

WZORCE KONSTRUKCYJNE

Budowniczy (Builder), s. 92. Oddziela tworzenie złożonego obiektu od jego reprezentacji, dzięki czemu ten sam proces konstrukcji może prowadzić do powstawania różnych reprezentacji.

Fabryka abstrakcyjna (Abstract Factory), s. 101. Udostępnia interfejs do tworzenia rodzin powiązanych ze sobą lub zależnych od siebie obiektów bez określania ich klas konkretnych.

Metoda wytwarzająca (Factory Method), s. 110. Określa interfejs do tworzenia obiektów, przy czym umożliwia podklasom wyznaczenie klasy danego obiektu. Metoda wytwarzająca umożliwia klasom przekazanie procesu tworzenia egzemplarzy podklasom.

Prototyp (Prototype), s. 120. Określa na podstawie prototypowego egzemplarza rodzaje tworzonych obiektów i generuje nowe obiekty przez kopowanie tego prototypu.

Singleton (Singleton), s. 130. Gwarantuje, że klasa będzie miała tylko jeden egzemplarz, i zapewnia globalny dostęp do niego.

WZORCE STRUKTURALNE

Adapter (Adapter), s. 141. Przekształca interfejs klasy na inny, oczekiwany przez klienta. Adapter umożliwia współdziałanie klasom, które z uwagi na niezgodne interfejsy standardowo nie mogą współpracować ze sobą.

Dekorator (Decorator), s. 152. Dynamicznie dołącza dodatkowe obowiązki do obiektu. Wzorzec ten udostępnia alternatywny elastyczny sposób tworzenia podklas o wzbogaconych funkcjach.

Fasada (Facade), s. 161. Udostępnia jednolity interfejs dla zbioru interfejsów z podsystemem. Fasada określa interfejs wyższego poziomu ułatwiający korzystanie z podsystemów.

Kompozyt (Composite), s. 170. Składa obiekty w struktury drzewiaste odzwierciedlające hierarchię typu część-całość. Wzorzec ten umożliwia klientom traktowanie poszczególnych obiektów i ich złożień w taki sam sposób.

Most (Bridge), s. 181. Oddziela abstrakcję od jej implementacji, dzięki czemu można modyfikować te dwa elementy niezależnie od siebie.

Pełnomocnik (Proxy), s. 191. Udostępnia zastępnik lub reprezentanta innego obiektu w celu kontroliowania dostępu do niego.

Pyłek (Flyweight), s. 201. Wykorzystuje współdzielenie do wydajnej obsługi dużej liczby małych obiektów.

WZORCE OPERACYJNE

Interpreter (Interpreter), s. 217. Określa reprezentację gramatyki języka oraz interpreter, który wykorzystuje tę reprezentację do interpretowania zdań z danego języka.

Iterator (Iterator), s. 230. Zapewnia sekwencyjny dostęp do elementów obiektu złożonego bez ujawniania jego wewnętrznej reprezentacji.

Łańcuch zobowiązań (Chain of Responsibility), s. 244. Pozwala uniknąć nadawcy żądania wiążącego go z odbiorcą, ponieważ umożliwia obsłuszenie żądania więcej niż jednemu obiekowi. Łączy w łańcuch obiekty odbiorcze i przekazuje między nimi żądanie do momentu obsłużenia go.

Mediator (Mediator), s. 254. Określa obiekt kapsułkujący informacje o interakcji między obiekami z danego zbioru. Wzorzec ten pomaga zapewnić luźne powiązanie, ponieważ zapobiega bezpośredniemu odwoływaniu się obiektów do siebie i umożliwia niezależne modyfikowanie interakcji między nimi.

Metoda szablonowa (Template Method), s. 264. Określa szkielet algorytmu i pozostawia doprecyzowanie niektórych jego kroków podklasom. Umożliwia modyfikację niektórych etapów algorytmu w podkласach bez zmiany jego struktury.

Obserwator (Observer), s. 269. Określa zależność „jeden do wielu” między obiekami. Kiedy zmieni się stan jednego z obiektów, wszystkie obiekty zależne od niego są o tym automatycznie powiadomiane i aktualizowane.

Odwiedzający (Visitor), s. 280. Reprezentuje operację wykonywaną na elementach struktury obiektowej. Wzorzec ten umożliwia zdefiniowanie nowej operacji bez zmianiania klas elementów, na których działa.

Pamiątka (Memento), s. 294. Bez naruszania kapsułkowania rejestruje i zapisuje w zewnętrznej jednostce wewnętrzny stan obiektu, co umożliwia późniejsze przywrócenie obiektu według zapamiętanego stanu.

Polecenie (Command), s. 302. Kapsułkuje żądanie w formie obiektu. Umożliwia to parametryzację klienta przy użyciu różnych żądań oraz umieszczanie żądań w kolejkach i dziennikach, a także zapewnia obsługę cofania operacji.

Stan (State), s. 312. Umożliwia obiekowi modyfikację zachowania w wyniku zmiany wewnętrznego stanu. Wygląda to tak, jakby obiekt zmienił klasę.

Strategia (Strategy), s. 321. Określa rodzinę algorytmów, kapsułkuje każdy z nich i umożliwia ich zamienne stosowanie. Wzorzec ten pozwala zmieniać algorytmy niezależnie od korzystających z nich klientów.

RECENZJE KSIĄŻKI

„Od dawna nie przeczytałem tak dobrze napisanej i niezwykle wnikliwej książki [...], autorzy pokazują zasadność stosowania wzorców w najlepszy możliwy sposób — nie przez przedstawianie powodów, ale na podstawie przykładów”.

— Stan Lippman, *C++ Report*

„[...] nowa książka Gammy, Helma, Johnsona i Vlissidesa ma cechy potrzebne do tego, aby wywarła istotny i trwałego wpływ na dziedzinę projektowania oprogramowania. Ponieważ autorzy książki *Wzorce projektowe* sami stwierdzają, że dotyczy ona tylko oprogramowania obiektowego, obawiam się, iż programiści stosujący inne paradigmaty mogą ją zignorować. Byłaby to wielka szkoda. Każdy, kto projektuje oprogramowanie, znajdzie w tej książce coś dla siebie. Wszyscy projektanci korzystają ze wzorców. Lepsze zrozumienie abstrakcji wielokrotnego użytku może podnieść poziom naszej pracy”.

— Tom DeMarco, *IEEE Software*

„Ogólnie uważam, że książka ta stanowi niezwykle cenny i wyjątkowy wkład w projektowanie obiektowe, ponieważ autorzy w zwięzlej i nadającej się do wielokrotnego użytku formie zawarli bogate doświadczenia z tej dziedziny. Z pewnością jest to książka, do której powinieneś często zaglądać w poszukiwaniu skutecznych pomysłów z obszaru projektowania obiektowego. W końcu na tym właśnie polega wielokrotne wykorzystywanie, prawda?”

— Sanjiv Gossain, *Journal of Object-Oriented Programming*

„Ta długo oczekiwana książka jest zgodna z tym, co mówiono na jej temat już od roku. Można porównać ją do podręcznika ze wzorcami architektonicznymi pełnego sprawdzonych i użytecznych projektów. Autorzy wybrali 23 wzorce z dziesięcioleti doświadczeń w programowaniu obiektowym. Geniusz tej książki związany jest ze zdyscyplinowaniem, którego dowodzi liczba wzorców. Podaruj kopię *Wzorców projektowych* każdemu znanemu Ci dobremu programiście, który chce podnieść swoje umiejętności”.

— Larry O'Brien, *Software Development*

„Niezaprzeczelnym faktem jest to, że wzorce mogą trwale zmienić dziedzinę inżynierii oprogramowania i pozwolić jej wznieść się do świata naprawdę eleganckich projektów. Spośród wszystkich poświęconych wzorców książek, które ukazały się do tej pory, *Wzorce projektowe* są zdecydowanie najlepsze. Jest to pozycja, którą należy przeczytać, przestudiować, przyswoić i pokochać. Ta książka na zawsze zmieni sposób, w jaki postrzegasz oprogramowanie”.

— Steve Billow, *Journal of Object-Oriented Programming*

„*Wzorce projektowe* to znacząca książka. Po poświęceniu niedługiego czasu na jej lekturę większość programistów języka C++ będzie mogła zacząć stosować »wzorce« do tworzenia lepszego oprogramowania. Ta pozycja to intelektualna dźwignia — oferuje konkretne narzędzia, które pomagają nam myśleć i wyrażać się w efektywniejszy sposób. Książka ta może całkowicie zmienić to, jak myślisz o programowaniu”.

— Tom Cargill, *C++ Report*

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

WZORCE PROJEKTOWE

Elementy oprogramowania
obiektowego wielokrotnego użytku

**NAUCZ SIĘ WYKORZYSTYWAĆ WZORCE
PROJEKTOWE I UŁATW SOBIE PRACĘ!**

Jak wykorzystać projekty, które już wcześniej okazały się dobre?

Jak stworzyć elastyczny projekt obiektowy?

Jak sprawnie rozwiązywać typowe problemy projektowe?



Tytuł oryginału: Design Patterns

Tłumaczenie: Tomasz Walczak

Projekt okładki: Studio Gravite / Olsztyn;
Obarek, Pokoński, Pazdrijowski, Zaprucki

ISBN: 978-83-246-2662-5

Authorized translation from the English language edition, entitled: *Design Patterns: Elements of Reusable Object-Oriented Software*, ISBN 0201633612, by Erich Gamma; and Richard Helm, published by Pearson Education, Inc., publishing as Addison-Wesley Professional; Copyright © 1995 by Addison-Wesley Longman, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education Inc.

Material from *A Pattern Language: Towns/Buildings/Construction* by Christopher Alexander, copyright © 1977 by Christopher Alexander is reprinted by permission of Oxford University Press, Inc.

Polish language edition published by Helion S.A.
Copyright © 2010.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Materiały graficzne na okładce zostały wykorzystane za zgodą iStockPhoto Inc.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GŁIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
<http://helion.pl/user/opinie?wzcole>
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Dla Karin
— E.G.

Dla Sylvie
— R.H.

Dla Faith
— R.J.

*Dla Dru Ann i Matthew
Księga Jozuego 24, 15*
— J.V.



SPIS TREŚCI

Przedmowa	9
Wstęp	11
Przewodnik dla Czytelników	13
Rozdział 1.	
Wprowadzenie	15
1.1. Czym jest wzorzec projektowy?	16
1.2. Wzorce projektowe w architekturze MVC w języku Smalltalk	18
1.3. Opisywanie wzorców projektowych	20
1.4. Katalog wzorców projektowych	22
1.5. Struktura katalogu	24
1.6. Jak wzorce pomagają rozwiązać problemy projektowe?	26
1.7. Jak wybrać wzorzec projektowy?	42
1.8. Jak stosować wzorce projektowe?	43
Rozdział 2.	
Studium przypadku — projektowanie edytora dokumentów	45
2.1. Problemy projektowe	45
2.2. Struktura dokumentu	47
2.3. Formatowanie	52
2.4. Ozdabianie interfejsu użytkownika	55
2.5. Obsługa wielu standardów wyglądu i działania	59
2.6. Obsługa wielu systemów okienkowych	63
2.7. Działania użytkowników	69
2.8. Sprawdzanie pisowni i podział słów	74
2.9. Podsumowanie	86
Rozdział 3.	
Wzorce konstrukcyjne	87
BUDOWNICZY (BUILDER)	92
FABRYKA ABSTRAKCYJNA (ABSTRACT FACTORY)	101
METODA WYTÓRCZA	110
PROTOTYP (PROTOTYPE)	120
SINGLETON (SINGLETON)	130
Omówienie wzorców konstrukcyjnych	137

Rozdział 4.	Wzorce strukturalne	139
	ADAPTER (ADAPTER)	141
	DEKORATOR (DECORATOR)	152
	FASADA (FAÇADE)	161
	KOMPOZYT (COMPOSITE)	170
	MOST (BRIDGE)	181
	PEŁNOMOCNIK (PROXY)	191
	PYŁEK (FLYWEIGHT)	201
	Omówienie wzorców strukturalnych	213
Rozdział 5.	Wzorce operacyjne	215
	INTERPRETER (INTERPRETER)	217
	ITERATOR (ITERATOR)	230
	ŁAŃCUCH ZOBOWIĄZAŃ (CHAIN OF RESPONSIBILITY)	244
	MEDIATOR (MEDIATOR)	254
	METODA SZABLONOWA (TEMPLATE METHOD)	264
	OBSERWATOR (OBSERVER)	269
	ODWIEDZAJĄCY (VISITOR)	280
	PAMIĘTKA (MEMENTO)	294
	POLECENIE (COMMAND)	302
	STAN (STATE)	312
	STRATEGIA (STRATEGY)	321
	Omówienie wzorców operacyjnych	330
Rozdział 6.	Podsumowanie	335
	6.1. Czego można oczekwać od wzorców projektowych?	335
	6.2. Krótka historia	339
	6.3. Społeczność związana ze wzorcami	340
	6.4. Zaproszenie	342
	6.5. Słowo na zakończenie	342
Dodatek A	Słowniczek	343
Dodatek B	Przewodnik po notacji	347
	B.1. Diagram klas	347
	B.2. Diagram obiektów	349
	B.3. Diagram interakcji	350
Dodatek C	Klasy podstawowe	351
	C.1. List	351
	C.2. Iterator	354
	C.3. ListIterator	354
	C.4. Point	355
	C.5. Rect	355
	Bibliografia	357
	Skorowidz	363

PRZEDMOWA

Książka ta nie jest wprowadzeniem do technologii obiektowych lub projektowania w tym parygmacie. Istnieje wiele dobrych pozycji poświęconych tym zagadnieniom. Zakładamy, że dobrze znasz przynajmniej jeden obiektowy język programowania, a także masz pewne doświadczenie w projektowaniu obiektowym. Z pewnością nie powinieneś musieć zaglądać do słownika, kiedy wspominamy o „typach” i „polimorfizmie” lub dziedziczeniu „interfejsów” (a nie „implementacji”).

Z drugiej strony nie jest to też zaawansowana rozprawa techniczna. Jest to książka o **wzorcach projektowych** opisująca proste i eleganckie rozwiązania specyficznych problemów w obszarze projektowania oprogramowania obiektowego. Wzorce projektowe to rozwiązania, które opracowywano i modyfikowano przez długi czas. Dlatego nie są to rozwiązania, które ludzie wymyślają od razu. Wzorce odzwierciedlają długie godziny przekształcania projektów i wprowadzania zmian w kodzie przez programistów, którzy usilnie starali się zwiększyć elastyczność i możliwość wielokrotnego wykorzystania rozwijanego oprogramowania. Wzorce projektowe ujmują te rozwiązania w zwięzłej i łatwej do zastosowania formie.

Wzorce projektowe nie wymagają korzystania ani z niestandardowych funkcji języka, ani z niezwykłych sztuczek programistycznych, którymi można zadziwić współpracowników i przełożonych. Wszystkie wzorce można zastosować za pomocą standardowych języków obiektowych, choć czasem trzeba włożyć w to nieco więcej pracy niż przy implementowaniu doraźnych rozwiązań. Jednak dodatkowy wysiłek z pewnością się opłaci z uwagi na większą elastyczność i możliwość powtórnego wykorzystania oprogramowania.

Kiedy zrozumiesz wzorce projektowe i będziesz mógł powiedzieć „Aha!” (a nie tylko: „O co chodzi?”), na zawsze przestaniesz myśleć o projektowaniu obiektowym w dawny sposób. Zaczniesz dostrzegać rozwiązania, dzięki którym Twoje projekty staną się elastyczniejsze, bardziej modularne, zrozumiałe i nadające się do wielokrotnego użytku. W końcu dlatego interesuje Cię programowanie obiektowe, prawda?

Słowo ostrzeżenia i zachęty — nie martw się, jeśli w czasie czytania tej książki po raz pierwszy nie zrozumiesz jej w pełni. Nam samym się to nie udało, kiedy ją pisaliśmy! Pamiętaj, że nie jest to pozycja, którą należy raz przeczytać i odłożyć na półkę. Mamy nadzieję, że będziesz wciąż do niej wracał po pomysły projektowe i inspiracje.

Książka ta powstawała przez długi czas. Odwiedziła z nami cztery kraje, widziała trzy śluby autorów i narodziny dwóch (niespokrewnionych) potomków. Wiele osób uczestniczyło w jej powstawaniu. Specjalne podziękowania należą się Bruce'owi Andersonowi, Kentowi Beckowi i André Weinandowi za inspirację i porady. Dziękujemy też wszystkim, którzy oceniali wersje wstępne książki. Byli to: Roger Bielefeld, Grady Booch, Tom Cargill, Marshall Cline, Ralph Hyre, Brian Kernighan, Thomas Laliberty, Mark Lorenz, Arthur Riel, Doug Schmidt, Clovis Tondo, Steve Vinoski i Rebecca Wirfs-Brock. Jesteśmy także wdzięczni za pomoc i cierpliwość zespołowi z wydawnictwa Addison-Wesley. W jego skład wchodzili: Kate Habib, Tiffany Moore, Lisa Raffaele, Pradeepa Siva i John Wait. Specjalne podziękowania składamy Carlowi Kesslerowi, Danny'emu Sabbah i Markowi Wegmanowi z IBM Research za niesłabnące wsparcie prac nad tą książką.

Na zakończenie — choć równie gorąco — dziękujemy wszystkim ludziom, którzy przez internet i innymi kanałami przesyłali do nas komentarze na temat różnych wersji wzorców i słowa zachęty, a także informowali nas, że to, co robimy, jest cenne. Te osoby to między innymi: Jon Avotins, Steve Berczuk, Julian Berdych, Matthias Bohlen, John Brant, Allan Clarke, Paul Chisholm, Jens Coldewey, Dave Collins, Jim Coplien, Don Dwiggins, Gabriele Elia, Doug Felt, Brian Foote, Denis Fortin, Ward Harold, Hermann Hueni, Nayeem Islam, Bikramjit Kalra, Paul Keefer, Thomas Kofler, Doug Lea, Dan LaLiberte, James Long, Ann Louise Luu, Pundi Madhavan, Brian Marick, Robert Martin, Dave McComb, Carl McConnell, Christine Mingins, Hanspeter Mössenböck, Eric Newton, Marianne Ozkan, Roxsan Payette, Larry Podmolik, George Radin, Sita Ramakrishnan, Russ Ramirez, Alexander Ran, Dirk Riehle, Bryan Rosenberg, Aamod Sane, Duri Schmidt, Robert Seidl, Xin Shu i Bill Walker.

Nie uważamy, że opisana tu kolekcja wzorców projektowych jest kompletna i niezmieniona. Jest to bardziej zapis naszych obecnych przemyśleń na temat projektowania. Chętnie usłyszyszmy komentarze na ich temat — krytykę przedstawionych przykładów, odwołania i znane zastosowania, które pominęliśmy, a także wzorce warte dodania. Można się z nami skontaktować za pośrednictwem wydawnictwa Addison-Wesley lub wysyłając e-mail na adres *design-patterns@cs.uiuc.edu*. Można też otrzymać elektroniczną wersję kodu źródłowego z punktów „Przykładowy kod” przez wysłanie wiadomości „send design pattern source” na adres *design-patterns-source@cs.uiuc.edu*. Najnowsze informacje i aktualizacje znajdują się obecnie na stronie <http://st-www.cs.illinois.edu/users/patterns/DPBook/DPBook.html>.

