                wait(nextSub.nextUpdate - curTime);
            } catch(InterruptedException ie) {}
        }
    }

    // aktualizuje subskrypcję
    public void run() {
        while(!quit) {
            RSSSubscription subs = getSubscription();

            try {
                RSSInfo rInfo = new RSSInfo();
                rInfo.parse(subs.url);
                cache.put(subs.url, rInfo);
            } catch(Exception ex) {
                ex.printStackTrace();
            }

            putSubscription(subs);
        }
    }

    public void quit() {
        quit = true;
        notify();
    }
}
```

Nasz mechanizm subskrypcji danych RSS działa w jednym serwerze, ale łatwo można sobie wyobrazić jego rozszerzenie na wiele serwerów. W obu przypadkach umożliwia on obsługę większej liczby równoczesnych żądań poprzez istnienie dedykowanego wątku zajmującego się aktualizacją i parsowaniem żądań. Z wyjątkiem początkowego żądania dla danej subskrypcji, żaden wątek nie musi odczytywać i parsować zdalnych danych. Zadanie to jest efektywnie wykonywane w tle.

Aby skorzystać z klasy `RSSSubscriber`, nasz własny znacznik wywołuje po prostu metodę `getRSSInfo()` dla przekazanego mu adresu URL. Metoda `getRSSInfo()` odczytuje dane z bufora lub tworzy nową subskrypcję, gdy danych tych nie ma w buforze. Na listingu 5.2 przedstawiony został zmodyfikowany kod znacznika.

Listing 5.2. Znacznik RSSChannelTag

```
import javax.servlet. *;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class RSSChannelTag extends BodyTagSupport {
    private static final String NAME_ATTR = "channelName";
    private static final String LINK_ATTR = "channelLink";

    private String url;
```

```
    private RSSSubscriber rssSubs;

    public RSSChannelTag() {
        resSubs = RSSSubscriber.getInstance();
    }

    public void setURL(String url) {
        this.url = url;
    }

    // zwraca najnowszy obiekt RSSInfo od subskrybenta;
    // wywoływana również przez znaczniki RSSItemsTag
    protected RSSInfo getRSSInfo() {
        try {
             return rssSubs.getInfo(url);
        } catch (Exception ex) {
            ex.printStackTrace();
        return null;
    }