*Mountain View, Kalifornia
Montreal, Quebec
Urbana, Illinois
Hawthorne, Nowy Jork*

Sierpień 1994

E. G.
R. H.
R. J.
J. V.

WSTĘP

Wszystkie dobrze ustrukturyzowane architektury obiektowe są pełne wzorców. Jeden ze sposobów używanych przeze mnie do pomiaru jakości systemu obiektowego polega na ocenie, czy programiści poświęcili wystarczająco dużo uwagi jednolitym metodom współdziałania między obiektytami. Koncentracja w czasie rozwijania systemu na takich mechanizmach może prowadzić do powstania mniej rozbudowanej, prostszej i dużo bardziej zrozumiałej architektury niż w przypadku zignorowania wzorców.

W innych dziedzinach wzorce już od dawna są uznawane za ważny aspekt tworzenia złożonych systemów. Christopher Alexander i jego współpracownicy to prawdopodobnie pierwsi ludzie, którzy zaproponowali wykorzystanie języka wzorców do opisu budynków oraz miast. Jego koncepcje i wkład innych osób znalazły obecnie zastosowanie w świecie oprogramowania obiektowego. Używanie wzorców projektowych przy rozwijaniu oprogramowania ma pomóc programistom wykorzystać wiedzę innych doświadczonych architektów.

W tej książce Erich Gamma, Richard Helm, Ralph Johnson i John Vlissides przedstawiają podstawy związane ze wzorcami projektowymi, a następnie proponują katalog takich wzorców. Dlatego książka ta wnosi dwojakiego wkład w dziedzinę. Po pierwsze, pokazuje znaczenie wzorców w projektowaniu złożonych systemów. Po drugie, przedstawia bardzo praktyczny zestaw starannie opracowanych wzorców, które zawodowy projektant może zastosować przy budowaniu konkretnych aplikacji.

Miałem zaszczyt bezpośrednio współpracować z niektórymi autorami tej książki przy projektowaniu architektury systemów. Dużo się od nich nauczyłem i podejrzewam, że Ty także dowiesz się wielu rzeczy dzięki zapoznaniu się z tą pozycją.

Grady Booch
Chief Scientist, Rational Software Corporation



PRZEWODNIK DLA CZYTELNIKÓW

Książka ma dwie główne części. Pierwsza (rozdziały 1. i 2.) opisuje, czym są wzorce projektowe i w jaki sposób pomagają projektować oprogramowanie obiektowe. Znajdują się tu studia przypadków pokazujące, jak stosować wzorce projektowe w praktyce. Druga część książki (rozdziały 3., 4. i 5.) obejmuje katalog wzorców projektowych.

Spis wzorców zajmuje większość książki. Rozdziały z tej części dzielą wzorce projektowe na trzy kategorie: wzorce konstrukcyjne, strukturalne i operacyjne. Z katalogu wzorców można korzystać na kilka sposobów. Jedna z możliwości to zapoznanie się ze spisem od początku do końca lub samo przejrzenie poszczególnych wzorców. Inne podejście to przestudiowanie jednego z rozdziałów. Pomoże to zobaczyć różnice między mocno powiązanymi ze sobą wzorcami.

Odsyłacze do innych wzorców można potraktować jako logiczne ścieżki do poruszania się po katalogu. To podejście pozwoli zrozumieć, w jaki sposób wzorce są powiązane ze sobą, jak je łączyć i które wzorce dobrze ze sobą współdziałają. Rysunek 1.1 (s. 25) przedstawia te powiązania w formie graficznej.

Jeszcze inny sposób zapoznawania się z katalogiem jest oparty na problemach. Przejdź do podrozdziału 1.6 (s. 26), aby się dowiedzieć, jakie standardowe trudności występują w obszarze projektowania oprogramowania obiektowego wielokrotnego użytku. Następnie zapoznaj się ze wzorcami, które są rozwiązaniem poszczególnych problemów. Niektóre osoby najpierw przeglądają cały katalog, a *następnie* stosują podejście oparte na problemach, aby zastosować wzorce we własnych projektach.

Jeśli nie masz doświadczenia w projektowaniu obiektowym, zacznij od najprostszych i najczęściej stosowanych wzorców:

- Adapter (s. 141)
- Dekorator (s. 152)
- Fabryka abstrakcyjna (s. 101)
- Kompozyt (s. 170)
- Metoda szablonowa (s. 264)
- Metoda twórca (s. 110)
- Obserwator (s. 269)
- Strategia (s. 321)

Trudno jest znaleźć system obiektowy, w którym nie zastosowano przynajmniej kilku z tych wzorców. W dużych systemach obecne są prawie wszystkie z nich. Ten podzbiór pomoże Ci zrozumieć wzorce projektowe, a bardziej ogólnie — dobre projekty obiektowe.

Rozdział 1.

Wprowadzenie

Projektowanie oprogramowania obiektowego jest trudne, a jeśli ma się ono nadawać do *wielokrotnego użytku*, zadanie jest jeszcze bardziej skomplikowane. Trzeba określić odpowiednie obiekty, podzielić je na klasy o odpowiedniej szczegółowości, zdefiniować interfejsy klas i hierarchie dziedziczenia oraz ustalić kluczowe relacje między elementami. Projekt powinien być dostosowany do rozwiązywanego problemu, a jednocześnie na tyle ogólny, aby uwzględniał przyszłe trudności i wymogi. Należy też dążyć do wyeliminowania (lub przynajmniej zminalizowania) późniejszego przekształcania projektu. Według doświadczonych projektantów obiektowych opracowanie przy pierwszym podejściu elastycznego i nadającego się do wielokrotnego użytku projektu jest trudne (jeśli nie niemożliwe). Zanim projekt zostanie ukończony, projektanci zwykle kilkakrotnie próbują go ponownie wykorzystać, za każdym razem wprowadzając poprawki.

Jednak doświadczeni projektanci obiektowi tworzą dobre projekty. Tymczasem nowicjusze są przytłoczeni mnogością dostępnych możliwości i często wracają do technik nieobiektowych, z których korzystali wcześniej. Początkującym dużo czasu zajmuje nauczenie się, jakie cechy ma dobry projekt obiektowy. Doświadczeni projektanci najwyraźniej wiedzą coś, z czego nie zdają sobie sprawy nowicjusze. Czym jest to „coś”?

Eksperci wiedzą między innymi, że *nie* należy rozwiązywać każdego problemu od podstaw. Zamiast tego doświadczeni projektanci ponownie wykorzystują projekty, które okazały się skuteczne w przeszłości. Kiedy znajdą dobre rozwiązanie, stosują je wielokrotnie. To podejście to jedna z cech charakteryzujących ekspertów. Dlatego wiele systemów obiektowych obejmuje powtarzające się wzorce klas i komunikujących się ze sobą obiektów. Te wzorce służą do rozwiązywania specyficznych problemów projektowych i sprawiają, że projekty obiektowe są elastyczniejsze i bardziej eleganckie, a w przyszłości można je ponownie wykorzystać. Pomaga to projektantom wielokrotnie używać udanych projektów przez oparcie nowych rozwiązań na wcześniejszych doświadczeniach. Projektant znający takie wzorce może po natrafieniu na problem natychmiast je wykorzystać, zamiast wymyślać od nowa.

W opisie tego zagadnienia pomoże analogia. Autorzy powieści i sztuk rzadko wymyślają fabułę od podstaw. Zamiast tego korzystają ze wzorców, takich jak „bohater tragiczny” (Makbet, Hamlet itd.) lub „powieść romantyczna” (niezliczone romanse). W ten sam sposób projektanci

obiektowi stosują wzorce w rodzaju „reprezentowanie stanu za pomocą obiektów” i „dekorowanie obiektów w celu ułatwienia dodawania i usuwania funkcji”. Po poznaniu wzorca wiele decyzji projektowych można podejmować automatycznie.

Wszyscy znamy wartość doświadczenia projektowego. Ile razy doświadczyłeś zjawiska *déjà vu*, czyli uczucia, że rozwiązałeś już wcześniej dany problem, nie wiedząc jednak, kiedy i w jaki sposób to zrobileś? Jeśli potrafiłbyś przypomnieć sobie szczegóły poprzedniego problemu i jego rozwiązanie, mógłbyś wykorzystać doświadczenie, zamiast wymyślać projekt od nowa. Jednak projektanci oprogramowania nie są biegli w zapisywaniu swych doświadczeń, z których mogliby skorzystać inni.

Celem tej książki jest zarejestrowanie w formie **wzorców projektowych** doświadczenia z obszaru projektowania oprogramowania obiektowego. W każdym z tych wzorców systematycznie nazwaliśmy, wyjaśniliśmy i oceniliśmy ważny oraz powtarzający się projekt z systemów obiektowych. Chcieliśmy ująć doświadczenie projektowe w postaci, którą inni będą mogli efektywnie wykorzystać. Dlatego udokumentowaliśmy niektóre z najważniejszych wzorców projektowych i przedstawiliśmy je w katalogu.

Wzorce projektowe ułatwiają powtórne wykorzystanie udanych projektów i architektur. Zapisanie sprawdzonych technik w formie wzorców ułatwia ich zastosowanie programistom nowych systemów. Wzorce pomagają wybrać możliwości projektowe umożliwiające powtórne wykorzystanie systemu i uniknąć rozwiązań, które to utrudniają. Wzorce pozwalają nawet ulepszyć dokumentację i usprawnić konserwację istniejących systemów przez zapewnienie jawnej specyfikacji klas i interakcji między obiektemi oraz przeznaczenia tych elementów. Ujmijmy to prosto — wzorce projektowe ułatwiają projektantom szybsze opracowanie prawnego projektu.

Żaden z wzorców projektowych z tej książki nie opisuje nowego lub niesprawdzonego projektu. Umieściliśmy tu tylko rozwiązania zastosowane więcej niż jeden raz w różnych systemach. Większość z omawianych projektów nie została nigdy wcześniej udokumentowana. Niektóre z nich są częścią niepisanej wiedzy społeczności programistów obiektowych, a inne to elementy udanych systemów obiektowych. Dla nowicjuszy nauka na podstawie tych dwóch źródeł nie jest łatwa. Dlatego choć omawiane projekty nie są nowe, ujeliśmy je w nowatorski i przystępny sposób — jako katalog wzorców projektowych o jednolitym formacie.

Mimo rozmiarów tej książki przedstawione w niej wzorce projektowe są tylko niewielkim wykiniem wiedzy ekspertów. Nie umieściliśmy tu żadnych wzorców związanych z przetwarzaniem równoległym, rozproszonym lub w czasie rzeczywistym. Nie omówiliśmy wzorców specyficznych dla danej dziedziny. Nie pokazujemy, jak budować interfejsy użytkownika, pisać sterowniki urządzeń lub korzystać z obiektowych baz danych. Każdy z tych obszarów związany jest z odrębnymi wzorcami. Warto, aby także je ktoś skatalogował.

1.1. CZYM JEST WZORZEC PROJEKTOWY?

Christopher Alexander stwierdził, że: „Każdy wzorzec opisuje problem powtarzający się w danym środowisku i istotę rozwiązania tego problemu w taki sposób, że można wykorzystać określone rozwiązanie milion razy i nigdy nie zrobić tego tak samo [AIS-77, s. X]. Choć Alexander

mówią o wzorcach dotyczących budynków i miast, jego słowa są prawdziwe także w przypadku obiektowych wzorców projektowych. Rozwiązania w tej dziedzinie wyrażamy w kategoriach obiektów i interfejsów, a nie ścian i drzwi, jednak istotą wzorców obu rodzajów jest rozwiązanie problemu w danym kontekście.

Ogólnie wzorce składają się z czterech kluczowych elementów:

1. **Nazwa wzorca** to skrót, którego można użyć do opisania w jednym lub dwóch słowach problemu projektowego, jego rozwiązania i konsekwencji zastosowania tego rozwiązania. Nazwanie wzorca bezpośrednio powiększa słownik projektowy i pozwala tworzyć projekty na wyższym poziomie abstrakcji. Słownik wzorców umożliwia rozmawianie o nich ze współpracownikami (a nawet z samym sobą) i stosowanie ich nazw w dokumentacji. Ułatwia to myślenie o projektach i opisywanie ich oraz wad i zalet rozwiązania innym. Znalezienie dobrych nazw było jednym z najtrudniejszych zadań przy tworzeniu katalogu wzorców.
2. Opis **problemu** określa, kiedy należy stosować wzorzec. Ta część wyjaśnia trudność i jej kontekst. Może to być omówienie specyficznego problemu projektowego, na przykład zapisu algorytmów w formie obiektów. Może to być też opis specyficznych dla nieelastycznych projektów struktur klas lub obiektów. Czasem problem obejmuje listę warunków, które muszą być spełnione, aby zastosowanie wzorca było uzasadnione.
3. **Rozwiązanie** to opis elementów składających się na projekt, ich przeznaczenia oraz relacji i współdziałania między nimi. Rozwiązanie nie jest wyjaśnieniem konkretnego projektu lub określonej implementacji, ponieważ wzorzec przypomina szablon — można go zastosować w wielu różnych sytuacjach. Wzorzec obejmuje abstrakcyjny opis problemu projektowego i ogólny układ elementów (tu są to klasy i obiekty).
4. **Konsekwencje** to efekty oraz koszty i zyski wynikające z zastosowania wzorca. Choć w opisie decyzji projektowych konsekwencje często nie są jawnie przedstawiane, mają kluczowe znaczenie przy ocenie różnych możliwości i pozwalają zrozumieć koszty oraz korzyści związane z danym wzorcem.

Konsekwencje w świecie oprogramowania często dotyczą kompromisu związanego z pamięcią i czasem przetwarzania. Mogą też obejmować zagadnienia specyficzne dla języka i implementacji. Ponieważ wielokrotne wykorzystanie projektów obiektowych często ma duże znaczenie, konsekwencje zastosowania wzorca obejmują jego wpływ na elastyczność, rozszerzalność i przenośność systemu. Jawnie wymienienie konsekwencji pomaga je zrozumieć i ocenić.

Określenie tego, co jest, a co nie jest wzorcem, zależy od punktu widzenia. To, co jedna osoba uzna za wzorzec, dla innej może być tylko prostym blokiem konstrukcyjnym. W tej książce skoncentrowaliśmy się na wzorcach z określonego poziomu abstrakcji. *Wzorce projektowe* to nie projekty w rodzaju list powiązanych i tablic haszujących, które można zapisać w formie klas oraz wielokrotnie wykorzystać w niezmienionej postaci. Nie są to też specyficzne dla danej dziedziny złożone projekty całych aplikacji lub podsystemów. W tej książce wzorce projektowe to opisy komunikujących się obiektów i klas przeznaczonych do rozwiązywania ogólnych problemów projektowych w określonym kontekście.

Za pomocą wzorców projektowych nadajemy nazwy standardowym strukturom projektowym, tworzymy ich abstrakcje i wskazujemy ich kluczowe aspekty. Robimy to tak, aby dana struktura była przydatna do tworzenia projektów obiektowych nadających się do wielokrotnego użytku. Wzorzec projektowy określa klasy i egzemplarze klas, ich role i współdziałanie między nimi, a także podział zadań. Każdy wzorzec projektowy dotyczy konkretnego problemu lub zagadnienia z obszaru projektowania obiektowego. Opisujemy, kiedy należy stosować dany wzorzec i czy można go wykorzystać w obliczu innych ograniczeń projektowych. Wyjaśniamy też konsekwencje jego zastosowania oraz płynące z tego koszty i zyski. Ponieważ projekt ostatecznie trzeba zaimplementować, we wzorcach przedstawiamy też przykładowy kod w językach C++ i (czasem) Smalltalk.

Choć wzorce projektowe opisują projekty obiektowe, są oparte na praktycznych rozwiązańach stosowanych w podstawowych obiektowych językach programowania, takich jak Smalltalk i C++, a nie na językach proceduralnych (Pascal, C, Ada) lub bardziej dynamicznych językach obiektowych (CLOS, Dylan, Self). Wybraliśmy języki Smalltalk i C++ z pragmatycznych przyczyn — mamy doświadczenie w korzystaniu z nich, a ponadto zyskują one coraz większą popularność.

To, jakie języki programowania stosujemy, ma znaczenie, ponieważ wpływa na perspektywę. We wzorcach zakładamy, że dostępne są funkcje z języków Smalltalk i C++, dlatego ich wybór określa, co można (i czego nie można) łatwo zaimplementować. Gdybyśmy zastosowali języki proceduralne, moglibyśmy dodać wzorce Dziedziczenie, Kapsulkowanie i Polimorfizm. Ponadto niektóre z opisywanych tu wzorców są bezpośrednio dostępne w mniej popularnych językach obiektowych. Na przykład język CLOS udostępnia wielometody, co zmniejsza konieczność stosowania wzorca Odwiedzający (s. 280). Nawet języki Smalltalk i C++ na tyle różnią się od siebie, że niektóre wzorce można zastosować łatwiej w jednym z nich — zobacz na przykład wzorzec Iterator (s. 230).

1.2. WZORCE PROJEKTOWE W ARCHITEKTURZE MVC W JĘZYKU SMALLTALK

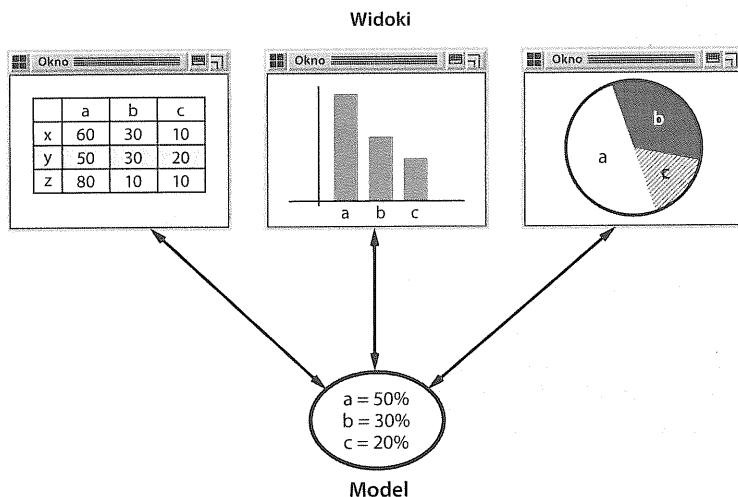
W języku Smalltalk-80 do tworzenia interfejsów użytkownika służą trzy klasy: modelu, widoku i kontrolera (ang. *Model/View/Controller* — MVC) [KP88]. Przeanalizowanie wzorców projektowych stosowanych w tej architekturze powinno pomóc w zrozumieniu, co oznacza dla nas pojęcie „wzorzec”.

Architektura MVC obejmuje obiekty trzech rodzajów. Model to obiekt aplikacji, View odpowiada prezentacji widocznej na ekranie, a Controller określa, jak interfejs ma reagować na działania użytkownika. Przed wprowadzeniem architektury MVC w interfejsach użytkownika wszystkie te obiekty były zwykle połączone ze sobą. MVC rozdziela je, co zwiększa elastyczność i możliwość powtórnego wykorzystania projektu.

Architektura MVC rozdziela widoki i modele przez ustanowienie protokołu subskrypcji i powiadamiania służącego do komunikowania się między nimi. Widok musi sprawdzać, czy jego wygląd odzwierciedla stan modelu. Kiedy dane modelu się zmieniają, powiadomi on o tym zależne od niego widoki. W odpowiedzi każdy widok może zaktualizować swój wygląd.

To podejście umożliwia połączenie modelu z wieloma widokami i udostępnienie różnych wersji prezentacji. Można też utworzyć nowe widoki dla modelu bez konieczności modyfikowania go.