    // używa zaktualizowanego obiektu RSSInfo
    public int doStartTag() throws JspException {
        try {
            RSSInfo rssInfo = getRSSInfo();
            pageContext.setAttribute(NAME_ATTR, rssinfo.getChannelTitle());
            pageContext.setAttribute(LINK_ATTR, rssinfo.getChannelLink());
            RSSSubscriber rssSubs = RSSSubscriber.getInstance();
         } catch (Exception ex) {
             throw new JspException("Nie można sparsować " + url, ex);
        }
        return Tag.EVAL_BODY_INCLUDE;
    }
}
```

Chociaż przykład odczytu danych w formacie *RSS* jest prosty, to ilustruje jedną z wielu okazji zastosowania komunikacji asynchronicznej w środowisku, którego działanie opiera się na przetwarzaniu żądań. Zastosowanie to zależy oczywiście od sposobu dostępu do danych. Wyobraźmy sobie na przykład odebranie, parsowanie i przechowywanie informacji o wszystkich akcjach notowanych na nowojorskiej giełdzie, po których to operacjach pojawiają się tylko dwa lub trzy żądania, zanim wszystkie informacje muszą być zaktualizowane ponownie. Czas procesora oraz pamięć poświęcone na asynchroniczne odebranie tych wartości zostaną więc zmarnowane. Oczywiście istnieją aplikacje, które nie mogą sobie pozwolić, by jakiekolwiek dane nie były aktualne niezależnie od sytuacji — na przykład gdy korzystamy z bankomatu. Oceniając przydatność komunikacji asynchronicznej w konkretnym przypadku musimy rozważyć jej koszty w kategoriach aktualności danych, zużycia pasma w sieci i zużycia pamięci w stosunku do zalet wynikających z poprawy wydajności i skalowalności.

Buforowanie zawartości dynamicznej

Istnieje jeszcze inna klasa dynamicznych danych, które nadają się do buforowania. Wyobraźmy sobie na przykład witrynę sprzedaży samochodów, której użytkownicy przeglądają kilka kolejnych stron, wybierając różne warianty wyposażenia, a na końcu otrzymują

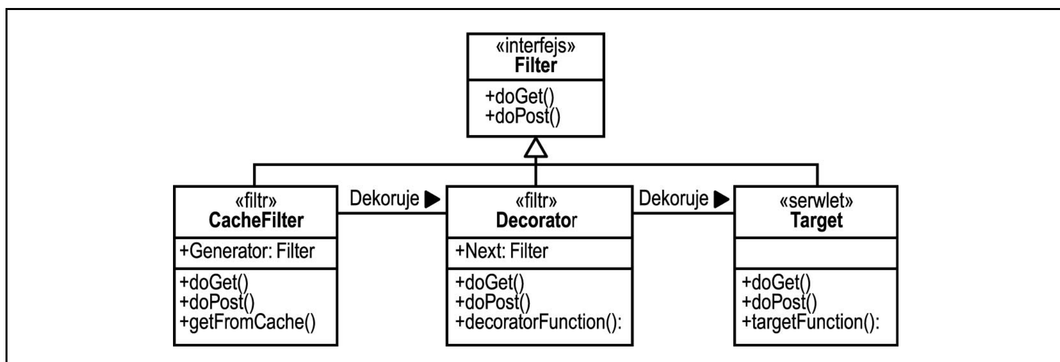
informację o cenie tak skonfigurowanej wersji pojazdu. Wyznaczenie tej ceny może być skomplikowanym procesem korzystającym z danych zewnętrznego systemu, w tym także zawierającego dane o aktualnie dostępnych egzemplarzach pojazdów.

Niektóre wersje samochodów oraz opcje wyposażenia — na przykład wersja sportowa czy otwierany dach — mogą być częściej wybierane niż inne. Ponieważ ten sam zestaw opcji wyposażenia daje w efekcie zawsze tę samą cenę, to obliczanie jej za każdym razem nie jest efektywnym rozwiązaniem. Jeszcze bardziej kosztowne z punktu widzenia wydajności systemu może być dynamiczne generowanie strony zawierającej cenę samochodu. Będzie ono wymagać dostępu do bazy danych oraz skonstruowania złożonego widoku. Dlatego też warto zastanowić się nad możliwością buforowania strony HTML zawierającej już gotową cenę.

Teoretycznie aplikacja nie musi zajmować się buforowaniem takich stron. Specyfikacja HTTP 1.1 umożliwia buforowanie dynamicznych żądań `GET`, jeśli tylko wyposażymy je w odpowiednie nagłówki HTTP*. Po nadaniu im odpowiednich wartości buforowanie może być wykonywane przez klienta, bufor pośredniczący lub nawet sam serwer HTTP. W praktyce jednak okazuje się, że tworząc aplikację, sami musimy zająć się problemem buforowania.

Wzorzec filtra buforującego

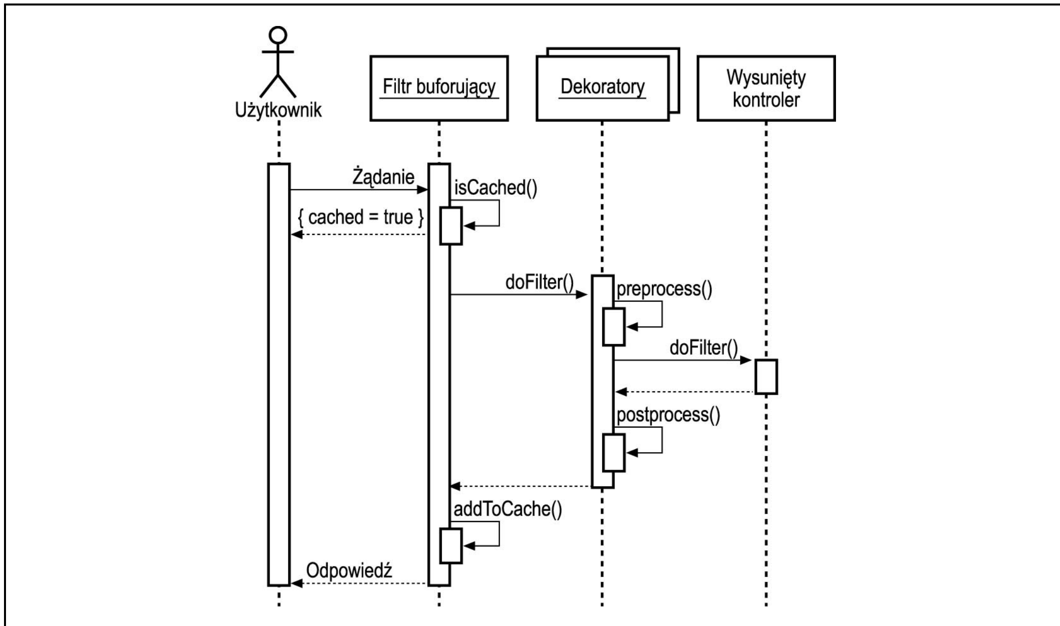
Wzorzec filtra buforującego wykorzystuje filtr serwletu do buforowania dynamicznie generowanych stron. Możliwości i sposób implementacji filtrów omówiliśmy już, przedstawiając wzorzec dekoratora w rozdziale 3. Filtr buforujący stanowi specyficzną implementację dekoratora. Zastosowany do wysuniętego kontrolera umożliwia pełne buforowanie stron generowanych dynamicznie. Klasy biorące udział we wzorcu filtra buforującego przedstawione zostały na rysunku 5.4.



Rysunek 5.4. Klasy wzorca filtra buforującego

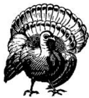
* Więcej informacji na temat różnych możliwości buforowania udostępnianych przez HTTP 1.1 można znaleźć w punkcie 13. dokumentu RFC 2616 pod adresem *http://www.w3.org/Protocols/rfc2616/rfc2616-sec13.html#sec13*.

Klasa `CacheFilter` reprezentująca filtr buforujący działa jak każdy inny filtr — udostępnia taki sam interfejs jak dekorowany obiekt oraz dodatkowe metody umożliwiające odczyt stron z bufora oraz umieszczenie w buforze wyników obsługi konkretnego żądania. Po nadejściu żądania zwracana jest strona z bufora, jeśli istnieje. Jeśli żądanej strony nie ma w buforze, to wykonana musi zostać pozostała cześć łańcucha, a jej wynik zostaje umieszczony w buforze. Proces obsługi żądania przedstawiony został na rysunku 5.5.



Rysunek 5.5. Interakcje we wzorcu filtra buforującego

Zwróćmy uwagę na miejsce filtra buforującego w łańcuchu filtrów. W zasadzie filtr buforujący może być umieszczony w dowolnym miejscu łańcucha i buforować wyniki działania wszystkich filtrów znajdujących się w dalszej części łańcucha. Możliwe jest nawet istnienie wielu buforów na różnych poziomach łańcucha, na przykład w celu buforowania części tworzonej strony, podczas gdy jej reszta nadal generowana jest dynamicznie.



Częsty błąd związany ze stosowaniem buforowania polega na umieszczeniu buforów przed mechanizmami bezpieczeństwa. Prowadzi to do nieautoryzowanego dostępu do buforowanych danych. Jeśli mechanizm bezpieczeństwa implementowany jest za pomocą filtra, to musi on być umieszczony w łańcuchu przed filtrami buforującymi.

Implementacja filtra buforującego

Aby buforować dane, musimy zmienić sposób ich dostarczania serwerowi. W wielu przypadkach klient żąda po prostu kolejnej strony, nie przekazując wszystkich parametrów. Generując odpowiedź, kontroler musi użyć kombinacji żądanej strony i parametrów przechowanych dla danej sesji. Żądanie wysłane przez klienta może mieć następującą postać:

```
GET /pages/finalPrice.jsp HTTP/1.1
```

Do żądania w tej postaci serwer dodałby następnie informację o wybranych opcjach. Niestety, żądanie `GET` wygląda zawsze tak samo, niezależnie od wybranych przez użytkownika opcji. To, że klient wybrał błękitny lakier i aluminiowe felgi nie znajduje wcale odbicia w postaci żądania. Aby odnaleźć odpowiednią stronę, konieczne jest użycie zmiennych sesji, co wiąże się z pewną utratą efektywności, ponieważ informacja ta nie jest buforowana.

Lepiej byłoby, gdyby adres URL zawierał wszystkie parametry w łańcuchu zapytania. Na przykład w taki sposób:

```
GET /pages/finalPrice.jsp?paint=Electric+Blue&Wheels=Alloy
```

Najefektywniej możemy zaimplementować buforowanie, jeśli łańcuch zapytania w pełni opisuje stronę (patrz „GET, POST i idempotencja")

Implementując filtr buforujący, wykorzystamy interfejs filtra serwletów. Podobnie jak w rozdziale 3. udekorujemy obiekt odpowiedzi przekazywany w dół łańcucha filtrów obiektem zawierającym wynik przetwarzania przez resztę łańcucha. Obiekt ten będzie należeć do klasy `CacheResponseWrapper` przedstawionej na listingu 5.3.

Listing 5.3. Klasa CacheResponseWrapper

```
public class CacheResponseWrapper extends HttpServletResponseWrapper {
    // zastępczy strumień wyjścia
    private CacheOutputStream replaceStream;
    // zastępczy obiekt zapisu
    private PrintWriter replaceWriter;

    // prosta implementacja klasy pochodnej klasy ServletOutputStream,
    // która przechowuje dane zapisane do strumienia
    class CacheOutputStream extends ServletOutputStream {
        private ByteArrayOutputStream bos;

        CacheOutputStream() {
            bos = new ByteArrayOutputStream();
        }

        public void write(int param) throws IOException {
            bos.write(param);
        }


        // zwraca zapisane dane
        protected byte[] getBytes() {
            return bos.toByteArray();
        }
    }

    public CacheResponseWrapper(HttpServletResponse original) {
        super(original);
    }

    public ServletOutputStream getOutputStream()
        throws IOException
    {
        if (replaceStream != null) {
```

GET, POST i idempotencja

Specyfikacja HTTP 1.1 stwierdza, że żądania `GET` są *idempotentne*. Termin ten pochodzi z łaciny i oznacza tutaj możliwość powtarzania tej samej operacji dowolną liczbę razy bez niepożądanych skutków. W naszym przykładzie sprzedaży samochodów idempotenetne jest na przykład żądanie podanie ceny samochodu — możemy wielokrotnie żądać podania ceny wybranej konfiguracji i nie ma to żadnego wpływu na podaną wartość.

Jednak sam zakup samochodu nie jest wcale idempotentny. Dlatego właśnie protokół HTTP dostarcza metody `POST` umożliwiającej wysłanie informacji do serwera. Operacja `POST` nie jest idempotentna i dlatego często wybierając przycisk przeglądarki służący ponownemu załadowaniu tej samej strony, otrzymujemy komunikat, że zawartość formularza musi zostać ponownie wysłana do serwera. Zaletą żądań `POST` jest też ukrycie przesyłanej informacji przed użytkownikiem(parametry żądania nie pojawiają się w adresie wyświetlanym przez przeglądarkę) oraz przed osobami posiadającymi dostęp do dzienników zapisywanych przez serwer. Na przykład informacje o kartach kredytowych powinny być zawsze przesyłane za pomocą żądań `POST`, nawet wtedy, gdy równocześnie wykorzystywany jest protokół SSL.

Ze względu na idempotencję, implementując buforowanie dynamicznej zawartości stron musimy ograniczyć się do zawartości generowanej za pomocą żądań `GET`. Nawet w ich przypadku należy zawsze się zastanawiać, czy buforowanie będzie bezpieczne.