Poniższy diagram ilustruje model i trzy widoki (dla uproszczenia pominęliśmy tu kontrolery). Model obejmuje dane, a widoki określające arkusz kalkulacyjny, histogram i wykres kołowy wyświetlają te informacje na różne sposoby. Model komunikuje się z widokami, kiedy zmienia się jego wartości, a widoki kontaktują się z modelem, aby uzyskać dostęp do danych.



W podstawowej postaci przykład ten przedstawia projekt oddzielający widoki od modeli. Jednak podejście to można zastosować do rozwiązania bardziej ogólnego problemu — do rozdzielenia obiektów, aby zmiany w jednym z nich mogły wpływać na dowolną liczbę innych, i to bez konieczności informowania modyfikowanego obiektu o szczegółach działania pozostały. Ten ogólniejszy projekt opisujemy we wzorcu projektowym Obserwator (s. 269).

Inną cechą architektury MVC jest możliwość zagnieźdzania widoków. Na przykład panel sterowania z przyciskami można zaimplementować jako złożony widok obejmujący zagnieżdżone widoki przycisków. Interfejs użytkownika w narzędziu do inspekcji obiektów może składać się z zagnieżdżonych widoków, które można ponownie wykorzystać w debuggerze. Architektura MVC obsługuje widoki zagnieżdżone za pomocą klasy CompositeView (jest to podklasa klasy View). Obiekty klasy CompositeView działają podobnie jak obiekty klasy View. Widoków złożonych można używać wszędzie tam, gdzie zwykłych widoków, jednak obejmują one także widoki zagnieżdżone i zarządzają nimi.

Także tu możemy pomyśleć o tym rozwiązaniu jak o projekcie pozwalającym traktować widok złożony w taki sam sposób jak jeden z jego komponentów. Jednak projekt ten można wykorzystać na potrzeby bardziej ogólnego problemu, który przytrafia się zawsze, kiedy chcemy połączyć obiekty i traktować ich grupę jak jeden obiekt. Ten ogólniejszy projekt opisujemy we wzorcu projektowym Kompozyt (s. 170). Umożliwia on zbudowanie hierarchii klas, w której niektóre podklasy określają obiekty proste (na przykład Button), a inne — obiekty złożone (CompositeView) łączące proste elementy w bardziej skomplikowane jednostki.

Architektura MVC umożliwia ponadto zmianę sposobu reagowania widoku na działania użytkownika bez modyfikowania wizualnej warstwy prezentacji. Możliwe, że programista zechce zmienić sposób reagowania aplikacji na wciśnięcie klawiszy klawiatury lub zastosować menu podrzczne zamiast klawiszy polecenia. W MVC mechanizm reagowania umieszczono w obiekcie `Controller`. Istnieje cała hierarchia klas kontrolerów, co ułatwia zbudowanie nowego kontrolera na podstawie jednego z istniejących.

Widok korzysta z egzemplarza podklasy klasy `Controller` do realizowania konkretnej strategii reagowania. Aby zastosować inną strategię, wystarczy zamienić ten egzemplarz na kontroler innego rodzaju. Można nawet zastępować kontrolery w czasie wykonywania programu, aby zmodyfikować sposób reagowania widoku na działania użytkownika. Na przykład w celu wyłączenia widoku (żeby nie reagował na działania użytkownika) wystarczy przypisać do niego kontroler ignorujący zdarzenia wejściowe.

Związek między widokiem i kontrolerem to przykład zastosowania wzorca projektowego `Strategia` (s. 321). Określa on obiekt reprezentujący algorytm. Wzorzec ten jest przydatny, kiedy programista chce statycznie lub dynamicznie zastępować algorytmy, korzysta z wielu wersji algorytmu lub napisał algorytm ze złożonymi strukturami danych i chce je zakapsulkować.

W MVC zastosowano też inne wzorce projektowe — na przykład Metodę wytwarzającą (s. 110) do określania domyślnej klasy kontrolera widoku lub Dekoratora (s. 152) w celu dodania przewijania do widoku. Jednak główne relacje w architekturze MVC wyznaczane są przez wzorce `Obserwator`, `Kompozyt` i `Strategia`.

1.3. OPISYWANIE WZORCÓW PROJEKTOWYCH

W jaki sposób opisujemy wzorce projektowe? Schematy graficzne — choć ważne i użyteczne — nie wystarczą. Ujmują one jedynie produkt końcowy procesu projektowania w formie relacji między klasami i obiektami. Aby powtórnie wykorzystać projekt, trzeba ponadto zapisać związane z nim decyzje, inne możliwości oraz koszty i korzyści jego użycia. Ważne są też konkretne przykłady, ponieważ pomagają zobaczyć zastosowanie projektu w praktyce.

Wzorce projektowe opisujemy za pomocą spójnego formatu. Każdy wzorzec jest podzielony na fragmenty według przedstawionego dalej szablonu. Nadaje on jednolitą strukturę informacjom, co ułatwia uczenie się, porównywanie i stosowanie wzorców projektowych.

NAZWA I KATEGORIA WZORCA

Nazwa wzorca zwięźle określa jego istotę. Dobra nazwa jest niezwykle ważna, ponieważ będzie częścią słownika projektowego. Kategoria wzorca jest oparta na schemacie przedstawionym w podrozdziale 1.5.

PRZEZNACZENIE

Jest to krótka odpowiedź na następujące pytania: „Jak działa ten wzorzec?”, „Jakie jest uzasadnienie powstania wzorca i jego przeznaczenie?”, „Jakie konkretne kwestie lub problemy projektowe rozwiązuje dany wzorzec?“.

INNE NAZWY

Inne znane nazwy wzorca (jeśli występują).

UZASADNIENIE

Jest to scenariusz ilustrujący problem projektowy i sposób, w jaki klasy i obiekty ze wzorca go rozwiązuje. Przykład pomaga zrozumieć przedstawiony dalej bardziej abstrakcyjny opis wzorca.

WARUNKI STOSOWANIA

W jakich sytuacjach można zastosować dany wzorzec? Jakie przykładowe złe projekty można poprawić przy jego użyciu? Jak rozpoznać takie sytuacje?

STRUKTURA

Graficzna reprezentacja klas ze wzorca przedstawiona w notacji opartej na OMT (ang. *Object Modeling Technique*) [RBP-91]. Stosujemy też diagramy interakcji [JCJO92, Boo94] do zilustrowania sekwencji żądań i współdziałania między obiektami. W dodatku B opisujemy szczegółowo obie te notacje.

ELEMENTY

Klasy i (lub) obiekty pojawiające się we wzorcu projektowym i ich zadania.

WSPÓŁDZIAŁANIE

Sposób współdziałania elementów przy realizowaniu zadań.

KONSEKWENCJE

W jaki sposób wzorzec realizuje stawiane mu cele? Jakie są koszty i korzyści oraz efekty jego zastosowania? Jakie elementy struktury systemu można niezależnie modyfikować dzięki wykorzystaniu wzorca?

IMPLEMENTACJA

O jakich pułapkach, wskazówkach i technikach należy pamiętać w czasie implementowania wzorca? Czy występują problemy specyficzne dla języka?

PRZYKŁADOWY KOD

Fragmenty kodu ilustrujące możliwą realizację wzorca w językach C++ lub Smalltalk.

ZNANE ZASTOSOWANIA

Przykłady wykorzystania wzorca w rzeczywistych systemach. Przedstawiamy przynajmniej po dwa przykłady z różnych dziedzin.

POWIĄZANE WZORCE

Które wzorce są mocno powiązane z danym? Jakie ważne różnice między nimi występują? Z którymi innymi wzorcami należy stosować opisywany?

W dodatkach przedstawiamy dodatkowe informacje pomagające zrozumieć wzorce i ich analizy. Dodatek A to słowniczek stosowanych przez nas pojęć. Wspomnieliśmy już o dodatku B, gdzie omawiamy różne notacje. Wybrane aspekty notacji opisujemy też przy ich wprowadzaniu w dalszych analizach. Dodatek C obejmuje kod źródłowy podstawowych klas stosowanych w przykładowym kodzie.

1.4. KATALOG WZORCÓW PROJEKTOWYCH

Katalog zaczyna się od strony 79 i obejmuje 23 wzorce projektowe. W tym miejscu w ramach przeglądu przedstawiamy ich nazwy i przeznaczenie. Liczba w nawiasach po każdej nazwie wzorca to numer strony z opisem danego wzorca (konwencję tę stosujemy w całej książce).

Adapter (s. 141). Przekształca interfejs klasy na inny, oczekiwany przez klienta. Adapter umożliwia współdziałanie klasom, które z uwagi na niezgodne interfejsy standardowo nie mogą współpracować ze sobą.

Budowniczy (s. 92). Oddziela tworzenie złożonego obiektu od jego reprezentacji, dzięki czemu ten sam proces konstrukcji może prowadzić do powstawania różnych reprezentacji.

Dekorator (s. 152). Dynamicznie dołącza dodatkowe obowiązki do obiektu. Wzorzec ten udostępnia alternatywny elastyczny sposób tworzenia podklas o wzbogaconych funkcjach.

Fabryka abstrakcyjna (s. 101). Udostępnia interfejs do tworzenia rodzin powiązanych ze sobą lub zależnych od siebie obiektów bez określania ich klas konkretnych.

Fasada (s. 161). Udostępnia jednolity interfejs dla zbioru interfejsów z podsystemu. Fasada określa interfejs wyższego poziomu ułatwiający korzystanie z podsystemów.

Interpreter (s. 217). Określa reprezentację gramatyki języka oraz interpreter, który wykorzystuje tę reprezentację do interpretowania zdań z danego języka.

Iterator (s. 230). Zapewnia sekwencyjny dostęp do elementów obiektu złożonego bez ujawniania jego wewnętrznej reprezentacji.

Kompozyt (s. 170). Składa obiekty w struktury drzewiaste odzwierciedlające hierarchię typu całość-część. Wzorzec ten umożliwia klientom traktowanie poszczególnych obiektów i ich złożień w taki sam sposób.

Łańcuch zobowiązań (s. 244). Pozwala uniknąć wiązania nadawcy żądania z jego odbiorcą, ponieważ umożliwia obsłużenie żądania więcej niż jednemu obiekowi. Łączy w łańcuch obiekty odbiorcze i przekazuje między nimi żądanie do momentu obsłużenia go.

Mediator (s. 254). Określa obiekt kapsułkujący informacje o interakcji między obiektami z danego zbioru. Wzorzec ten pomaga zapewnić luźne powiązanie, ponieważ zapobiega bezpośredniemu odwoływaniu się obiektów do siebie i umożliwia niezależne modyfikowanie interakcji między nimi.

Metoda szablonowa (s. 264). Określa szkielet algorytmu i pozostawia doprecyzowanie niektórych jego kroków podklasom. Umożliwia modyfikację niektórych etapów algorytmu w podklasach bez zmiany jego struktury.

Metoda twórcza (s. 110). Określa interfejs do tworzenia obiektów, przy czym umożliwia podklasom wyznaczenie klasy danego obiektu. Metoda twórcza umożliwia klasom przekazanie procesu tworzenia egzemplarzy podklasom.

Most (s. 181). Oddziela abstrakcję od jej implementacji, dzięki czemu można modyfikować te dwa elementy niezależnie od siebie.

Obserwator (s. 269). Określa zależność „jeden do wielu” między obiektami. Kiedy zmieni się stan jednego z obiektów, wszystkie obiekty zależne od niego są o tym automatycznie powiadamiane i aktualizowane.

Odwiedzający (s. 280). Reprezentuje operację wykonywaną na elementach struktury obiektowej. Wzorzec ten umożliwia zdefiniowanie nowej operacji bez zmianiania klas elementów, na których działa.

Pamiątka (s. 294). Bez naruszania kapsułkowania rejestruje i zapisuje w zewnętrznej jednostce wewnętrzny stan obiektu, co umożliwia późniejsze przywrócenie obiektu według zapamiętanego stanu.

Pełnomocnik (s. 191). Udostępnia zastępnik lub reprezentanta innego obiektu w celu kontrowłowania dostępu do niego.

Polecenie (s. 302). Kapsułkuje żądanie w formie obiektu. Umożliwia to parametryzację klienta przy użyciu różnych żądań oraz umieszczanie żądań w kolejkach i dziennikach, a także zapewnia obsługę cofania operacji.

Prototyp (s. 120). Określa na podstawie prototypowego egzemplarza rodzaje tworzonych obiektów i generuje nowe obiekty przez kopowanie tego prototypu.

Pyłek (s. 201). Wykorzystuje współdzielenie do wydajnej obsługi dużej liczby małych obiektów.

Singleton (s. 130). Gwarantuje, że klasa będzie miała tylko jeden egzemplarz, i zapewnia globalny dostęp do niego.

Stan (s. 312). Umożliwia obiektowi modyfikację zachowania w wyniku zmiany wewnętrznego stanu. Wygląda to tak, jakby obiekt zmienił klasę.

Strategia (s. 321). Określa rodzinę algorytmów, kapsułkuje każdy z nich i umożliwia ich zamienne stosowanie. Wzorzec ten pozwala zmieniać algorytmy niezależnie od korzystających z nich klientów.

1.5. STRUKTURA KATALOGU

Wzorce projektowe różnią się poziomem szczegółowości i abstrakcji. Ponieważ jest ich wiele, potrzebny jest sposób na ich uporządkowanie. W tym podrozdziale klasyfikujemy wzorce projektowe, aby można odwoływać się do rodzin powiązanych wzorców. Ten podział pomoże w szybkim poznaniu wzorców z katalogu, a ponadto może ułatwić wyszukiwanie nowych wzorców.

Wzorce projektowe podzieliliśmy według dwóch kryteriów (tabela 1.1). Pierwsze z nich, zwane **rodzajem**, dotyczy działania wzorca. Wyróżniamy wzorce **konstrukcyjne**, **strukturalne** i **operacyjne**. Wzorce konstrukcyjne związane są z procesem tworzenia obiektów. Wzorce strukturalne dotyczą składania klas lub obiektów. Wzorce operacyjne określają sposób współdziałania klas lub obiektów oraz podział zadań między nimi.

TABELA 1.1. Przestrzeń wzorców projektowych

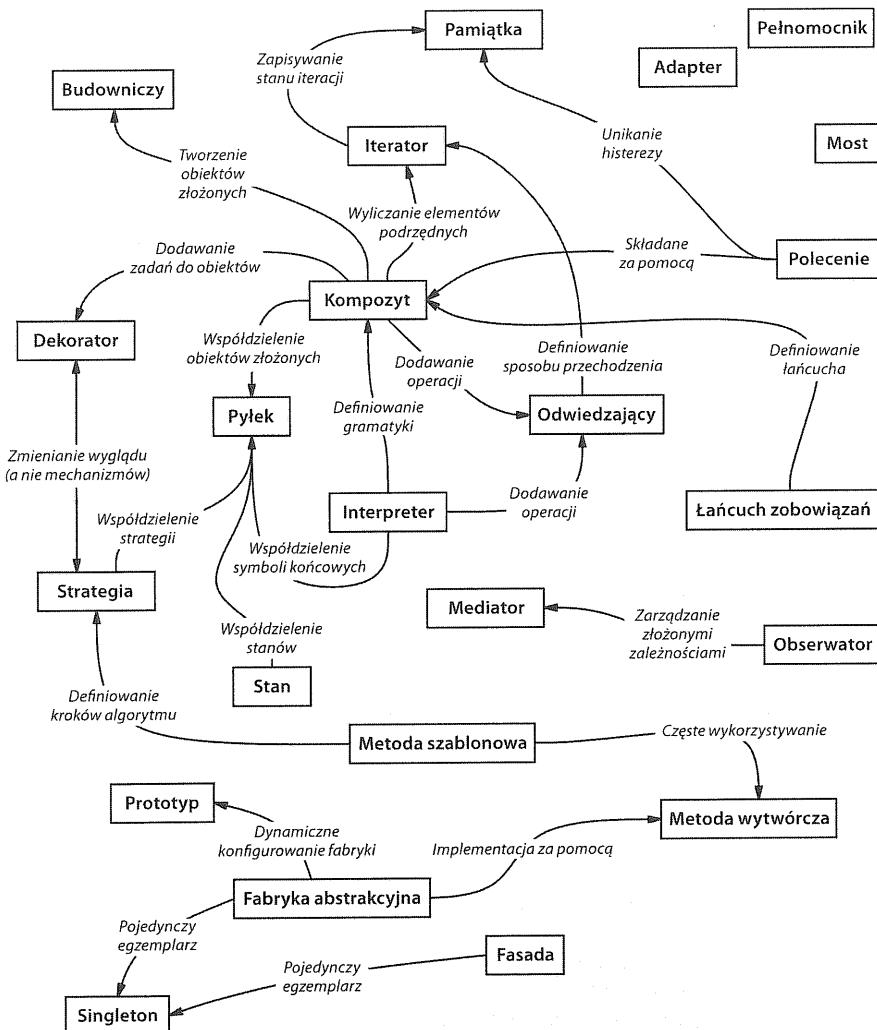
		Rodzaj		
Zasięg	Klasa	Konstrukcyjne	Strukturalne	Operacyjne
		Metoda wytwórcza (s. 110)	Adapter (klasy; s. 141)	Interpreter (s. 217) Metoda szablonowa (s. 264)
	Obiekt	Fabryka abstrakcyjna (s. 101) Budowniczy (s. 92) Prototyp (s. 120) Singleton (s. 130)	Adapter (obiekty; s. 141) Most (s. 181) Kompozyt (s. 170) Dekorator (s. 152) Fasada (s. 161) Pyłek (s. 201) Pełnomocnik (s. 191)	Łańcuch zobowiązań (s. 244) Polecenie (s. 302) Iterator (s. 230) Mediator (s. 254) Pamiątka (s. 294) Obserwator (s. 269) Stan (s. 312) Strategia (s. 321) Odwiedzający (s. 280)

Drugie kryterium, tak zwany **zasięg**, informuje, czy wzorzec dotyczy głównie klas czy obiektów. Wzorce związane z klasami określają relacje między klasami i ich podklasami. Te związki są wyznaczane przez dziedziczenie, co oznacza, że są statyczne (ustalane w czasie komplikacji). Wzorce związane z obiektami dotyczą relacji między tymi jednostkami. Związki tego typu można zmieniać w czasie wykonywania programu, dlatego są bardziej dynamiczne. W prawie wszystkich wzorcach w pewnym stopniu wykorzystano dziedziczenie. Dlatego jedyne wzorce określone jako specyficzne dla klas to te, które dotyczą głównie relacji między klasami. Warto zauważyć, że większość wzorców obejmuje zasięgiem obiekty.

We wzorcach konstrukcyjnych dotyczących klas część procesu tworzenia obiektów jest przekazywana do podklas. W podobnych wzorcach dotyczących obiektów fragment tego procesu jest realizowany przez inne obiekty. We wzorcach strukturalnych związanych z klasami zastosowano dziedziczenie do budowania klas, natomiast wzorce z tej kategorii dotyczące obiektów ilustrują sposoby składania obiektów. We wzorcach operacyjnych powiązanych z klasami zastosowano dziedziczenie do opisu algorytmów i przepływu sterowania. Podobne wzorce dotyczące obiektów ilustrują współdziałanie grup obiektów przy realizacji zadań niemożliwych do wykonania przez pojedynczy obiekt.

Istnieją też inne sposoby porządkowania wzorców. Niektóre wzorce są często stosowane razem. Na przykład Kompozyt często współwystępuje z wzorcami Iterotor lub Odwiedzający. Część wzorców można stosować zamiennie. Na przykład Prototyp często można zastąpić Fabryką abstrakcyjną. Część wzorców prowadzi do utworzenia podobnego projektu, choć ich przeznaczenie jest inne. Na przykład diagramy strukturalne dla wzorców Kompozyt i Dekorator są podobne do siebie.