```
            return replaceStream;
        }

        // sprawdza, czy obiekt zapisu nie został już zainicjowany
        if (replaceWriter != null) {
            throw new IOException("Obiekt zapisu jest już używany");
        }

        replaceStream = new CacheOutputStream();

        return replaceStream;
    }

    public PrintWriter getWriter() throws IOException {
        if (replaceWriter != null) {
            return replaceWriter;
        }

        // sprawdza, czy strumień wyjściowy nie został już zainicjowany
        if (replacStream != null) {
            throw new IOException("Strumień wyjściowy jest już używany");
        }

        replaceWriter = new PrintWriter(
             new OutputStreamWriter(new CacheOutputStream()));

        return replaceWriter;
    }
```

```
    // zwraca przechowane dane
    protected byte[] getBytes() {
        if (replaceStream == null)
            return null;
        return outStream.getBytes();
    }
}
```

Przekazując obiekt `CacheResponseWrapper` do następnego filtra w łańcuchu, możemy umieścić dane w tablicy bajtowej, która następnie zostanie zbuforowana w pamięci lub na dysku.

Działanie samego filtra buforującego jest dość proste. Po odebraniu żądania najpierw sprawdza on, czy strona może być buforowana. Jeśli tak, to filtr kontroluje, czy strona ta znajduje się już w buforze i zwraca wersję z bufora. W przeciwnym razie tworzy nową stronę i umieszcza ją w buforze. Kod filtra buforującego przedstawiony został na listingu 5.4.

Listing 5.4. Klasa CacheFilter

```
public class CacheFilter implements Filter {
    private FilterConfig filterConfig;

    // bufor danych
    private HashMap cache;

    public void doFilter(ServletRequest request,
                         ServletResponse response,
                         FilterChain chain)
        throws IOException, ServletException
    {
        HttpServletRequest req = (HttpServletRequest) request;
        HttpServletResponse res = (HttpServletResponse) response;

        // klucz tworzony jest następująco: URI + łańcuch zapytania
        String key = req.getRequestURI() + "?" + req.getQueryString();

        // buforuje jedynie te żądania GET, które zawierają dane
        // nadające się do buforowania
        if (req.getMethod().equalsIgnoreCase("get") && isCacheable(key)) {
            // próbuje pobrać dane z bufora
            byte[] data = (byte[]) cache.get(key);

            // jeśli danych nie ma w buforze, to generuje wynik w zwykły sposób
            // i dodaje go do bufora
            if (data == null) {
                CacheResponseWrapper crw = new CacheResponseWrapper(res);
                chain.doFilter(request, crw);
                data = crw.getBytes();
                cache.put(key, data);
            }

            // jeśli dane zostały odnalezione lub dodane do bufora,
            // to używa ich do wygenerowania wyniku
            if (data != null) {
                res.setContentType("text/html");
                res.setContentLength(data.length);
```

```
                try {
                    OutputStream os = res.getOutputStream();
                    os.write(data);
                    os.flush();
                    os.close();
                } catch(Exception ex) {
                 ...
                }
            }
        }
        // generuje dane w zwykły sposób
        // w pozostałych przypadkach
        chain.doFilter(request, response);
    }

    // sprawdza, czy dane nadają się do buforowania
    private boolean isCacheable(String key) {
        ...
    }

    // inicjuje bufor
    public void init(FilterConfig filterConfig) {
        this.filterConfig = filterConfig;

        cache = new HashMap();
    }
}
```

Zauważmy, że nasz bufor nie jest zmienną statyczną. Zgodnie ze specyfikacją filtrów dla każdego elementu `filter` umieszczonego w deskryptorze instalacji zostaje utworzona tylko jedna instancja filtra. W każdym z filtrów możemy więc używać osobnego bufora, umożliwiając tym samym istnienie wielu buforów w różnych punktach łańcucha filtrów.

Nasz prosty filtr pomija dwa trudne aspekty buforowania. Pierwszy polega na ustaleniu, czy strona może być buforowana. W większości środowisk występują strony nadające się do buforowania, jak i strony, które nie mogą być buforowane. W naszym przykładzie sprzedaży samochodów do buforowania nadają się strony opisujące różne konfiguracje aut, ale z pewnością nie może być buforowana strona zawierająca informacje o karcie kredytowej klienta. Typowe rozwiązanie tego problemu polega na stworzeniu odwzorowania w języku XML opisującego strony, które mogą być buforowane (podobnego do przedstawionych wcześniej odwzorowań serwletów bądź filtrów).

Druga trudność polega na zapewnieniu aktualności bufora. W naszym przykładzie założyliśmy, że generowane dynamicznie strony nie zmieniają swej zawartości. Jeśli cena pewnego elementu wyposażenia ulegnie zmianie, to klientowi nadal prezentowana będzie strona z bufora zawierająca poprzednią, nieaktualną już cenę. Istnieje wiele strategii zachowania aktualności stron w buforze zależnych od natury generowanych stron. Najprostsze rozwiązanie polega oczywiście na przeterminowaniu zawartości bufora co pewien określony okres czasu. W przeciwnym razie nie tylko mogą się pojawić w buforze nieaktualne dane, ale jego zawartość może wzrastać w nieograniczony sposób, prowadząc do sytuacji opisanej w rozdziale 12. podczas omawiania antywzorca wycieku z kolekcji.

Pule zasobów

Przez *pulę zasobów* rozumiemy kolekcję utworzonych wcześniej obiektów, które mogą być wielokrotnie wynajmowane z puli, co pozwala uniknąć kosztów tworzenia tych obiektów za każdym razem od nowa. W środowisku J2EE pule zasobów są wszechobecne. Na przykład, gdy pojawia się nowe połączenie, to do obsługi żądania pobierany jest wątek z puli wątków. Jeśli przetwarzanie wymaga komponentu EJB, to może on zostać przydzielony z puli komponentów EJB, a jeśli komponent taki wymaga dostępu do bazy danych, to uzyska połączenie z bazą — niespodzianka! — z puli połączeń.

Pule są tak często wykorzystywane, ponieważ rozwiązują jednocześnie dwa problemy: poprawiają skalowalność poprzez podział kosztów tworzenia skomplikowanych obiektów na wiele instancji, a także umożliwiają precyzyjne strojenie równoległości i wykorzystania pamięci.

Zastosowanie pul zilustrujemy klasycznym przykładem połączeń do bazy danych. Proces łączenia się z bazą danych, zwłaszcza zdalną, może być skomplikowany i kosztowny zarazem. Chociaż połączenie takie odbywać się może poprzez wywołanie pojedynczej metody, to jednak utworzenie instancji połączenia może wymagać niektórych albo nawet wszystkich z wymienionych poniżej operacji:

1. Utworzenie obiektu reprezentującego połączenie z bazą danych.
2. Przekazanie obiektowi adresu bazy danych oraz informacji uwierzytelniającej użytkownika.
3. Utworzenie połączenia sieciowego z bazą danych.
4. Wykonanie skomplikowanej negocjacji obsługiwanych opcji.
5. Wysłanie informacji uwierzytelniającej użytkownika do bazy danych.
6. Weryfikacja uprawnień użytkownika.

Po wykonaniu tych operacji połączenie jest gotowe do użytku. Wykonanie tych operacji jest nie tylko czasochłonne. Każdy obiekt połączenia musi przechowywać wiele różnych opcji i wobec tego wymaga pewnego obszaru pamięci. Jeśli połączenia nie są współdzielone, to ich liczba i koszt wzrastają wraz z liczbą żądań. Koszt utrzymywania wielu połączeń zaczyna w pewnym momencie ograniczać liczbę klientów, których może obsłużyć aplikacja.

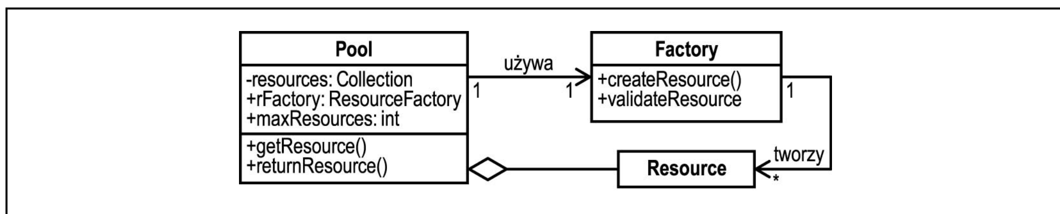
Współdzielenie połączeń jest więc obowiązkowe z punktu widzenia skalowalności. Podobnie współdzielenie kosztów tworzenia połączeń. W skrajnym przypadku można tak zaprojektować całą aplikację, by obsługiwała wszystkich klientów za pomocą pojedynczego połączenia z bazą danych. Rozwiązanie takie skutecznie eliminuje koszt tworzenia połączeń z bazą danych, jednak ogranicza równoległość, ponieważ dostęp do bazy danych musi być jakoś zsynchronizowany. Współdzielenie połączeń z bazą danych pozwala także komponentom aplikacji uniknąć uczestniczenia w transakcjach (patrz rozdział 10.).

Lepsze rozwiązanie polega na utworzeniu puli zawierającej pewną liczbą połączeń współdzielonych przez klientów. Gdy klient wymaga połączenia z bazą danych, pobiera je z puli. Po zakończeniu komunikacji z bazą danych klient zwraca połączenie do puli i może ono zostać użyte przez kolejnego klienta. Ponieważ połączenia są w ten sposób współdzielone przez klientów, to koszty utworzenia i utrzymania połączeń rozkładają się na wielu klientów. Liczba połączeń dostępnych w puli musi mieć górną granicę, aby koszty ich tworzenia i utrzymywania nie wymknęły się spod kontroli.