Jeszcze inny sposób uporządkowania wzorców projektowych związany jest z zależnościami między nimi opisymi w poszczególnych punktach „Powiązane wzorce”. Rysunek 1.1 przedstawia te relacje w formie graficznej.



RYSUNEK 1.1. Relacje między wzorcami projektowymi

Widać więc, że wzorce projektowe można uporządkować na wiele sposobów. Liczne podejścia do myślenia o wzorcach pomagają lepiej zrozumieć, jak wzorce działają, kiedy należy je stosować oraz jakie są różnice i podobieństwa między nimi.

1.6. JAK WZORCE POMAGAJĄ ROZWIĄZAĆ PROBLEMY PROJEKTOWE?

Wzorce projektowe na wiele rozmaitych sposobów pomagają rozwiązywać liczne codzienne problemy, z którymi stykają się projektanci obiektowi. Oto kilka takich trudności wraz z wyjaśnieniami, jak wzorce pozwalają się z nimi uporać.

ZNAJDOWANIE ODPOWIEDNICH OBIEKTÓW

Programy obiektowe składają się z obiektów. **Obiekt** obejmuje zarówno dane, jak i operujące na nich procedury. Procedury są zwykle nazywane **metodami** lub **operacjami**. Obiekt uruchamia operację, kiedy otrzyma **żądanie** (lub **komunikat**) od **klienta**.

Zgłoszenie żądania to *jedyny* sposób na zmuszenie obiektu do uruchomienia operacji. Z kolei jej wywołanie to *jedyny* sposób na zmodyfikowanie wewnętrznych danych obiektu. Z uwagi na te ograniczenia mówi się, że wewnętrzny stan obiektu jest **zakapsułkowany**. Nie można bezpośrednio uzyskać dostępu do stanu obiektu, a jego reprezentacja jest niewidoczna poza nim.

W projektowaniu obiektowym trudność sprawia podział systemu na obiekty. Zadanie to jest skomplikowane, ponieważ należy uwzględnić wiele czynników: kapsułkowanie, szczegółowość, zależności, elastyczność, wydajność, zmiany, możliwość powtórnego wykorzystania itd. Wszystkie te aspekty wpływają na podział systemu i często sugerują sprzeczne rozwiązania.

Metodologie projektowania obiektowego zalecają wiele różnych podejść. Można przygotować opis problemu, wyodrębnić z niego rzecznowniki i czasowniki oraz utworzyć odpowiadające im klasy i operacje. Można też skoncentrować się na współdziałaniu i zadaniach systemu. Jeszcze inne podejście to utworzenie modelu rzeczywistego świata i przekształcenie obiektów wykrytych w trakcie analiz na projekt. Zawsze będą toczyć się spory na temat tego, które z tych rozwiązań jest najlepsze.

Wiele obiektów w projekcie jest efektem analiz. Jednak projekty obiektowe często ostatecznie obejmują klasy, dla których nie istnieją odpowiedniki w rzeczywistym świecie. Niektóre z tych elementów to klasy niskopoziomowe, reprezentujące na przykład tablice. Inne to klasy ze znacznie wyższego poziomu. Na przykład wzorzec Kompozyt (s. 170) przedstawia abstrakcję umożliwiającą traktowanie obiektów w taki sam sposób. Struktura ta nie ma odpowiednika fizycznego. Bezpośrednie modelowanie świata rzeczywistego prowadzi do powstawania systemów, które odzwierciedlają dzisiejsze warunki, ale już niekoniecznie stan jutrzejszy. Abstrakcje powstające w czasie projektowania to klucz do tworzenia elastycznych projektów.

Wzorce projektowe pomagają zidentyfikować mniej oczywiste abstrakcje i odpowiadające im obiekty. Na przykład obiekty reprezentujące proces lub algorytm nie występują w naturze, a są kluczowym elementem elastycznych projektów. Wzorzec Strategia (s. 321) opisuje, jak zaimplementować wymienne rodziny algorytmów. Wzorzec Stan (s. 312) ilustruje, jak reprezentować stan każdej jednostki za pomocą obiektu. Takie obiekty trudno jest wymyślić w czasie analiz, a nawet na wczesnych etapach rozwijania projektu; są odkrywane później, w czasie pracy nad zwiększaniem elastyczności projektu i możliwości jego powtórnego wykorzystania.

OKREŚLANIE POZIOMU SZCZEGÓŁOWOŚCI OBIEKTU

Obiekty mogą znacznie różnić się między sobą pod względem wielkości i liczby. Obiekty mogą reprezentować wszystko — od sprzętu po całe aplikacje. Jak określić, co powinno być obiektem?

Wzorce projektowe rozwiązuje także tę kwestię. Wzorzec Fasada (s. 161) opisuje, jak przedstawić kompletne podsystemy jako obiekty, a wzorzec Pyłek (s. 201) — jak obsługiwać dużą liczbę bardzo szczegółowych obiektów. Inne wzorce projektowe przedstawiają konkretne sposoby podziału obiektów na mniejsze części. Wzorce Fabryka abstrakcyjna (s. 101) i Budowniczy (s. 92) pozwalają tworzyć obiekty służące wyłącznie do generowania innych obiektów. Wzorce Odwiedzający (s. 280) i Polecanie (s. 302) umożliwiają budowanie obiektów przeznaczonych tylko do implementowania żądań kierowanych do innych obiektów lub ich grup.

OKREŚLANIE INTERFEJSÓW OBIEKTÓW

Każda operacja zadeklarowana w obiekcie ma nazwę, obiekty przyjmowane jako parametry i zwracaną wartość. Elementy te składają się na **sygnaturę**. Zestaw wszystkich sygnatur operacji obiektu to jego **interfejs**. Interfejs obiektu określa kompletny zbiór żądań, jakie można skierować do obiektu. Przesłać można dowolne żądanie pasujące do interfejsu obiektu.

Typ to nazwa stosowana do określania danego interfejsu. Mówimy, że obiekt ma typ **Window**, jeśli przyjmuje wszystkie żądania uruchomienia operacji zdefiniowanych w interfejsie o nazwie **Window**. Obiekt może mieć wiele typów, a ten sam typ mogą mieć bardzo różne obiekty. Część interfejsu obiektu może być określona przez jeden typ, a inne części — przez odmienne typy. Dwa obiekty tego samego typu muszą mieć wspólną tylko część interfejsu. Interfejsy mogą obejmować inne interfejsy. Mówimy, że typ jest **podtypem** innego, jeśli jego interfejs obejmuje interfejs **nadtypu**. Często określa się, że podtyp *dziedziczy* interfejs typu nadzawanego.

Interfejsy są podstawowym elementem systemów obiektowych. Obiekty są znane tylko poprzez interfejsy. Nie można dowiedzieć się czegokolwiek o obiekcie lub zażądać od niego wykonania operacji z pominięciem interfejsu. Interfejs nie określa implementacji obiektu. W różnych obiektach żądania mogą być realizowane w inny sposób. Oznacza to, że dwa obiekty o zupełnie innej implementacji mogą mieć identyczny interfejs.

To, którą operację obiekt uruchomi po otrzymaniu żądania, zależy *zarówno* od żądania, jak i samego obiektu. Dwa obiekty obsługujące te same żądania mogą mieć różne implementacje operacji uruchamianych w celu spełnienia tych żądań. Łączenie w czasie wykonywania programu żądania skierowanego do obiektu z jedną z jego operacji to tak zwane **wiązanie dynamiczne**.

Wiązanie dynamiczne powoduje, że zgłoszenie żądania nie determinuje konkretnej implementacji do momentu uruchomienia programu. Oznacza to, że można pisać programy korzystające z obiektu o określonym interfejsie, ponieważ wiadomo, że dowolny obiekt o właściwym interfejsie obsługuje dane żądanie. Ponadto wiązanie dynamiczne pozwala zastępować w czasie wykonywania programu jedne obiekty innymi o identycznych interfejsach. Takie podmienianie jest nazywane **polimorfizmem**. Jest to niezwykle ważna cecha systemów obiektowych. Polimorfizm sprawia, że przy tworzeniu obiektów klienckich nie trzeba czynić wielu założeń na temat innych obiektów — wystarczy wiedzieć, że te ostatnie obsługują konkretny interfejs. Polimorfizm upraszcza definiowanie klientów, pozwala rozdzielić obiekty i umożliwia zmianę powiązań między nimi w czasie wykonywania programu.

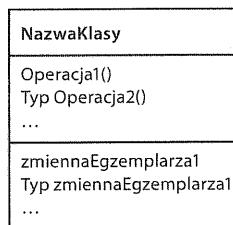
Wzorce projektowe pomagają definiować interfejsy, ponieważ określają ich kluczowe elementy i rodzaje danych przesyłanych do interfejsu. Wzorzec projektowy może też wskazywać, czego *nie* należy umieszczać w interfejsie. Dobrym przykładem takiego wzorca jest Pamiątka (s. 294). Opisuje on, jak kapsułkować i zapisywać wewnętrzny stan obiektu, aby później odtworzyć obiekt w tym samym stanie. Ten wzorzec wymaga, aby obiekt Memento miał dwa interfejsy — ograniczony (pozwalający klientom przechowywać i kopiować zapamiętane dane) i uprzywilejowany (dostępny tylko dla pierwotnego obiektu do zapisywania i pobierania stanu z pamiętki).

Wzorce projektowe określają też relacje między interfejsami. Często wymagają, aby niektóre klasy miały podobne interfejsy, lub nakładają ograniczenia na interfejsy pewnych klas. Na przykład wzorce Dekorator (s. 152) i Pełnomocnik (s. 191) wymagają, aby interfejsy obiektów Decorator i Proxy były identyczne jak obiektu dekorowanego lub korzystającego z pełnomocnika. Według wzorca Odwiedzający (s. 280) interfejs Visitor musi odzwierciedlać wszystkie klasy obiektów, które odwiedzający może odwiedzić.

OKREŚLANIE IMPLEMENTACJI OBIEKTÓW

Do tej pory niewiele pisaliśmy o tym, jak definiowane są obiekty. Implementację obiektu wyznacza jego **klasa**. Określa ona wewnętrzne dane obiektu, jego reprezentację i operacje wykonywane przez obiekt.

W notacji opartej na OMT (opisujemy ją w dodatku B) klasa jest przedstawiana jako prostokąt z nazwą klasy napisaną pogrubioną czcionką. Pod tą nazwą wymienione są operacje zapisane zwykłą czcionką. Jeszcze niżej znajdują się dane zdefiniowane w klasie. Nazwę klasy od operacji i operacje od danych oddzielają linie.



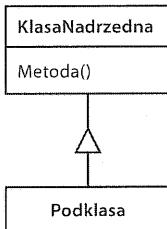
Typy zwracanych danych i zmiennych egzemplarza są opcjonalne, ponieważ zakładamy, że nie korzystamy z języka ze statyczną kontrolą typów.

Obiekty powstają w wyniku **tworzenia egzemplarza** klasy. Mówimy, że obiekt jest **egzemplarzem** danej klasy. W procesie tworzenia egzemplarza klasy przydzielana jest pamięć na wewnętrzne dane obiektu (składające się ze **zmiennych egzemplarza**), a operacje są wiązane z tymi danymi. Poprzez tworzenie egzemplarzy klasy można wygenerować wiele podobnych obiektów.

Przerywana linia ze strzałką służy do oznaczania klasy, która tworzy egzemplarze innej klasy. Strzałka prowadzi do klasy generowanych obiektów.



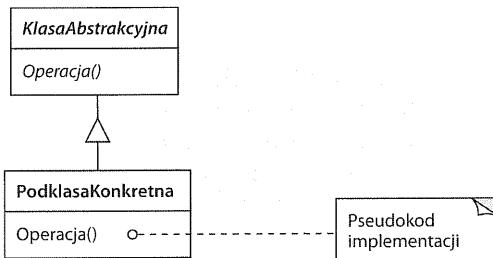
Nowe klasy można definiować na podstawie istniejących za pomocą **dziedziczenia klas**. Jeśli podklaśa dziedziczy po **klasie nadzędnej**, obejmuje definicje wszystkich danych i operacji z określonej klasy nadzędnej. Egzemplarze podklaśy obejmują wszystkie dane zdefiniowane przez tą podklaśę i jej klasy nadzędne, dlatego mogą wykonywać wszystkie operacje zdefiniowane w podklaście i klasach nadzędnych. Relację dziedziczenia oznaczamy za pomocą pionowej linii i trójkąta.



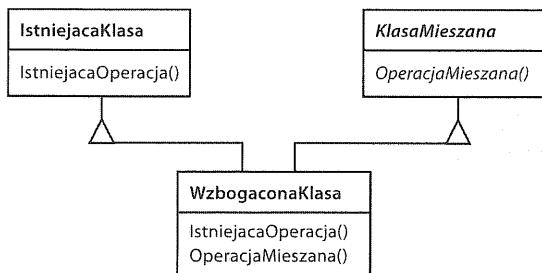
Klasy abstrakcyjne służą głównie do definiowania wspólnego interfejsu dla ich podklaś. Implementacja części lub wszystkich operacji klasy abstrakcyjnej znajduje się w podklaśach. Z tego powodu nie można tworzyć egzemplarzy klas abstrakcyjnych. Operacje zadeklarowane, ale niezaimplementowane w klasie abstrakcyjnej, to **operacje abstrakcyjne**. Klasa, które nie są abstrakcyjne, to **klasy konkretne**.

W podklaśach można dopracowywać lub modyfikować działanie operacji z klas nadzędnych. Ujmijmy to dokładniej — w klasie można **przesłonić** metodę zdefiniowaną w klasie nadzędnej. Przesłanianie umożliwia podklaśom samodzielna obsługę żądań (bez pomocy klas nadzędnych). Dziedziczenie pozwala definiować klasę przez wzbogacanie innych klas, co ułatwia tworzenie rodzin obiektów o podobnych funkcjach.

Nazwy klas abstrakcyjnych zapisywane są kursywą w celu odróżnienia ich od klas konkretnych. Pochyla czcionka służy też do wymieniania operacji abstrakcyjnych. Diagram może obejmować pseudokod implementacji metody. Wtedy kod znajduje się w ramce z zagęszczonym rogiem połączoną z nazwą implementowanej metody przerywaną linią.



Klasa mieszana (ang. *mix-in class*) ma udostępniać opcjonalne interfejsy lub funkcje w innych klasach. Przypomina klasę abstrakcyjną, ponieważ klasa obu tych rodzajów nie służy do tworzenia egzemplarzy. Klasa mieszana wymaga zastosowania wielodziedziczenia.



DZIEDZICZENIE KLAS I INTERFEJSÓW

Ważne jest, aby zrozumieć różnicę między *klasą* i *typem* obiektu.

Klasa obiektu określa, w jaki sposób dany obiekt jest zaimplementowany. Klasa wyznacza wewnętrzny stan obiektu i implementację jego metod. Z drugiej strony typ obiektu określa tylko jego interfejs, czyli zestaw żądań, które obiekt obsługuje. Obiekt może mieć wiele typów, a obiekty różnych klas mogą mieć ten sam typ.

Oczywiście klasy i typy są ze sobą mocno związane. Ponieważ klasa określa operacje, które obiekt może wykonywać, wyznacza też typ obiektu. Kiedy mówimy, że obiekt jest egzemplarzem klasy, wnioskujemy, iż obsługuje interfejs definiowany przez tę klasę.

W niektórych językach, takich jak C++ i Eiffel, klasy służą do określania zarówno typu, jak i implementacji obiektu. W programach w języku Smalltalk typ zmiennych nie jest deklarowany. Dlatego kompilator nie sprawdza, czy typy obiektów przypisywane do zmiennej są podtypami typu tej zmiennej. Wysłanie komunikatu wymaga sprawdzenia, czy klasa odbiorcy obsługuje dany komunikat, jednak nie trzeba określać, czy odbiorca jest egzemplarzem danej klasy.

Należy też zrozumieć różnicę między dziedziczeniem klas i interfejsów (tworzeniem podtypów). Dziedziczenie klasy polega na zdefiniowaniu implementacji obiektu w kategoriach implementacji innego obiektu. Ujmijmy to krótko — jest to mechanizm do współużytkowania kodu i reprezentacji obiektów. Z drugiej strony dziedziczenie interfejsu (tworzenie podtypu) określa, kiedy jednego obiektu można użyć zamiast innego.

Łatwo jest pomylić te dwa zagadnienia, ponieważ w wielu językach nie są one wyraźnie rozróżnione. W językach C++ i Eiffel dziedziczenie dotyczy zarówno interfejsu, jak i implementacji. Standardowy sposób dziedziczenia interfejsu w języku C++ polega na dziedziczeniu publicznym po klasie z (czysto) wirtualnymi funkcjami składowymi. Dziedziczenie samego interfejsu można zrealizować w C++ przez dziedziczenie publiczne po klasach czysto abstrakcyjnych. Przybliżeniem dziedziczenia samej implementacji lub klasy jest dziedziczenie prywatne. W języku Smalltalk dziedziczenie oznacza tylko dziedziczenie implementacji. Do zmiennej można przypisać egzemplarze dowolnej klasy, jeśli obsługuje ona operacje wykonywane na wartości danej zmiennej.

Choć w większości języków programowania nie ma rozróżnienia na dziedziczenie interfejsów i implementacji, użytkownicy w praktyce uwzględniają ten podział. Programiści języka Smalltalk zwykle pracują tak, jakby podklasy były podtypami (choć istnieją od tego dobrze znane wyjątki [Coo92]). Programiści języka C++ manipulują obiektami za pomocą typów zdefiniowanych w postaci klas abstrakcyjnych.

Wiele wzorców projektowych wymaga uwzględnienia opisanego rozróżnienia. Na przykład obiekty w Łańcuchu zobowiązań (s. 244) muszą mieć wspólny typ, jednak zwykle nie mają tej samej implementacji. We wzorcu Kompozyt (s. 170) klasa Component określa wspólny interfejs, natomiast klasa Composite obejmuje wspólną implementację. Wzorce Polecenie (s. 302), Obserwator (s. 269), Stan (s. 312) i Strategia (s. 321) są często realizowane za pomocą klas abstrakcyjnych będących czystym interfejsem.

PROGRAMOWANIE POD KĄTEM INTERFEJSU, A NIE IMPLEMENTACJI

Dziedziczenie klas to w istocie mechanizm służący tylko do wzbogacania funkcji aplikacji przez ponowne wykorzystanie możliwości klas nadzędnych. Podejście to umożliwia szybkie definiowanie obiektów nowego rodzaju w kategoriach istniejących obiektów. Pozwala to przygotować nowe implementacje niemal bez nakładów pracy — przez odziedziczenie większości potrzebnych elementów po istniejących klasach.

Jednak ponowne wykorzystanie implementacji to dopiero połowa sukcesu. Ważna jest też możliwość zastosowania dziedziczenia do definiowania rodzin obiektów o *identycznych* interfejsach (zwykle przez dziedziczenie po klasach abstrakcyjnych). Dlaczego? Ponieważ polymorfizm jest oparty na tej technice.

Jeśli dziedziczenie jest stosowane starannie (niektórzy powiedzą, że *prawidłowo*), wszystkie klasy pochodne od klasy abstrakcyjnej współdzielą jej interfejs. Oznacza to, że w podklasach programista tylko dodaje lub przesyła metody i nie ukrywa operacji z klasy nadzędnej. Wszystkie podklasy mogą następnie reagować na żądania opisane w interfejsie danej klasy abstrakcyjnej, a tym samym są jej podtypami.