Kolejną istotną zaletą puli jest to, że tworzy ona pojedynczy punkt efektywnego strojenia. Jeśli w puli umieścimy więcej obiektów, to zwiększymy czas jej tworzenia oraz zużycie pamięci, ale zwykle uzyskamy również możliwość obsługi większej liczby klientów. Natomiast mniejszy rozmiar puli sprzyja skalowalności systemu, ponieważ jej obsługa wymaga mniej pamięci i czasu procesora. Zmieniając parametry puli podczas działania aplikacji, możemy dopasować stopień wykorzystania pamięci i procesora do możliwości systemu, w którym działa aplikacja.

Wzorzec puli zasobów

Wzorzec puli zasobów może być zastosowany do wielu kosztownych operacji. Pule zasobów ujawniają najwięcej zalet w przypadku takich zasobów, których tworzenie związane jest ze znacznymi kosztami, takich jak połączenia z bazą danych lub wątki. Zastosowanie w tych przypadkach wzorca powoduje rozłożenie kosztów na wiele klientów. Pule mogą być również zaadaptowane do operacji — takich jak parsowanie — których wykonanie jest czasochłonne. Pozwalają one wtedy kontrolować sposób wykorzystania pamięci i czasu procesora. Na rysunku 5.6 przedstawiony został ogólny diagram wzorca puli zasobów.



Rysunek 5.6. Wzorzec puli zasobów

Obiekt `Pool` reprezentuje pulę zasobów i jest odpowiedzialny za tworzenie, utrzymanie i kontrolę dostępu do obiektów `Resource` reprezentujących zasoby. Klient wywołuje metodę `getResource()` w celu uzyskania instancji zasobu. Gdy zasób nie jest mu już potrzebny, zwraca go do puli za pomocą metody `returnResource()`.

Pula wykorzystuje obiekt fabryki `Factory` do tworzenia instancji zasobu. Korzystając z fabryki, ta sama pula może być wykorzystywana dla wielu różnych rodzajów obiektów. Metoda fabryki `createResource()` używana jest w celu wygenerowania nowych instancji zasobu. Zanim instancja zasobu zostanie przekazana kolejnemu klientowi do ponownego użycia, pula wywołuje metodę fabryki `validateResource()` nadającą zasobowi określony stan początkowy. Jeśli zasób jest, na przykład, połączeniem z bazą danych, które zostało

już zamknięte, to metoda `validateResource()` może po prostu zwrócić wartość false, wskazując na konieczność dodania nowego połączenia do puli. Dla zwiększenia efektywności fabryka może nawet próbować „naprawić" zasób — na przykład próbując ponownie otworzyć połączenie z bazą danych. Dlatego też metodę `validateResource()` nazywa się czasami *metodą odzysku*.

Wzorzec nie nakłada żadnych ograniczeń na klasę `Resource`. Zawartość i wykorzystanie zasobu muszą być jedynie skoordynowane pomiędzy twórcą puli a jej klientami. Zwykle pule przechowują obiekty jednego typu, ale nie jest to wymagane. Zaawansowane implementacje puli czasami pozwalają nawet na przekazanie metodzie `getResource()` filtra w celu określenia kryteriów, które powinien spełniać zasób.

Implementacja puli zasobów

Silnik serwletów wykorzystuje pulę wątków do obsługi żądań. Każde żądanie obsługiwane jest przez pojedynczy wątek pochodzący z tej puli, który uzyskuje dostęp do współdzielonych instancji serwletów w celu wygenerowania wyniku. Większość serwerów aplikacji umożliwia konfigurowanie liczby wątków dostępnych w puli w czasie działania serwera. Liczba ta stanowi krytyczną zmienną dla skalowalności aplikacji internetowej — jeśli będzie zbyt mała, to pewne żądania niektórych klientów zostaną odrzucone lub będą obsługiwane zbyt długo. Jeśli pula będzie zbyt duża, może przeciążyć serwer i w efekcie spowolnić działanie aplikacji.

Istnienie puli wątków nie wyczerpuje jednak zastosowań puli w aplikacjach biznesowych. Pula wątków serwletów stanowi dość zgrubne rozwiązanie. Korzystając z tej jednej puli, zakładamy tym samym, że te same ograniczenia dotyczą wszystkich żądań, na przykład szybkości parsowania plików *XML* lub połączeń z bazą danych. W praktyce jednak różne żądania mają różne ograniczenia. Osobne pule dla parserów XML i połączeń z bazami danych mogą umożliwić zwiększenie całkowitej liczby wątków i wprowadzenie różnych ograniczeń na czas parsowania i czas połączenia z bazą danych.

Opis wzorca sugeruje, że implementacja puli zasobów w języku Java jest całkiem prosta. Najlepiej jest stworzyć dostatecznie ogólną pulę, aby następnie tworzyć pule dowolnych obiektów, zmieniając tylko ich fabrykę. Ze względu na możliwość korzystania z puli równocześnie przez wiele wątków (każda pula używana w środowisku serwletów musi obsługiwać taką możliwość) musimy zapewnić synchronizację dostępu do puli. Na listingu 5.5 przedstawiona została implementacja prostej puli.

Listing 5.5. Klasa ResourcePool