Są dwie zalety manipulowania obiektami wyłącznie za pomocą interfejsu zdefiniowanego w klasach abstrakcyjnych:

1. Klienci nie muszą znać typów używanych obiektów, o ile tylko obiekty te mają interfejs oczekiwany przez klienta.
2. Klienci nie muszą znać klas z implementacją używanych obiektów. Wiedzą jedynie, które klasy abstrakcyjne definiują interfejs.

Te cechy w tak dużym stopniu zmniejszają zależności implementacyjne między podsystemami, że warto stosować się do poniższej zasady tworzenia projektów obiektowych wielokrotnego użytku:

Programuj pod kątem interfejsu, a nie implementacji.

Nie deklaruj zmiennych jako egzemplarzy określonych klas konkretnych. Zamiast tego wykorzystaj tylko interfejs zdefiniowany w klasie abstrakcyjnej. Zobaczysz, że jest to powtarzający się motyw we wzorcach projektowych omawianych w tej książce.

Oczywiście, w pewnym miejscu systemu trzeba utworzyć egzemplarze klas konkretnych (czyli określić specyfczną implementację). Umożliwiają to wzorce konstrukcyjne: Fabryka abstrakcyjna (s. 101), Budowniczy (s. 92), Metoda wytwarzca (s. 110), Prototyp (s. 120) i Singleton (s. 130). Poprzez abstrakcyjne ujęcie procesu tworzenia obiektów wzorce te zapewniają kilka sposobów na niewidoczne łączenie interfejsu z implementacją w momencie powstawania obiektu. Wzorce z tej grupy gwarantują, że system będzie napisany w kategoriach interfejsów, a nie implementacji.

ZASTOSOWANIE MECHANIZMÓW POWTÓRNEGO WYKORZYSTANIA ROZWIĄZANIA

Większość osób potrafi zrozumieć działanie obiektów, interfejsów, klas i dziedziczenia. Trudność polega na zastosowaniu ich do zbudowania oprogramowania elastycznego i możliwego do wielokrotnego wykorzystania. Wzorce projektowe pokazują, jak to zrobić.

DZIEDZICZENIE A SKŁADANIE

Dwie najczęściej stosowane techniki powtórnego wykorzystania funkcji w systemach obiektowych to dziedziczenie klas i **składanie obiektów**. Wyjaśniliśmy już, że dziedziczenie klas umożliwia definiowanie implementacji jednej klasy w kategoriach innej. Ponowne wykorzystanie przez tworzenie podklas jest często nazywane **otwartym powtórnym wykorzystaniem** (ang. *white-box reuse*). Pojęcie „otwarте” dotyczy widoczności — przy dziedziczeniu wewnętrzne mechanizmy klas nadzędnych są często widoczne w jej podklasach.

Alternatywą dla dziedziczenia klas jest składanie obiektów. W tym podejściu nowe mechanizmy można dodać przez zestawianie lub **składanie obiektów** w celu uzyskania bardziej złożonych funkcji. Składanie obiektów wymaga, aby łączone obiekty miały dobrze zdefiniowane interfejsy. Te metoda to **zamknięte powtórne wykorzystanie** (ang. *black-box reuse*), ponieważ żadne wewnętrzne mechanizmy obiektów nie są widoczne. Obiekty te mają postać „czarnych skrzynek”.

Dziedziczenie i składanie mają specyficzne zalety i wady. Dziedziczenie klas jest definiowane statycznie w czasie komplikacji, a korzystanie z tej techniki nie sprawia problemów, ponieważ jest ona obsługiwana bezpośrednio przez języki programowania. Ponadto dziedziczenie klas ułatwia modyfikowanie powtórnie wykorzystywanej implementacji. Jeśli podklaśa przesyłania tylko niektóre metody, może to mieć wpływ także na dziedziczone operacje (przy założeniu, że wywołują one przesyłane metody).

Jednak dziedziczenie klas ma też pewne wady. Po pierwsze, nie można zmienić odziedziczonej po klasie nadzędnej implementacji w czasie wykonywania programu, ponieważ dziedziczenie jest definiowane na etapie komplikacji. Po drugie — i jest to zwykle większy problem — klasa nadzędna często określa przynajmniej część reprezentacji fizycznej podklas. Ponieważ dziedziczenie powoduje ujawnienie w podklaście szczegółów implementacji klasy nadzędnej, często mówi się, że „dziedziczenie narusza zasady kapsułkowania” [Sny86]. Implementacja podklaści staje się w tak znacznym stopniu zależna od implementacji klasy nadzędnej, że zmiany w implementacji klasy nadzędnej wymuszają modyfikacje w podklaści.

Zależności implementacyjne mogą spowodować problemy przy próbie ponownego wykorzystania podklaści. Jeśli dowolny aspekt odziedziczonej implementacji jest nieodpowiedni dla nowej dziedziny problemowej, klasę nadzędną trzeba napisać od nowa lub zastąpić właściwszą klasą. Rozwiązaniem tego problemu jest dziedziczenie tylko po klasach abstrakcyjnych, ponieważ mają one ubogą implementację lub całkowicie są jej pozbawione.

Składanie obiektów jest definiowane dynamicznie w czasie wykonywania programu przez zapisywanie w obiektach referencji do innych obiektów. Składanie wymaga zgodności między interfejsami poszczególnych obiektów, co z kolei oznacza konieczność stosowania starannie

zaprojektowanych interfejsów, które nie uniemożliwiają połączenia jednego obiektu z wieloma innymi. Ma to jednak pewną zaletę. Ponieważ dostęp do obiektów istnieje tylko poprzez interfejsy, nie powoduje to naruszenia kapsułkowania. Dowolny obiekt można w czasie wykonywania programu zastąpić innym, pod warunkiem że mają one ten sam typ. Ponadto z uwagi na to, że implementacja obiektu jest pisana w kategoriach interfejsów, powstaje dużo mniej zależności implementacyjnych.

Stosowanie składania obiektów wpływa na projekt systemu w jeszcze jeden sposób. Przedkładanie tej techniki nad dziedziczenie klas pomaga zachować kapsułkowanie każdej klasy i skoncentrować się w niej na jednym zadaniu. Klasy i ich hierarchie pozostają wtedy niewielkie, a rozrośnięcie się ich do rozmiaru utrudniającego zarządzanie jest mniej prawdopodobne. Z drugiej strony projekt oparty na składaniu obejmuje więcej obiektów (nawet jeśli klas jest mniej), a działanie systemu nie jest zdefiniowane w jednej klasie, ale zależy od relacji między obiekttami.

Prowadzi to do drugiej zasady projektowania obiektowego:

Przedkładaj kompozycję obiektów nad dziedziczenie klas.

W idealnych warunkach tworzenie nowych komponentów w celu powtórnego wykorzystania rozwiązania nie powinno być konieczne. Należy dążyć do tego, aby wszystkie potrzebne funkcje można było uzyskać przez połączenie istniejących komponentów za pomocą składania. Jednak rzadko jest to możliwe, ponieważ w praktyce zestaw dostępnych komponentów nigdy nie jest wystarczająco bogaty. Powtórne wykorzystanie przez dziedziczenie ułatwia tworzenie nowych komponentów, które można połączyć z istniejącymi. Dlatego dziedziczenie i składanie obiektów się uzupełniają.

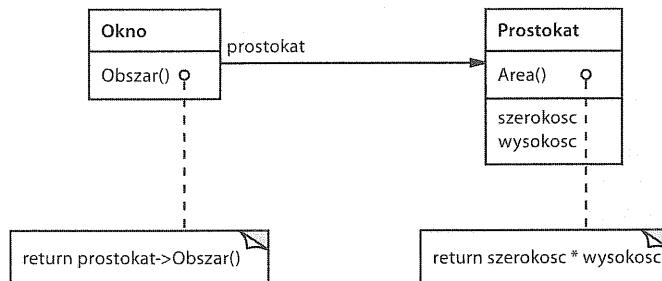
Jednak według naszego doświadczenia projektanci nadużywają dziedziczenia, podczas gdy zastosowanie w większym stopniu składania obiektów często pozwala sprawić, że projekt będzie lepiej nadawał się do powtórnego wykorzystania (i będzie prostszy). We wzorcach projektowych wielokrotnie zobaczysz przykłady zastosowania składania obiektów.

DELEGOWANIE

Delegowanie to sposób, dzięki któremu składanie może okazać się równie skuteczną techniką powtórnego wykorzystania rozwiązania jak dziedziczenie [Lie86, JZ91]. W podejściu tym do obsługi żądania służą *dwa* obiekty — odbiorca deleguje do wykonania operacji **delegata**. Przypomina to przekazywanie żądań przez podklasy do klas nadzorczych. Jednak przy stosowaniu dziedziczenia w odziedziczonych operacjach zawsze można wskazać odbiorcę za pomocą zmiennej składowej *this* (język C++) lub *self* (język Smalltalk). Aby uzyskać ten sam efekt za pomocą delegowania, odbiorca musi przekazać sam siebie do delegata, co umożliwia wskazanie odbiorcy w delegowanej operacji.

Na przykład zamiast tworzyć na podstawie klasy Prostokat podkласę Okno (ponieważ okna są prostokątne), w klasie Okno można powtórnie wykorzystać działanie klasy Prostokat przez wykorzystanie jej jako zmiennej egzemplarza i *oddelegowanie* do niej specyficznych zadań. Ujmijmy to inaczej — klasa Okno zamiast być klasą Prostokat, będzie ją *obejmować*. W tym modelu klasa Okno musi bezpośrednio przekazywać żądania do egzemplarza klasy Prostokat, natomiast w opisany wcześniej rozwiązaniu odziedziczyłyby potrzebne operacje.

Poniższy diagram ilustruje klasę Okno delegującą wywołanie metody Obszar do egzemplarza klasy Prostokat.



Zwykła strzałka wskazuje, że klasa przechowuje referencję do egzemplarza innej klasy. Ta referencja ma opcjonalną nazwę, którą tu jest „prostokat”.

Główną zaletą delegowania jest to, że technika ta ułatwia składanie zachowań w czasie wykonywania programu i modyfikowanie ich zestawu. Aby w czasie wykonywania programu zmienić okno z prostokątnego na okrągłe, wystarczy zastąpić egzemplarz klasy Prostokat egzemplarzem klasy Okrag (zakładamy, że obiekty te mają ten sam typ).

Delegowanie ma pewną wadę, od której nie są wolne też inne techniki zwiększenia elastyczności oprogramowania za pomocą składania obiektów. Chodzi o to, że dynamiczne, wysoce sparametryzowane oprogramowanie trudniej jest zrozumieć niż bardziej statyczne programy. Występują też problemy z wydajnością w czasie wykonywania programu, jednak w dłuższej perspektywie ważniejszy jest brak wydajności programistów. Zastosowanie delegowania to dobry wybór projektowy, jeśli technika ta więcej rzeczy upraszcza, niż komplikuje. Nie łatwo jest wymyślić reguły, która pozwoli precyjnie określić, kiedy należy stosować to podejście. Efektywność delegowania zależy od kontekstu i doświadczenia w korzystaniu z tej techniki. Działa ona najlepiej, jeśli jest stosowana w ścisłe określony sposób — czyli na podstawie standardowych wzorców.

Delegowanie wykorzystano w kilku wzorcach projektowych. Zależne od tej techniki są wzorce: Stan (s. 312), Strategia (s. 321) i Odwiedzający (s. 280). We wzorcu Stan obiekt deleguje żądania do reprezentującego jego obecny stan obiektu State. We wzorcu Strategia obiekt deleguje specyficzne żądanie do obiektu reprezentującego strategię obsługi tego żądania. Obiekt ma wtedy tylko jeden stan, ale może stosować wiele strategii dla różnych żądań. Celem stosowania obu tych wzorców jest modyfikacja działania obiektu przez zmianę delegatów, do których przekazywane są żądania. We wzorcu Odwiedzający operacje wykonywane na każdym elemencie struktury obiektowej zawsze są przekazywane do obiektu Visitor.

W innych wzorcach delegowanie wykorzystano w mniejszym stopniu. Wzorzec Mediator (s. 254) wprowadza obiekt pośredniczący do procesu komunikacji między innymi obiektami. Czasem w obiekcie Mediator operacje są wykonywane przez przekazanie ich do innych obiektów. W niektórych przypadkach obiekt ten przekazuje referencję do samego siebie, a tym samym stosuje delegowanie. We wzorcu Łańcuch zobowiązań (s. 244) żądania są obsługiwane przez przekazywanie ich od jednego obiektu do innego za pośrednictwem łańcucha obiektów.

Czasem te żądania obejmują referencję do pierwotnego obiektu (odbiorcy żądania). Wtedy to we wzorcu stosowane jest delegowanie. Wzorzec Most (s. 181) rozdziela abstrakcję od implementacji. Jeśli abstrakcja i określona implementacja są ściśle dopasowane, abstrakcja może po prostu oddelegować wykonanie operacji do danej implementacji.

Delegowanie to krańcowy przykład składania obiektów. Technika ta dowodzi, że w celu powtórnego wykorzystania rozwiązania zawsze można zastosować składanie obiektów zamiast dziedziczenia.

DZIEDZICZENIE A TYPY SPARAMETRYZOWANE

Inna (niespecyficzna dla programowania obiektowego) technika powtórnego wykorzystania funkcji to **typy sparametryzowane**, nazywane też **typami generycznymi** (języki Ada i Eiffel) albo **szablonami** (język C++) . Ta metoda pozwala zdefiniować typ bez określania wszystkich wykorzystywanych w nim typów. Nieokreślone typy w momencie ich użycia są podawane jako *parametry*. Na przykład w klasie `List` można podać jako parametr typ elementów przechowywanych w liście. Aby zadeklarować listę liczb całkowitych, należy podać typ „`integer`” jako parametr typu sparametryzowanego `List`. W celu zadeklarowania listy obiektów typu `String` należy przekazać jako parametr typ „`String`”. Implementacja wbudowana w język utworzy dostosowaną do każdego typu elementów wersję szablonu klasy `List`.

Typy sparametryzowane to trzeci (obok dziedziczenia klas i składania obiektów) sposób na łączenie zachowań w systemach obiektowych. Wiele projektów można zrealizować za pomocą dowolnej z tych trzech technik. Aby sparametryzować procedurę sortującą za pomocą operacji porównywania elementów, można tę operację zapisać jako:

1. Metodę zaimplementowaną w podklasach (zastosowanie Metody szablonowej, s. 264).
2. Zadanie obiektu przekazywanego do procedury sortującej (Strategia, s. 321).
3. Argument szablonu języka C++ lub typu generycznego języka Ada określający nazwę funkcji, którą należy wywołać w celu porównania elementów.

Miedzy tymi technikami występują ważne różnice. Składanie obiektów umożliwia zmianę zachowań w czasie wykonywania programu, jednak wymaga wywołań pośrednich i może okazać się mniej wydajne. Dziedziczenie pozwala podać domyślne implementacje metod i przesłonić je w podklasach. Typy sparametryzowane umożliwiają zmianę typów używanych w klasie. Jednak ani dziedziczenie, ani typy sparametryzowane nie obsługują wprowadzania zmian w czasie wykonywania programu. To, które podejście jest najlepsze, zależy od ograniczeń związanych z projektem i implementacją.

Żaden ze wzorców z tej książki nie dotyczy typów sparametryzowanych, choć czasem korzystamy z nich do dostosowania implementacji wzorców w języku C++ do specyficznych potrzeb. Typy sparametryzowane w ogóle nie są potrzebne w językach, w których nie występuje sprawdzanie typu w czasie kompilacji (działa tak na przykład język Smalltalk).

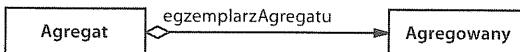
ZWIĄZKI MIĘDZY STRUKTURAMI CZASU WYKONYWANIA PROGRAMU I STRUKTURAMI CZASU KOMPILACJI

Struktura programu obiektowego w czasie jego wykonywania znacznie różni się od struktury kodu. Kod jest „zamrażany” w czasie komplikacji. Składa się z klas pozostających w trwałych relacjach dziedziczenia. Struktura programu w czasie jego wykonywania obejmuje szybko zmieniające się sieci komunikujących się ze sobą obiektów. Dwie wspomniane struktury są w dużym stopniu niezależne od siebie. Próba zrozumienia jednej na podstawie drugiej przypomina próbę określenia dynamiki ekosystemu na podstawie statycznej taksonomii roślin i zwierząt (i na odwrót).

Zastanówmy się nad rozróżnieniem na **agregowanie** i **znajomość** obiektów oraz odmiennym przejawianiem się tych mechanizmów w czasie komplikacji i wykonywania programu. Agregowanie oznacza, że jeden obiekt obejmuje inny lub jest za niego odpowiedzialny. Ogólnie mówimy, że obiekt *ma* inny obiekt lub jest jego *częścią*. Agregowanie powoduje, że czas istnienia agregowanego obiektu i jego właściciela jest identyczny.

Znajomość oznacza, że obiekt jedynie *wie* o istnieniu innego. Czasem znajomość jest nazywana „asocjacją” lub relacją „używania”. Obiekty pozostające w tej relacji mogą zażądać od siebie uruchomienia operacji, jednak nie odpowiadają za siebie. Znajomość jest słabszą relacją od agregacji i wskazuje na znacznie luźniejsze powiązanie między obiektami.

Na przedstawianych diagramach znajomość jest przedstawiona za pomocą zwykłej linii ze strzałką. Podobna linia z dodatkowym rombem oznacza agregację:



Łatwo jest pomylić ze sobą agregację ze znajomością, ponieważ relacje te często są implementowane w ten sam sposób. W języku Smalltalk wszystkie zmienne to referencje do innych obiektów. W tym języku agregacja i znajomość nie różnią się od siebie. W języku C++ agregację można zaimplementować przez zdefiniowanie zmiennych składowych będących rzeczywistymi egzemplarzami danej klasy, jednak częściej zmienne te definiuje się jako wskaźniki lub referencje do takich egzemplarzy. Do implementowania znajomości także wykorzystuje się wskaźniki i referencje.

Ostatecznie to, czy relacja jest znajomością czy agregacją, wyznacza bardziej jej przeznaczenie niż bezpośrednie mechanizmy języka. W strukturach czasu komplikacji rozróżnienie między tymi relacjami jest trudne do zauważenia, ale ma istotne znaczenie. Relacje agregacji są zwykle rzadsze i trwalsze od znajomości. Z kolei relacje znajomości są nawiązywane i odnawiane częściej — czasem trwają tylko przez czas wykonywania operacji. Znajomości są ponadto bardziej dynamiczne, co utrudnia ich dostrzeżenie w kodzie źródłowym.

Przy tak znacznych różnicach między strukturami czasu wykonywania programu i czasu komplikacji oczywiste jest, że na podstawie kodu nie można dowiedzieć się wszystkiego o działaniu systemu. Struktury czasu wykonania są wyznaczane w większym stopniu przez projektanta niż przez język. Relacje między obiektami i ich typami trzeba projektować z wielką ostrożnością, ponieważ określają one, jak dobre (lub zle) będą struktury czasu wykonania.

W wielu wzorcach projektowych (przede wszystkim tych z zasięgu obiektów) rozróżnienie na struktury czasu komplikacji i czasu wykonywania programu jest uwzględnione bezpośrednio. Wzorzec Kompozyt (s. 170) i Dekorator (s. 152) są wyjątkowo przydatne do tworzenia złożonych struktur czasu wykonywania programu. Wzorzec Obserwator (s. 269) dotyczy struktur czasu wykonywania programu, które często trudno jest zrozumieć bez znajomości tego wzorca. Wzorzec Łańcuch zobowiązań (s. 244) także prowadzi do powstawania wzorców komunikacji, których nie pozwala dostrzec dziedziczenie. Ogólnie bez zrozumienia określonych wzorców trudno jest wykryć w kodzie struktury czasu wykonywania programu.

PROJEKTOWANIE POD KĄTEM ZMIAN

Kluczem do zmaksymalizowania powtórnego wykorzystania rozwiązania jest przewidywanie nowych potrzeb i zmian w istniejących wymogach oraz projektowanie systemów tak, aby mogły w odpowiedni sposób ewoluować.

Aby zaprojektować system odporny na zmiany, trzeba uwzględnić, jakie modyfikacje mogą być konieczne w trakcie jego stosowania. Projekt, w którym nie uwzględniono zmian, może w przyszłości wymagać poważnych przekształceń. Poprawki mogą obejmować przygotowanie nowej definicji lub implementacji klasy, zmodyfikowanie klienta i przeprowadzenie ponownych testów. Przekształcanie wpływa na wiele części systemu, a nieprzewidziane zmiany zawsze są kosztowne.

Wzorce projektowe pomagają uniknąć takich sytuacji, ponieważ gwarantują, że system można zmodyfikować na określone sposoby. Każdy wzorzec projektowy pozwala zmieniać pewien aspekt struktury systemu niezależnie od innych, dzięki czemu system staje się odporniejszy na modyfikacje danego rodzaju.

Oto kilka typowych przyczyn przekształcania projektów oraz zestaw wzorców projektowych rozwiązujących poszczególne problemy:

1. *Tworzenie obiektu przez bezpośrednie podanie klasy.* Wskazanie nazwy klasy przy tworzeniu obiektu zmusza do zastosowania konkretnej implementacji, a nie określonego interfejsu. Może to utrudnić późniejsze wprowadzanie zmian. Aby tego uniknąć, należy tworzyć obiekty pośrednio.

Wzorce projektowe: Fabryka abstrakcyjna (s. 101), Metoda wytwórcza (s. 110), Prototyp (s. 120).

2. *Zależność od specyficznych operacji.* Jeśli określisz specyficzną operację, ograniczysz się do jednego sposobu obsługi żądań. Unikanie zapisanych na stałe operacji ułatwia zmianę sposobu obsługi żądań zarówno w czasie komplikacji, jak i w czasie wykonywania programu.

Wzorce projektowe: Łańcuch zobowiązań (s. 244), Polecanie (s. 302).

3. *Zależność od platformy sprzętowej lub programowej.* W poszczególnych platformach sprzętowych i programowych znajdują się różne zewnętrzne interfejsy systemu operacyjnego i interfejsy programowania aplikacji (ang. *Application Programming Interface* — API). Oprogramowanie zależne od konkretnej platformy trudniej jest przenieść na inne platformy. Czasem trudno jest nawet aktualizować kod na potrzeby macierzystej platformy. Dlatego ważne jest, aby projektować system pod kątem minimalizacji zależności od platformy.

Wzorce projektowe: Fabryka abstrakcyjna (s. 101), Most (s. 181).

4. *Zależność od reprezentacji lub implementacji obiektu.* Klienci znające reprezentację, sposób przechowywania, lokalizację i implementację obiektu mogą wymagać zmian po jego zmodyfikowaniu. Ukrywanie wymienionych informacji przed klientami pomaga zatrzymać kasadowe rozprzestrzenianie się zmian.

Wzorce projektowe: Fabryka abstrakcyjna (s. 101), Most (s. 181), Pamiątka (s. 294), Pełnomocnik (s. 191).

5. *Zależność od algorytmów.* Algorytmy w czasie programowania i powtórnego korzystania z kodu często są rozbudowywane, optymalizowane oraz zastępowane. Obiekty zależne od algorytmów także trzeba wtedy zmodyfikować. Dlatego algorytmy, które mogą ulec zmianie, należy izolować.

Wzorce projektowe: Budowniczy (s. 92), Iterator (s. 230), Strategia (s. 321), Metoda szablonowa (s. 264), Odwiedzający (s. 280).

6. *Ścisłe powiązanie.* Jeśli klasy są ścisłe powiązane, trudno jest je powtórnie wykorzystać niezależnie od pozostałych z uwagi na zależności między nimi. Ścisłe powiązanie prowadzi do powstawania monolitycznych systemów, w których nie można zmienić lub usunąć klas bez zrozumienia i zmodyfikowania wielu innych klas. Takie systemy są bardzo zawiłe — trudno jest je poznawać, przenosić i pielęgnować.

Luźne powiązanie zwiększa prawdopodobieństwo powtórnego wykorzystania pojedynczych klas oraz ułatwia poznawanie, przenoszenie, modyfikowanie i rozbudowywanie systemów. Dlatego we wzorcach projektowych stosowane są techniki w rodzaju powiązania abstrakcyjnego i podziału na warstwy, wspomagające tworzenie luźno powiązanych systemów.

Wzorce projektowe: Fabryka abstrakcyjna (s. 101), Most (s. 181), Łańcuch zobowiązań (s. 244), Polecanie (s. 302), Fasada (s. 161), Mediator (s. 254), Obserwator (s. 269).

7. *Rozszerzanie możliwości oprogramowania przez tworzenie podklas.* Dostosowywanie obiektów przez tworzenie podklas często nie jest proste. Z każdą nową klasą związane są określone koszty implementacyjne (inicjowanie, finalizowanie itd.). Zdefiniowanie podklasy wymaga też dogłębniego zrozumienia działania klasy nadzędnej. Na przykład przesłonięcie jednej operacji zmusza czasem do przesłonięcia także innych, a przesłonięta operacja może wymagać wywołania operacji odziedziczonej. Ponadto tworzenie podklas prowadzi nieraz do znacznego wzrostu ich liczby, jeśli dodawanie nowych podklas jest konieczne nawet przy wprowadzaniu prostych zmian.

Składanie klas, a w szczególności delegowanie, to bardziej elastyczna od dziedziczenia metoda łączenia zachowań. Nowe funkcje można dodać do aplikacji przez złożenie istniejących obiektów w inny sposób, a nie przez definiowanie nowych podklas na podstawie istniejących klas. Z drugiej strony intensywne stosowanie składania obiektów może utrudniać zrozumienie projektu. Wiele wzorców projektowych prowadzi do tworzenia projektów, w których niestandardowe funkcje można dodać przez zdefiniowanie tylko jednej podklasy i złożenie jej egzemplarzy z egzemplarzami istniejących klas.

Wzorce projektowe: Most (s. 181), Łańcuch zobowiązań (s. 244), Kompozyt (s. 170), Dekorator (s. 152), Obserwator (s. 269), Strategia (s. 321).

8. *Brak możliwości wygodnego modyfikowania klas.* Czasem trzeba zmodyfikować klasę, której nie można zmienić w wygodny sposób. Możliwe, że potrzebujesz kodu źródłowego, a nie masz do niego dostępu (może się to zdarzyć przy korzystaniu z komercyjnej biblioteki klas). Możliwe, że każda zmiana będzie wymagać zmodyfikowania wielu istniejących podklas. Wzorce projektowe umożliwiają wprowadzanie zmian w klasach także w takich warunkach.

Wzorce projektowe: Adapter (s. 141), Dekorator (s. 152), Odwiedzający (s. 280).

Te przykłady ilustrują elastyczność, jaką wzorce projektowe pomagają wbudować w oprogramowanie. Jej znaczenie zależy od rodzaju rozwijanego programu. Przyjrzyjmy się roli wzorców projektowych w tworzeniu oprogramowania z trzech ogólnych kategorii: aplikacji, pakietów narzędziowych i platform.

APLIKACJE

Przy tworzeniu aplikacji, takich jak edytor dokumentów lub arkusz kalkulacyjny, ważne są: możliwość wewnętrznego powtórnego wykorzystania, łatwość konserwacji i rozszerzalność kodu. Możliwość wewnętrznego powtórnego wykorzystania rozwiązania oznacza, że będziesz musiał projektować i implementować tylko to, co konieczne. Wzorce projektowe zmniejszające zależności mogą poprawić poziom wewnętrznego wykorzystania. Luźne powiązanie zwiększa prawdopodobieństwo tego, że klasa obiektu będzie współdziałała z innymi klasami. Na przykład wyeliminowanie zależności między operacjami przez ich odizolowanie i zakapsułkowanie ułatwia powtórne wykorzystanie operacji w różnych kontekstach. Podobne skutki ma usunięcie zależności związanych z algorytmami i reprezentacjami.

Ponadto wzorce projektowe ułatwiają konserwację kodu, jeśli programista zastosuje je do podziału systemu na warstwy i ograniczenia zależności od platformy. Wzorce wspomagają też rozszerzanie oprogramowania, ponieważ pokazują, jak rozbudowywać hierarchie klas i jak wykorzystać składanie obiektów. Mniejsze powiązanie pozytywnie wpływa także na rozszerzalność. Rozbudowywanie pojedynczych klas jest łatwiejsze, jeśli dana klasa nie zależy od wielu innych.

PAKIETY NARZĘDZIOWE

Aplikacje często obejmują klasy z jednego lub kilku pakietów narzędziowych, czyli bibliotek gotowych klas. Pakiet narzędziowy to zestaw powiązanych i nadających się do wielokrotnego wykorzystania klas. Udostępniają one przydatne funkcje ogólnego użytku. Przykładem pakietu narzędziowego jest zestaw klas kolekcji, obsługujących listy, tablice asocjacyjne, stosy itd. Biblioteka I/O stream języka C++ to następny przykład takiego zestawu. Pakiety narzędziowe nie wymuszają stosowania w aplikacji określonego projektu, a jedynie udostępniają funkcje pomagające wykonywać programowi zadania. Pozwalają programistom uniknąć ponownego pisania kodu standardowych funkcji. W pakietach narzędziowych naciśnięty jest na powtórne wykorzystanie kodu. Zestawy te to obiekty odpowiedni dla bibliotek procedur.

Niektórzy sądzą, że projektowanie pakietów narzędziowych jest trudniejsze od projektowania aplikacji, ponieważ pakiety muszą działać w wielu programach, aby były przydatne. Ponadto autor takiego pakietu nie wie, w jakich aplikacjach kod będzie wykorzystywany i jakie są ich specyficzne wymagania. Dlatego tym ważniejsze jest uniknięcie założeń i zależności, które mogłyby ograniczyć elastyczność danego pakietu narzędziowego, a tym samym jego zastosowania i efektywność.

PLATFORMY

Platforma (ang. *framework*) to zbiór współdziałających klas, które składają się na projekt wielokrotnego użytku dla oprogramowania określonego rodzaju [Deu89, JF88]. Platforma może służyć na przykład do tworzenia różnych edytorów graficznych — do rysunku artystycznego, komponowania muzyki i wspomaganego komputerowo projektowania systemów mechanicznych [VL90, Joh92]. Inna platforma może pomagać w tworzeniu kompilatorów dla różnych języków programowania i docelowych maszyn [JML92]. Jeszcze następna grupa może służyć do budowania aplikacji do modelowania zjawisk finansowych [BE93]. Platformy są dostosowywane do konkretnych aplikacji przez tworzenie specyficznych podklas na podstawie klas abstrakcyjnych z platformy.

Platforma wyznacza architekturę aplikacji. Określa ogólną strukturę programu, podział na klasy i obiekty, kluczowe zadania tych jednostek, współdziałanie między klasami i obiektami oraz przepływ sterowania. Te aspekty projektu są zdefiniowane w platformie, dzięki czemu projektant lub programista aplikacji może skoncentrować się na specyficznych aspektach programu. Platformy są oparte na decyzjach projektowych standardowych dla aplikacji z danej dziedziny. Dlatego w platformach nacisk położony jest na *powtórne wykorzystanie projektu*, choć zwykle obejmują one konkretne podklasy, które można od razu zastosować.

Powtórne wykorzystanie projektu prowadzi do odwrócenia zależności między aplikacją i oprogramowaniem będącym jej podstawą. Jeśli programista używa pakietu narzędziowego (lub standardowej biblioteki procedur), pisze główną część aplikacji i wywołuje kod, który chce powtórnie wykorzystać. Jeżeli używa platformy, powtórnie korzysta z części głównej i rozwija wywoływany *przez nią* kod. Trzeba wtedy przygotować operacje o określonych nazwach i sposobach wywoływania, jednak podejmowanie wielu decyzji projektowych nie jest konieczne.

Nie tylko pozwala to szybciej rozwijać aplikacje, ale też gwarantuje, że będą one miały podobną strukturę. Łatwiej jest je konserwować i wyglądają one spójniej z perspektywy użytkowników. Z drugiej strony programista traci nieco swobody twórczej, ponieważ wiele decyzji projektowych jest już podjętych za niego.

Projektowanie aplikacji sprawia problemy, trudniejsze jest opracowanie pakietu narzędziowego, ale największe kłopoty związane są z projektowaniem platform. Ich projektanci zakładają, że jedna architektura będzie odpowiednia dla wszystkich aplikacji z danej dziedziny. Każda istotna zmiana w projekcie platformy znacznie zmniejsza korzyści płynące z jej stosowania, ponieważ głównym wkładem platformy w rozwijanie aplikacji jest architektura. Dlatego konieczne jest projektowanie platformy tak, aby była jak najbardziej elastyczna i rozszerzalna.

Ponadto z uwagi na wysoką zależność projektów aplikacji od platformy programy są wyjątkowo wrażliwe na zmiany w interfejsach platform. Ewolucja platformy wymaga wprowadzenia modyfikacji w aplikacji. Dlatego tak ważne jest zachowanie luźnego powiązania. Jeśli projektant go nie zapewni, nawet niewielka zmiana w platformie będzie miała poważne skutki.

Omówione tu kwestie są kluczowe w projekcie platformy. Projektant, który uwzględnia je i zastosuje właściwe wzorce projektowe, prawdopodobnie zapewni znacznie wyższy poziom powtórnego wykorzystania projektu i kodu niż osoby pomijające opisane problemy. Dojrzałe platformy zwykle obejmują kilka wzorców projektowych. Pomagają one sprawić, że architektura platformy będzie odpowiednia dla wielu różnych aplikacji bez konieczności jej przekształcania.

Dodatkowa korzyść związana jest z dokumentowaniem platformy za pomocą zastosowanych w niej wzorców projektowych [BJ94]. Osoby znające wzorce szybciej zrozumieją taką platformę. Jednak także pozostały ludzie odniosą korzyść ze struktury nadawanej przez wzorce dokumentacji platformy. Wzbogacanie dokumentacji jest ważne w oprogramowaniu każdego rodzaju, jednak w przypadku platform ma to szczególne znaczenie. Uczenie się platform jest często trudne, ale konieczne, aby można z nich z pożytkiem korzystać. Choć wzorce projektowe nie sprawiają, że proces nauki jest zupełnie prosty, ułatwiają poznawanie platformy, ponieważ w bardziej bezpośredni sposób ilustrują kluczowe elementy jej projektu.

Ponieważ wzorce i platformy mają pewne cechy wspólne, ludzie często się zastanawiają, w jakich aspektach (a nawet, czy w ogóle) różnią się one od siebie. Odmienność ta przejawia się na trzy podstawowe sposoby:

1. *Wzorce projektowe są bardziej abstrakcyjne od platform.* Platformy można wyrazić w kodzie, natomiast to samo można zrobić tylko z przykładami wzorców. Siłą platformy jest to, że można ją zapisać w językach programowania i nie tylko przestudiować, ale też bezpośrednio uruchomić i ponownie wykorzystać. Z drugiej strony wzorce projektowe opisane w tej książce trzeba przy każdym użyciu zaimplementować. Wzorce ponadto określają przeznaczenie, korzyści i koszty oraz konsekwencje zastosowania danego projektu.
2. *Wzorce projektowe są mniejszymi elementami architektonicznymi od platform.* Typowa platforma obejmuje kilka wzorców projektowych, natomiast odwrotne twierdzenie nigdy nie jest prawdziwe.
3. *Wzorce projektowe są mniej wyspecjalizowane od platform.* Platformy zawsze powiązane są z określoną dziedziną. Platformę do tworzenia edytorów graficznych można wykorzystać w symulacji działania fabryki, ale trudno jest ją pomylić z platformą do tworzenia symulacji. Z drugiej strony wzorce projektowe z katalogu nadają się do zastosowania w niemal dowolnej aplikacji. Choć oczywiście można utworzyć bardziej wyspecjalizowane wzorce od tych opisanych w niniejszej książce (na przykład wzorce na potrzeby systemów rozproszonych lub programowania równoległego), nawet one nie wyznaczają architektury aplikacji w takim stopniu, jak robią to platformy.

Platformy stają się coraz powszechniejsze i ważniejsze. Są one rozwiązaniem, któreumożliwia maksymalne powtórne wykorzystanie systemów obiektowych. Większe aplikacje obiektowe będą ostatecznie składać się z warstw współdziałających ze sobą platform. Większa część projektu i kodu programu będzie pochodzić z zastosowanej platformy lub zostanie na niej oparta.

1.7. JAK WYBRAĆ WZORZEC PROJEKTOWY?

Ponieważ katalog obejmuje ponad 20 wzorców projektowych, znalezienie rozwiązania konkretnego problemu projektowego może sprawiać trudność, zwłaszcza jeśli katalog jest dla Ciebie czymś nowym i nieznanym. Oto kilka sposobów na wyszukiwanie odpowiednich wzorców:

- ▶ *Pomyśl o tym, jak wzorce rozwiązują problemy projektowe.* W podrozdziale 1.6 analizujemy, w jaki sposób wzorce pomagają znaleźć właściwe obiekty oraz określić ich szczegółowość i interfejs. Przedstawiamy też kilka innych aspektów rozwiązywania problemów projektowych za pomocą wzorców. Zaglądanie do tych analiz może pomóc w znalezieniu odpowiedniego wzorca.
- ▶ *Przeglądaj punkty „Przeznaczenie”.* W podrozdziale 1.4 (s. 22) przytoczyliśmy treść punktów „Przeznaczenie” dotyczących każdego wzorca z katalogu. Zapoznaj się z celem stosowania poszczególnych wzorców, aby znaleźć rozwiązania adekwatne do problemu. Jeśli chcesz zawęzić poszukiwania, wykorzystaj klasyfikację z tabeli 1.1 (s. 24).
- ▶ *Przeanalizuj relacje między wzorcami.* Rysunek 1.1 (s. 25) przedstawia w formie graficznej relacje między wzorcami projektowymi. Przeanalizowanie tych zależności może pomóc w wyborze właściwego wzorca lub ich grupy.
- ▶ *Przeanalizuj wzorce o podobnym przeznaczeniu.* Katalog obejmuje trzy rozdziały poświęcone: wzorcom konstrukcyjnym, wzorcom strukturalnym i wzorcom operacyjnym. Każdy rozdział zaczyna się od wprowadzenia na temat wzorców, a kończy podrozdziałem porównującym je. Te punkty pozwolą Ci zrozumieć podobieństwa i różnice między wzorcami o podobnym przeznaczeniu.
- ▶ *Zbadaj przyczyny przekształcania projektów.* Prześledź powody przekształcania projektów (ich opis zaczyna się od s. 37), aby ustalić, czy problem jest z nimi związany. Następnie przyjrzyj się wzorcem pomagającym wyeliminować określone przyczyny.
- ▶ *Zastanów się, co powinno być zmienne w projekcie.* To podejście jest przeciwnieństwem koncentrowania się na przyczynach przekształcania projektu. Zamiast się zastanawiać, co może wymuszać zmiany, rozważ, jakie elementy chcesz móc zmieniać bez konieczności modyfikowania projektu. Najważniejsze jest przy tym *kapsułkowanie zmiennych elementów*. Motyw ten powtarza się w wielu wzorcach projektowych. Tabela 1.2 przedstawia aspekty projektów, które dzięki wzorcem można modyfikować niezależnie od innych, a tym samym uniknąć konieczności przekształcania projektu.