```
import java.util.*;

public class ResourcePool {
    private ResourceFactory factory;
    private int maxObjects;
    private int curObjects;
    private boolean quit;

    // zasoby wynajęte
```

```
    private Set outResources;

    // zasoby do wynajęcia
    private List inResources;

    public ResourcePool(ResourceFactory factory, int maxObjects) {
        this.factory = factory;
        this.maxObjects = maxObjects;

        curObjects = 0;

        outResources = new HashSet(maxObjects);
        inResources = new LinkedList();
    }

    // pobiera zasób z puli
    public synchronized Object getResource() throws Exception {
        while(!quit) {

            // najpierw próbuje zwrócić istniejący zasób
            if (!inResources.isEmpty()) {
                Object o = inResources.remove(0);

                // jeśli nie jest poprawny, to tworzy kolejny
                if(!factory.validateResource(o))
                    o = factory.createResource();

                outResources.add(o);
                return o;
            }

            // następnie tworzy nowy zasób,
            // jeśli limit nie został jeszcze osiągnięty
            if(curObjects < maxObjects) {
                Object o = factory.createResource();
                outResources.add(o);
                curObjects++;

                return o;
            }

            // jeśli brak jest dostępnych zasobów,
            // to oczekuje, aż jakiś zostanie zwrócony
            try { wait(); } catch(Exception ex) {}
        }

        // pula została usunięta
        return null;
    }

    // zwraca zasób do puli
    public synchronized void returnResource(Object o) {

        // coś jest nie tak, metoda poddaje się
        if(!outResources.remove(o))
            return;

        inResources.add(o);
        notify();
    }

    public synchronized void destroy() {
```

```
            quit = true;
            notifyAll();
        }
    }
```

W powyższym przykładzie założyliśmy, że fabryki mają bardzo prosty interfejs naszkicowany już wcześniej:

```
public interface ResourceFactory {
    public Object createResource();
    public boolean validateResource(Object o);
}
```

Zastosowanie puli zilustrujemy na przykładzie operacji parsowania plików *XML*. Podobnie jak w przypadku połączeń do baz danych, również tworzenie i utrzymywanie parserów XML może być kosztowne. Stosując pulę parserów, nie tylko rozkładamy koszt ich tworzenia na wiele klientów, ale zyskujemy równocześnie możliwość kontroli, jak wiele wątków wykonuje kosztowną operację parsowania w danym momencie. Aby stworzyć pulę parserów, musimy jedynie dostarczyć odpowiednią fabrykę, na przykład taką jak pokazana na listingu 5.6.

Listing 5.6. Klasa XMLParserFactory

```
import javax.xml.parsers.*;

public class XMLParserFactory implements ResourceFactory {
    DocumentBuilderFactory dbf;

    public XMLParserFactory() {
        dbf = DocumentBuilderFactory.newInstance();
    }

    // tworzy nowy obiekt DocumentBuilder w celu dodania do puli
    public Object createResource() {
        try {
            return dbf.newDocumentBuilder();
        } catch (ParserConfigurationException pce) {
            ...
            return null;
        }
    }

    // sprawdza, czy obiekt DocumentBuilder jest poprawny
    // i nadaje mu domyślne wartości atrybutów
    public boolean validateResource(Object o) {
        if (!(o instanceof DocumentBuilder)) {
            return false;
        }

        DocumentBuilder db = (DocumentBuilder) o;
        db.setEntityResolver(null);
        db.setErrorHandler(null);

        return true;
    }
}
```

Kod klienta używającego puli parserów XML może wyglądać następująco:

```
public class XMLClient implements Runnable {
    private ResourcePool pool;

    public XMLClient(int poolsize) {
        pool = new ResourcePool(new XMLParserFactory(), poolsize);
        ...
        // uruchamia wątki itd.
        Thread t = new Therad(this);
        t.start();
        ...
        // czeka na wątki
        t.join();
        // usuwa pulę
        pool.destroy();
    }

    public void run() {
        try {
            // pobiera parser z puli
            DocumentBuilder db = (DocumentBuilder)pool.getResource();
        } catch (Exception ex) {
            return;
        }