TABELA 1.2. Aspekty projektów, które można zmieniać dzięki wzorcom projektowym

Rodzaj	Wzorzec projektowy	Aspekty, które można zmieniać
Konstrukcyjne	Fabryka abstrakcyjna (s. 101)	Rodziny obiektów-produktów
	Budowniczy (s. 92)	Sposób tworzenia obiektu złożonego
	Metoda wytwarzająca (s. 110)	Podklasa tworzonego obiektu
	Prototyp (s. 120)	Klasa tworzonego obiektu
	Singleton (s. 130)	Jedyny egzemplarz klasy
Strukturalne	Adapter (s. 141)	Interfejs obiektu
	Most (s. 181)	Implementacja obiektu
	Kompozyt (s. 170)	Struktura i skład obiektu
	Dekorator (s. 152)	Zadania obiektu (bez tworzenia podklaśc)
	Fasada (s. 161)	Interfejs podsystemu
	Pytak (s. 201)	Koszty przechowywania obiektów
	Pełnomocnik (s. 191)	Sposób dostępu do obiektu i jego lokalizacja
Operacyjne	Łańcuch zobowiązań (s. 244)	Obiekt potrafiący obsłużyć żądanie
	Polecenie (s. 302)	Warunki i sposób obsługi żądania
	Interpreter (s. 217)	Gramatyka i interpretacja języka
	Iterator (s. 230)	Sposób dostępu do elementów i przechodzenia po nich
	Mediator (s. 254)	Jak i które obiekty wchodzą ze sobą w interakcję
	Pamiątka (s. 294)	Które prywatne informacje są przechowywane poza obiektem i kiedy
	Obserwator (s. 269)	Liczba obiektów zależnych od innego obiektu; sposób aktualizowania obiektów zależnych
	Stan (s. 312)	Stany obiektu
	Strategia (s. 321)	Algorytm
	Metoda szablonowa (s. 264)	Kroki algorytmu
	Odwiedzający (s. 280)	Operacje, które można zastosować do obiektów bez zmiany ich klas

1.8. JAK STOSOWAĆ WZORCE PROJEKTOWE?

Jak zastosować wzorzec projektowy po jego wybraniu? Oto opisana krok po kroku metoda efektywnego korzystania ze wzorców projektowych:

1. Przeczytaj raz opis wzorca w ramach jego przeglądu. Zwróć szczególną uwagę na punkty „Zastosowania” i „Konsekwencje”, aby się upewnić, że wzorzec jest odpowiedni dla problemu.
2. Cofnij się i przeanalizuj punkty „Struktura”, „Elementy” i „Współdziałanie”. Upewnij się, że rozumiesz działanie klas i obiektów ze wzorca oraz relacje między nimi.

3. *Zajrzyj do punktu „Przykładowy kod”, aby zapoznać się z konkretnym przykładem zastosowania wzorca w kodzie.* Analiza kodu pomoże Ci się dowiedzieć, jak zaimplementować wzorzec.
4. *Wybierz dla elementów wzorca nazwy mające odpowiednie znaczenie w kontekście aplikacji.* Nazwy elementów wzorca projektowego są zwykle zbyt abstrakcyjne, aby bezpośrednio stosować je w aplikacji. Jednak przydatne jest włączenie nazw elementów wzorca w nazwy używane w programie. Dzięki temu łatwiej jest zauważać w kodzie, że zastosowano dany wzorzec. Na przykład jeśli korzystasz ze wzorca Strategia do tworzenia algorytmu określającego układ tekstu, możesz nazwać klasy SimpleLayoutStrategy lub TeXLayoutStrategy.
5. *Zdefiniuj klasy.* Zadeklaruj ich interfejsy, określ relacje dziedziczenia oraz zdefiniuj zmienne egzemplarza, reprezentujące dane i referencje do obiektów. Ustal istniejące klasy aplikacji, na które wpłynie zastosowanie wzorca, i odpowiednio je zmodyfikuj.
6. *Ustal specyficzne dla aplikacji nazwy operacji używanych we wzorcu.* Także tu nazwy zwykle zależą od programu. Jako wskazówkę wykorzystaj zadania i sposób współdziałania danej operacji. Zachowaj też spójność w konwencjach nazewniczych. Możesz na przykład stale używać przedrostka „Create” do określania metod wytwórczych.
7. *Zaimplementuj operacje, aby zrealizować zadania i zapewnić współdziałanie opisane we wzorcu.* Punkt „Implementacja” obejmuje wskazówki dotyczące implementacji. Pomocne mogą być także przykłady z punktu „Przykładowy kod”.

To tylko wytyczne, które pomogą Ci zacząć. Z czasem opracujesz własny sposób korzystania ze wzorców projektowych.

Żadna analiza na temat stosowania wzorców projektowych nie byłaby kompletna bez kilku słów na temat tego, jak *nie* należy z nich korzystać. Wzorców nie należy używać bez zastanowienia. Często pozwalają one uzyskać elastyczność i możliwość wprowadzania zmian przez dodanie nowych poziomów pośredniości. Może to skomplikować projekt i (lub) pogorszyć wydajność. Wzorce projektowe należy stosować tylko wtedy, kiedy większa elastyczność jest naprawdę potrzebna. Przy ocenianiu korzyści i kosztów związanych ze wzorcami najbardziej pomocne są punkty „Konsekwencje”.

Rozdział 2.

Studium przypadku — projektowanie edytora dokumentów

W tym rozdziale przedstawiamy studium przypadku na przykładzie projektowania edytora dokumentów typu WYSIWYG (ang. *What You See Is What You Get*, czyli otrzymujesz to, co widzisz). Edytor ten nosi nazwę Lexi¹. Pokażemy, jak wzorce projektowe umożliwiają rozwiązanie problemów projektowych związanych z programem Lexi i podobnymi aplikacjami. W czasie lektury rozdziału zdobędziesz doświadczenie w korzystaniu z ośmiu wzorców, poznając je na przykładach.

Rysunek 2.1 przedstawia interfejs użytkownika edytora Lexi. Reprezentacja dokumentu w trybie WYSIWYG zajmuje duży prostokątny obszar w środkowej części aplikacji. W dokumencie można swobodnie łączyć tekst i grafikę, sformatowane na różne sposoby. Wokół dokumentu znajdują się standardowe menu rozwijane i paski przewijania, a także zbiór ikon oznaczających strony, umożliwiających przeskakiwanie do określonej strony.

2.1. PROBLEMY PROJEKTOWE

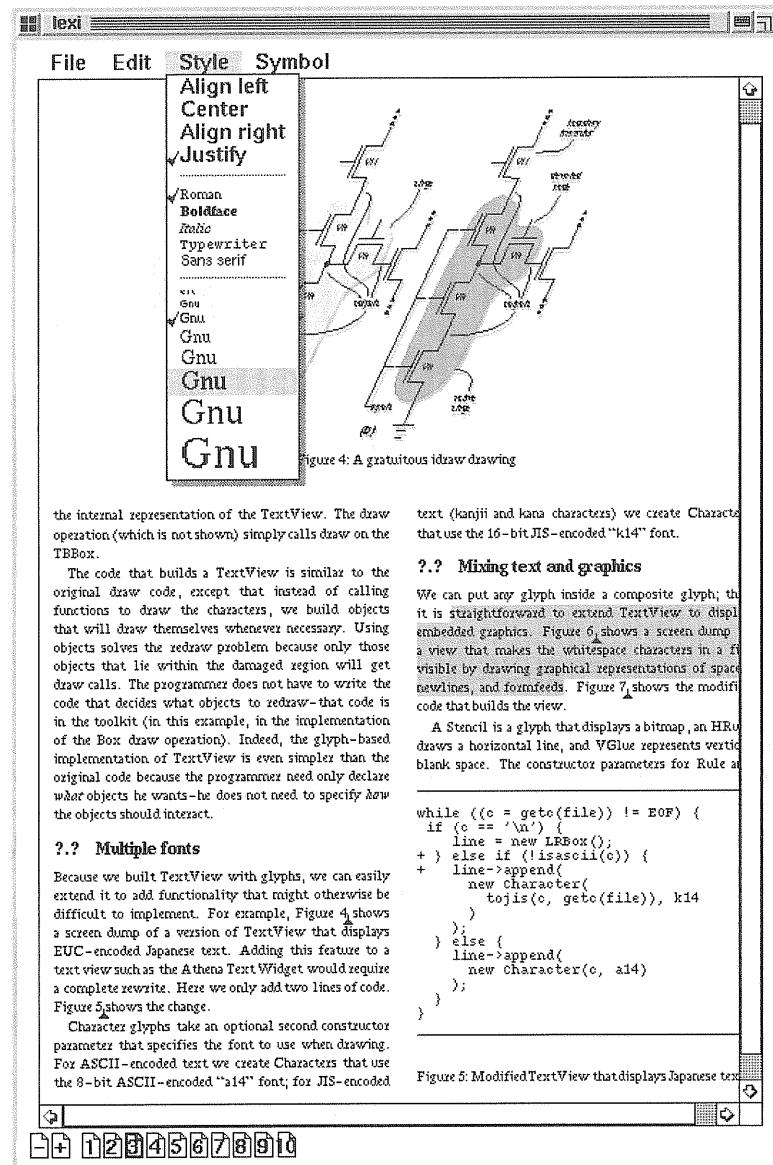
Przeanalizujemy siedem problemów związanych z projektem edytora Lexi:

1. *Struktura dokumentu*. Wybór wewnętrznej reprezentacji dokumentu wpływa na niemal każdy aspekt projektu edytora. Z tej reprezentacji trzeba będzie korzystać przy edytowaniu, formatowaniu, wyświetlaniu i analizowaniu tekstu. Sposób uporządkowania informacji wpływa na projekt pozostałych części aplikacji.

¹ Projekt edytora Lexi jest oparty na aplikacji Doc — edytorze tekstu opracowanym przez Caldera [CL92].

RYSUNEK 2.1.

Interfejs użytkownika edytora Lexi



2. *Formatowanie.* W jaki sposób edytor Lexi porządkuje tekst i grafikę w wierszach oraz kolumnach? Które obiekty odpowiadają za obsługę różnych zasad formatowania? W jaki sposób reguły te wchodzą w interakcje z wewnętrzną reprezentacją dokumentu?
 3. *Ozdabianie interfejsu użytkownika.* Interfejs użytkownika edytora Lexi obejmuje paski przewijania, ramki i cienie, które ozdabiają interfejs dokumentu w trybie WYSIWYG. Te ozdobniki prawdopodobnie będą modyfikowane w czasie ewoluuowania interfejsu użytkownika. Dlatego ważne jest, aby można było dodawać i usuwać bez wpływu na inne elementy aplikacji.

4. *Obsługa wielu standardów wyglądu i działania.* Edytor Lexi powinien łatwo dostosowywać się do różnych standardów wyglądu i działania (takich jak Motif i Presentation Manager) bez konieczności wprowadzania istotnych zmian.
5. *Obsługa wielu systemów okienkowych.* Odmienne standardy wyglądu i działania są zwykle stosowane w różnych systemach okienkowych. Projekt edytora Lexi powinien być tak niezależny od systemu, jak to możliwe.
6. *Operacje wykonywane przez użytkowników.* Użytkownicy kontrolują edytor Lexi za pomocą różnych interfejsów użytkownika — między innymi przycisków i menu rozwijanych. Funkcje związane z tymi interfejsami są rozproszone w wielu obiektach aplikacji. Trudność polega na zapewnieniu jednolitego mechanizmu zapewniającego dostęp do tych funkcji i wycofującego skutki ich zastosowania.
7. *Sprawdzanie pisowni i podziału słów.* W jaki sposób Lexi obsługuje operacje analityczne, takie jak wykrywanie błędów w pisowni i określanie miejsc podziału słów? Jak można zminimalizować liczbę klas, które trzeba będzie zmodyfikować w celu dodania nowych operacji analitycznych?

Te problemy projektowe omawiamy w następnych punktach. Każda trudność jest związana z określonymi celami i ograniczeniami w ich osiąganiu. Przed zaproponowaniem rozwiązania szczegółowo wyjaśniamy owe cele i ograniczenia. Problem i jego rozwiązanie ilustrują przy najmniej jeden wzorzec projektowy. Analiza poszczególnych trudności kończy się krótkim wprowadzeniem do odpowiednich wzorców.

2.2. STRUKTURA DOKUMENTU

Dokument to w swej istocie tylko układ podstawowych elementów graficznych, takich jak znaki, linie, wielokąty i inne kształty. Te jednostki ujmują wszystkie informacje o dokumencie. Jednak autor często postrzega elementy nie tylko jako jednostki graficzne, ale też w ramach fizycznej struktury dokumentu — jako wiersze, kolumny, tabele i inne struktury podrzędne². Z kolei te struktury mają własne struktury podrzędne itd.

Interfejs edytora Lexi powinien umożliwiać użytkownikom bezpośrednie manipulowanie strukturami podrzędnymi. Na przykład możliwe powinno być traktowanie diagramu jako jednostki, a nie kolekcji poszczególnych prostych elementów graficznych. Użytkownik powinien móc wskazywać tabelę jako całość, a nie jako nieustrukturyzowany zbiór elementów tekstowych i graficznych. Dzięki temu interfejs będzie prosty i intuicyjny. Aby implementacja edytora Lexi miała podobne cechy, zastosujemy wewnętrzną reprezentację odpowiadającą fizycznej strukturze dokumentu.

² Autorzy często postrzegają dokument także w kategoriach jego struktury *logicznej*, czyli jako zdania, akapity, punkty, podrozdziały i rozdziały. Aby nie komplikować przykładu, w wewnętrznej reprezentacji nie będziemy bezpośrednio zapisywać informacji o strukturze logicznej. Jednak opisywane rozwiązanie również dobrze nadaje się do reprezentowania takich danych.

Wewnętrzna reprezentacja powinna w szczególności mieć następujące cechy:

- ▶ Przechowywać fizyczną strukturę dokumentu, czyli układ tekstu i grafiki w wierszach, kolumnach, tabelach itd.
- ▶ Generować i przedstawiać dokument w postaci wizualnej.
- ▶ Odwzorowywać pozycję na ekranie na elementy z wewnętrznej reprezentacji dokumentu. Umożliwia to edytoriowi Lexi ustalenie, co interesuje użytkownika, kiedy wskazuje dany fragment reprezentacji wizualnej.

Oprócz tych celów istnieją też pewne ograniczenia. Po pierwsze, tekst i grafikę należy traktować w jednolity sposób. Interfejs aplikacji umożliwia użytkownikom swobodne zagnieżdżanie tekstu w grafice i na odwrót. Należy unikać traktowania grafiki jako specjalnego rodzaju tekstu lub tekstu jako specyficznego typu grafiki. Inne rozwiązanie będzie wymagało zastosowania nadmiarowych mechanizmów formatowania i manipulowania. Zarówno tekst, jak i grafikę powinien obsługiwać jeden zestaw mechanizmów.

Po drugie, w implementacji nie należy wprowadzać w wewnętrznej reprezentacji dokumentu rozróżnienia na pojedyncze elementy i ich grupy. Edytor Lexi powinien traktować elementy proste i złożone w taki sam sposób, co umożliwia tworzenie dowolnie skomplikowanych dokumentów. Dziesiątym elementem w piątym wierszu drugiej kolumny może być na przykład znak lub złożony diagram z wieloma elementami podrzędnymi. Jeśli wiadomo, że element potrafi wyświetlić swoją reprezentację i określić swoje wymiary, jego złożoność nie powinna mieć wpływu na to, w jakiej postaci i w którym miejscu strony się pojawi.

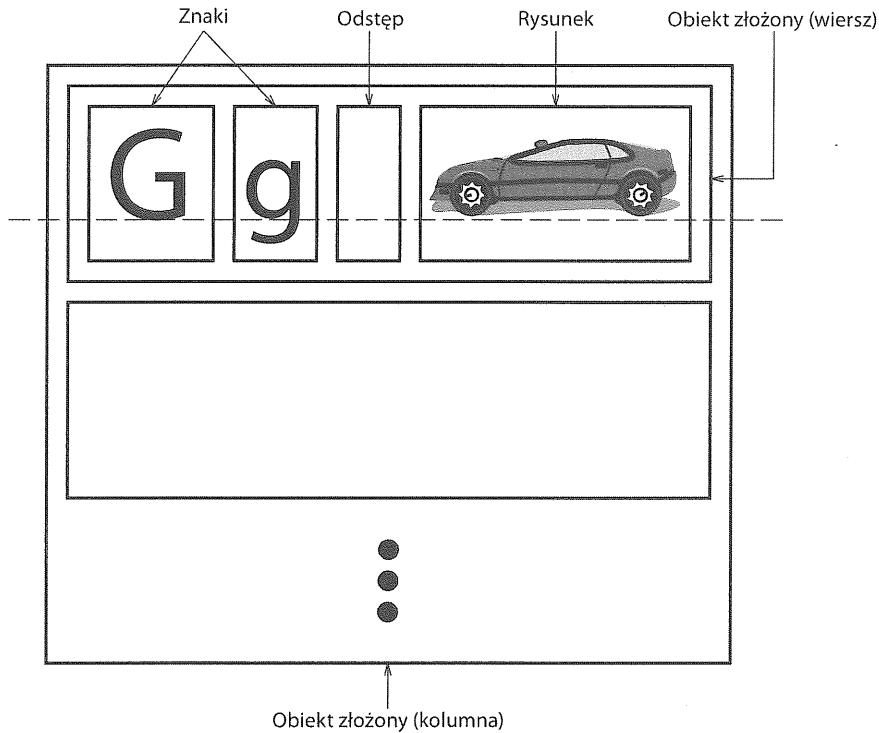
Sprzeczna z drugim ograniczeniem jest potrzeba analizowania tekstu pod kątem błędów w pisowni i ewentualnych miejsc podziału słów. Często nie ma znaczenia, czy dany element wiersza jest obiektem prostym czy złożonym. Jednak czasem proces analizy zależy od sprawdzanego obiektu. Na przykład nie ma większego sensu sprawdzanie pisowni wielokąta lub dzielenie go. Przy projektowaniu wewnętrznej reprezentacji należy uwzględnić to, a także inne potencjalnie sprzeczne ograniczenia.

SKŁADANIE REKURENCYJNE

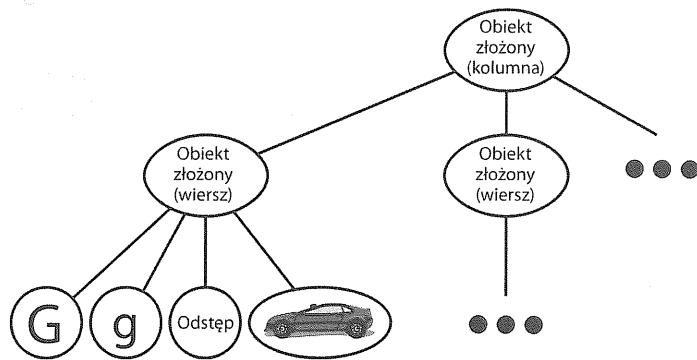
Standardowy sposób przedstawiania informacji o strukturze hierarchicznej oparty jest na technice **składania rekurencyjnego**. Polega ona na tworzeniu coraz bardziej złożonych elementów z prostszych jednostek. Rekurencyjne składanie umożliwia zbudowanie dokumentu z prostych elementów graficznych. W pierwszym kroku można uporządkować zbiór znaków i symboli graficznych od lewej do prawej, aby utworzyć wiersz w dokumencie. Następnie można uporządkować wiele wierszy w kolumnę, zbiór kolumn w stronę itd. (rysunek 2.2).