        try {
            ...
            // wykonuje parsowanie
            ...
        } catch (Exception ex) {
            ...
        } finally {
            // zawsze zwraca zasób do puli
            pool.returnResource(db);
        }
    }
}
```

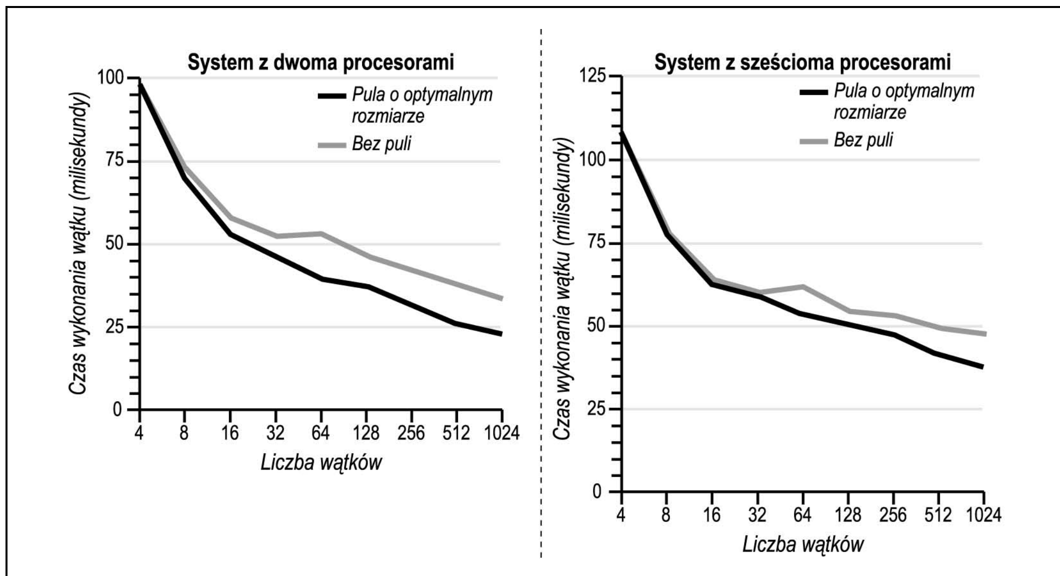
Pule zasobów wyglądają doskonale na papierze, ale czy rzeczywiście pomogą nam parsować pliki *XML*? A jeśli tak, to w jaki sposób należy dobrać właściwy rozmiar puli? Przyjrzyjmy się zatem praktyce zastosowań puli i ich wydajności.

Niezwykle istotne jest dobranie właściwego rozmiaru puli. W naszych testach użyliśmy dwóch serwerów, z których jeden miał dwa procesory, a drugi wyposażony był w sześć procesorów. Oba serwery dysponowały wyjątkowo dużą pamięcią operacyjną. Spodziewaliśmy się, że serwery o takich konfiguracjach będą obsługiwać dużą liczbę wątków. Używając programu testowego podobnego do przedstawionego powyżej, sprawdziliśmy czas parsowania pliku *XML* zawierającego 2 000 wierszy przy zastosowaniu różnych kombinacj liczby wątków i rozmiaru puli. W tabeli 5.1 przedstawione zostały optymalne rozmiary puli uzyskane dla różnej liczby wątków dla obu serwerów. Nie jest zaskoczeniem, że dla parsowania pliku *XML* wymagającego przede wszystkim czasu procesora optymalny rozmiar puli jest zbliżony do liczby procesorów w systemie. Dla zadań intensywniej używających operacji wejścia i wyjścia uzyskalibyśmy z pewnością zupełnie inne wyniki.

Tabela 5.1. Optymalne rozmiary puli dla różnej liczby wątków

Liczba wątków	Optymalny rozmiar puli 2 procesory	Optymalny rozmiar puli 6 procesorów
1	1	1
2	2	2
4	4	4
8	2	8
16	2	15
32	2	6
64	2	6
128	2	6
256	2	6
512	2	4
1024	2	6

Znając optymalne rozmiary puli, możemy ocenić poprawę skalowalności uzyskaną przez ich zastosowanie. W tym celu wykonaliśmy testy dla dwóch wersji programu, z których jedna używała puli o optymalnym rozmiarze, natomiast druga była jej w ogóle pozbawiona. Na rysunku 5.7 przedstawione zostały uzyskane przez nas wyniki jako zależności czasu przetwarzania wątku od liczby wątków.



Rysunek 5.7. Szybkość parsowania plików XML a zastosowanie puli

Zastosowanie puli zdecydowanie poprawia szybkość parsowania plików *XML*, zwłaszcza gdy aktywnych jest więcej niż 32 wątki. Uzyskane wyniki potwierdzają naszą teorię, że zastosowanie puli zwiększa skalowalność w przypadku przetwarzania zadań intensywnie

wykorzystujących procesor, ponieważ ogranicza narzut związany z przełączaniem zbyt wielu zadań. Nie jest również zaskoczeniem, że zastosowanie puli oprócz zwiększenia szybkości przetwarzania spowodowało również mniejsze odchylenia wyników uzyskiwanych w kolejnych próbach.

W niniejszym rozdziale omówiliśmy trzy wzorce, które zwiększają skalowalność warstwy prezentacji. Wzorzec strony asynchronicznej opisuje sposób buforowania danych pobieranych ze źródeł zewnętrznych. Wzorzec filtra buforującego przedstawia metodę buforowania kompletnych stron generowanych dynamicznie. Wzorzec puli zasobów buduje pulę obiektów, których tworzenie związane jest ze znacznym kosztem i umożliwia ich wynajmowanie.