Tę fizyczną strukturę można przedstawić przez utworzenie odrębnego obiektu na potrzeby każdego ważnego elementu. Istotne są tu nie tylko widoczne jednostki, takie jak znaki i grafika, ale też niewidoczne elementy strukturalne — wiersze i kolumny. Efektem jest struktura obiektów przedstawiona na rysunku 2.3.

RYSUNEK 2.2.
Składanie rekurencyjne tekstu i grafiki



RYSUNEK 2.3.
Struktura obiektów w składaniu rekurencyjnym tekstu i grafiki



Użycie odrębnego obiektu dla każdego znaku i elementu graficznego w dokumencie zwiększa elastyczność na najniższym poziomie projektu edytora Lexi. Dzięki temu można traktować tekst i grafikę w jednolity sposób pod względem wyświetlania, formatowania i zagnieżdżania. Można też wzbogacić edytor Lexi o obsługę nowych zestawów znaków bez naruszania innych funkcji. Struktura obiektów aplikacji odzwierciedla wtedy strukturę fizyczną dokumentu.

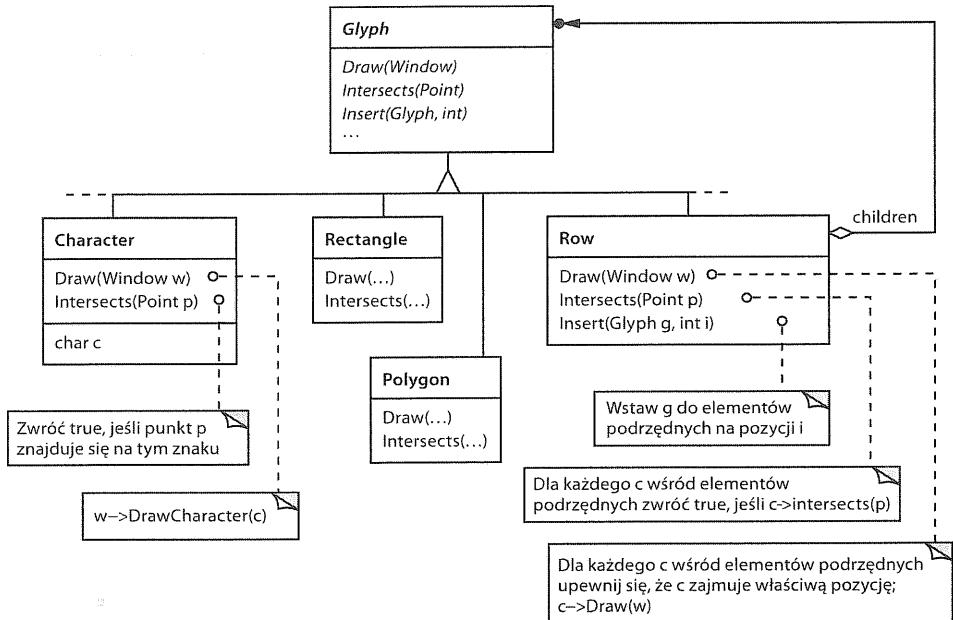
To podejście ma dwie ważne konsekwencje. Pierwsza z nich jest oczywista — obiektom muszą odpowiadać klasy. Druga może być mniej widoczna — klasy muszą mieć zgodne interfejsy, ponieważ chcemy traktować obiekty w jednolity sposób. W językach podobnych do C++ zgodność interfejsów można uzyskać przez powiązanie klas relacją dziedziczenia.

KLASY Z RODZINY GLYPH

Zdefiniujmy klasę abstrakcyjną **Glyph** na potrzeby wszystkich obiektów, które mogą pojawić się w strukturze dokumentu³. Jej podklasy określają zarówno proste elementy graficzne (takie jak znaki i rysunki), jak i elementy strukturalne (na przykład wiersze i kolumny). Rysunek 2.4 ilustruje reprezentatywną część hierarchii klasy **Glyph**, a tabela 2.1 przedstawia bardziej szczegółowy opis interfejsu glifów w języku C++⁴.

RYSUNEK 2.4.

Część
hierarchii
klasy Glyph



Glify mają trzy główne zadania. Muszą wiedzieć: (1) jak wyświetlić swoją reprezentację; (2) jaką przestrzeń zajmują; (3) jakie elementy podrzędne (dzieci) i nadrzędne (rodziców) mają.

³ Jako pierwszy pojęcia „glyph” (czyli glif) w tym kontekście użył Calder [CL90]. W większości współczesnych edytorów dokumentów — prawdopodobnie ze względu na wydajność — nie stosuje się odrębnych obiektów do reprezentowania każdego znaku. Calder w swojej rozprawie dowodził, że opisane przez niego podejście jest realne [Cal93]. Zastosowane przez nas glify są mniej rozbudowane od glifów Caldera, ponieważ dla uproszczenia ograniczyliśmy się do zwięzlej hierarchii. Glify Caldera można współużytkować (w celu zmniejszenia ilości potrzebnej pamięci), tworząc w ten sposób acykliczne grafy skierowane. Aby uzyskać ten sam efekt, mogliśmy zastosować wzorzec Pylek (s. 195), jednak napisanie takiego rozwiązania pozostawiamy jako ćwiczenie dla Czytelnika.

⁴ Opisany tu interfejs celowo ograniczyliśmy do minimum, aby uprościć analizy. Kompletny interfejs powinien obejmować operacje do zarządzania atrybutami graficznymi (na przykład do zmiany koloru, czcionki i współrzędnych), a także operacje do zaawansowanego manipulowania elementami podrzędnymi.

TABELA 2.1. Podstawowy interfejs glifów

Zadanie	Operacja
Wygląd	<code>virtual void Draw(Window*)</code> <code>virtual void Bounds(Rect&)</code>
Wykrywanie trafień	<code>virtual bool Intersects(const Point&)</code>
Struktura	<code>virtual void Insert(Glyph*, int)</code> <code>virtual void Remove(Glyph*)</code> <code>virtual Glyph* Child(int)</code> <code>virtual Glyph* Parent()</code>

Podklasy klasy `Glyph` obejmują nowe definicje operacji `Draw`, co umożliwia wyświetlanie egzemplarzy tych podklas w oknie. Egzemplarze te przyjmują w wywołaniu operacji `Draw` referencję do obiektu `Window`. Klasa `Window` określa operacje graficzne potrzebne do wyświetlania tekstu i podstawowych kształtów w oknie na ekranie. W podklasie `Rectangle` klasy `Glyph` operację `Draw` można zdefiniować w następujący sposób:

```
void Rectangle::Draw (Window* w) {
    w->DrawRect(_x0, _y0, _x1, _y1);
}
```

Parametry `_x0`, `_y0`, `_x1` i `_y1` to składowe klasy `Rectangle` wyznaczające dwa przeciwnieległe wierzchołki prostokąta. `DrawRect` to operacja klasy `Window` wyświetlająca prostokąt na ekranie.

Glif nadzędny często musi wiedzieć, jaką przestrzeń zajmuje glif podrzędny. Jest to potrzebne na przykład do uporządkowania glifów w wierszu tak, aby nie nachodziły na siebie (ilustruje to rysunek 2.2). Operacja `Bounds` zwraca informacje o prostokątnym obszarze zajmowanym przez dany glif. Mają one postać współrzędnych przeciwnieległych wierzchołków najmniejszego prostokąta obejmującego ten glif. W podklasach klasy `Glyph` operacja ta jest ponownie zdefiniowana i zwraca prostokątny obszar, w którym elementy wyświetla dana podkلاś.

Operacja `Intersects` zwraca informacje o tym, czy dany punkt znajduje się na glifie. Za każdym razem, kiedy użytkownik kliknie dokument, edytor Lexi wywołuje tę operację, aby ustalić, który glif lub która struktura znajduje się pod kursemem myszy. W klasie `Rectangle` ponownie zdefiniowano tę operację, aby sprawdzała, czy dany punkt znajduje się w określonym prostokącie.

Ponieważ glify mogą mieć elementy podrzędne, potrzebny jest wspólny interfejs do dodawania i usuwania tych elementów oraz uzyskiwania do nich dostępu. Na przykład elementy podrzędne klasy `Row` to glify porządkowane do postaci wiersza. Operacja `Insert` wstawia glif w miejscu określonym przez całkowitoliczbowy indeks⁵. Operacja `Remove` usuwa określony glif, jeśli rzeczywiście jest on elementem podrzędnym.

⁵ Indeks całkowitoliczbowy prawdopodobnie nie jest najlepszym sposobem wskazywania elementów podrzędnych glifu, choć zależy to od struktury danych zastosowanej w glifie. Jeśli elementy podrzędne są przechowywane na liście powiązanej, wtedy wydajniejszy jest wskaźnik do tej listy. Lepsze rozwiązanie problemu indeksowania przedstawiamy w podrozdziale 2.8 przy okazji omawiania analizowania dokumentu.

Operacja `Child` zwraca element podrzędny (jeśli taki istnieje) dla danego indeksu. W glifach podobnych do obiektów `Row`, które mogą mieć elementy podrzędne, należy wewnętrznie używać operacji `Child`, zamiast bezpośrednio korzystać ze struktur danych takich elementów. Dzięki temu po zmianie struktury danych na przykład z tablicy na listę powiązaną nie trzeba będzie modyfikować takich operacji jak `Draw`, które są powtarzane dla wszystkich elementów podrzędnych. Podobnie operacja `Parent` zapewnia standardowy interfejs dostępu do elementu nadzrzednego (jeśli taki istnieje). Glify w edytorze Lexi przechowują referencję do elementu nadzrzednego, a operacja `Parent` po prostu zwraca tę referencję.

WZORZEC KOMPOZYT

Składanie rekurencyjne jest skuteczne nie tylko w przypadku dokumentów. Technikę tę można wykorzystać do utworzenia reprezentacji dowolnej — potencjalnie złożonej — struktury hierarchicznej. Wzorzec Kompozyt (s. 170) ujmuję istotę składania rekurencyjnego w kategoriach obiektowych. Teraz jest dobry moment na zajrzenie do tego wzorca i przeanalizowanie go. W razie potrzeby możesz wrócić do niniejszego opisu.

2.3. FORMATOWANIE

Ustaliliśmy już sposób *reprezentowania* fizycznej struktury dokumentu. Następnie musimy ustalić, jak utworzyć *konkretną* strukturę fizyczną odpowiadającą poprawnie sformatowanemu dokumentowi. Reprezentowanie i formatowanie to nie to samo. Możliwość uchwycenia fizycznej struktury dokumentu nie określa, jak uzyskać tę konkretną strukturę. Za to ostatnie zadanie odpowiada przede wszystkim edytor Lexi. Musi on dzielić tekst na wiersze, wiersze na kolumny itd., a przy tym uwzględniać wysokopoziomowe oczekiwania użytkownika. Dana osoba może na przykład zechcieć zmienić szerokość marginesu, wcięcia lub tabulacji, zastosować pojedyncze albo podwójne odstępy, a prawdopodobnie także ustalić wiele innych ograniczeń dotyczących formatu⁶. W algorytmie formatowania w edytorze Lexi trzeba uwzględnić wszystkie te kwestie.

Przy okazji — pojęcie „formatowanie” stosujemy tylko do określania podziału kolekcji glifów na wiersze. Będziemy stosować zamiennie zwroty „formatowanie” i „podział na wiersze”. Techniki, które opisujemy, dotyczą w równym stopniu podziału wierszy na kolumny, jak i podziału kolumn na strony.

KAPSUŁKOWANIE ALGORYTMU FORMATOWANIA

Proces formatowania — z uwagi na wszystkie ograniczenia i szczegóły — niełatwo jest zautomatyzować. Istnieje wiele podejść do tego problemu, a programiści wymyślają różne algorytmy formatowania o specyficznych zaletach i wadach. Ponieważ Lexi to edytor działający w trybie

⁶ Użytkownik w jeszcze większym stopniu będzie określał *logiczną* strukturę dokumentu — podział na zdania, akapity, punkty, rozdziały itd. Struktura *fizyczna* jest w porównaniu z tym mniej interesująca. Dla większości osób nie ma znaczenia, gdzie znajdują się punkty podziału wierszy w akapicie, o ile tylko on sam jest poprawnie sformatowany. To samo dotyczy formatowania kolumn i stron. Dlatego użytkownicy określają tylko wysokopoziomowe ograniczenia dotyczące struktury fizycznej i pozwalają wykonać ciężką pracę edytorowi Lexi, który ma spełnić ich oczekiwania.

WYSIWYG, ważną kwestią do rozważenia jest zachowanie równowagi między jakością i szybkością formatowania. Zwykle pożądane jest, aby edytor reagował szybko, a przy tym zapewniał atrakcyjny wygląd dokumentu. Poziom równowagi między tymi aspektami zależy od wielu czynników, a nie wszystkie z nich można ustalić w czasie komplikacji. Na przykład użytkownik może pogodzić się z wolniejszym czasem reakcji, jeśli otrzyma w zamian lepsze formatowanie. Dlatego może się okazać, że zupełnie odmienny algorytm formatowania będzie bardziej odpowiedni od obecnie stosowanego. Inne, bardziej zależne od implementacji kwestie dotyczą wymogów związanych z szybkością i pamięcią. Czasem można skrócić czas formatowania przez zapisanie w buforze większej ilości informacji.

Ponieważ algorytmy formatowania zwykle są złożone, pożądane jest, aby stanowiły zamkniętą całość lub — co jeszcze lepsze — były zupełnie niezależne od struktury dokumentu. W idealnym rozwiązaniu można dodać nową podklasę klasy `Glyph` bez zwracania uwagi na algorytm formatowania. Z drugiej strony dodanie nowego algorytmu tego rodzaju nie powinno wymagać modyfikowania istniejących glifów.

Te cechy wskazują na to, że należy zaprojektować edytor Lexi w taki sposób, aby można łatwo zmienić algorytm formatowania przynajmniej w czasie komplikacji (a najlepiej także w czasie wykonywania programu). Można odizolować algorytm i jednocześnie ułatwić jego zastępowanie przez jego zakapsułkowanie w obiekcie. Ujmijmy to precyzyjniej — należy zdefiniować odrębną hierarchię klas obiektów kapsułkujących algorytmy formatowania. Element główny tej hierarchii powinien zawierać definicję interfejsu dla wielu różnorodnych algorytmów formatowania, a w każdej podklasie należy zaimplementować ten interfejs w celu zrealizowania konkretnego algorytmu. Następnie można dodać podklasę klasy `Glyph` automatycznie określającą strukturę elementów podrzędnych za pomocą danego obiektu algorytmu.

KLASY COMPOSITOR I COMPOSITION

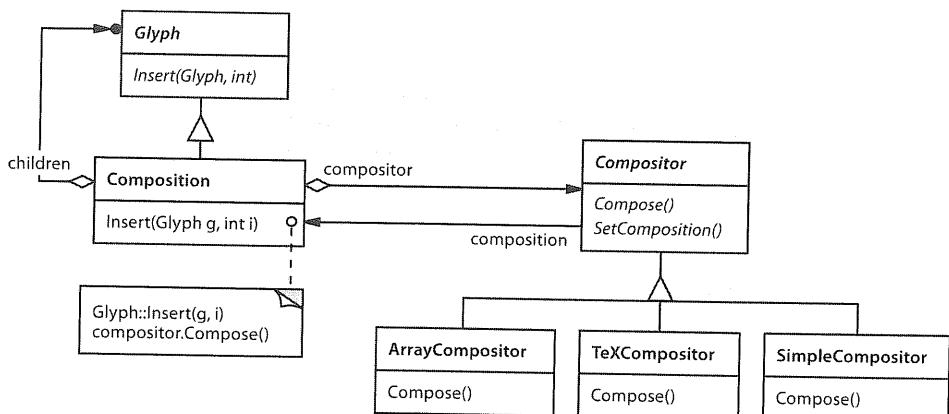
Zdefiniujmy klasę `Compositor` na potrzeby obiektów kapsułkujących algorytm formatowania. Interfejs (tabela 2.2) umożliwia obiektom tej klasy ustalenie, które glify należy sformatować i kiedy to zrobić. Formatowane glify to elementy podrzędne specjalnej podklasy klasy `Glyph` — `Composition`. Obiekty tej klasy otrzymują w momencie powstawania egzemplarz podklasy `Compositor` (wyspecjalizowanej w obsłudze określonego algorytmu podziału na wiersze) i przez wywołanie operacji `Compose` nakazują mu w razie potrzeby (na przykład po zmodyfikowaniu dokumentu przez użytkownika) połączenie glifów. Rysunek 2.5 przedstawia relacje między klasami `Composition` i `Compositor`.

TABELA 2.2. Podstawowy interfejs obiektu łączącego

Zadanie	Operacje
Co formatować?	<code>void SetComposition(Composition*)</code>
Kiedy formatować?	<code>virtual void Compose()</code>

RYSUNEK 2.5.

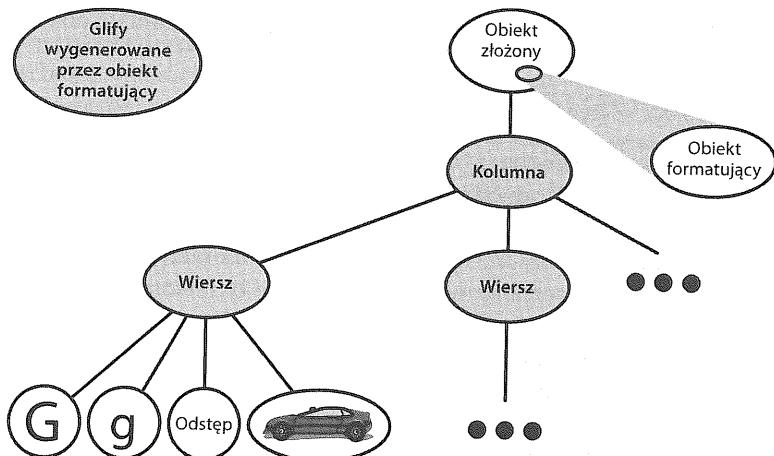
Relacje między klasami Composition i Compositor



Niesformatowany obiekt `Composition` obejmuje tylko widoczne glify składające się na podstawową zawartość dokumentu. Nie zawiera glifów określających fizyczną strukturę dokumentu, na przykład obiektów `Row` lub `Column`. Obiekt `Composition` znajduje się w tym stanie bezpośrednio po jego utworzeniu i zainicjowaniu za pomocą glifów, które obiekt ten ma sformatować. Kiedy zachodzi potrzeba sformatowania elementów, obiekt wywołuje operację `Compose` obiektu `Compositor`. Wtedy obiekt `Compositor` przechodzi do elementach podrzędnych obiektu `Composition` i wstawia nowe glify `Row` oraz `Column` zgodnie z algorytmem podziału na wiersze⁷. Rysunek 2.6 ilustruje uzyskaną w ten sposób strukturę obiektów. Glify utworzone i wstawione do struktury obiektów przez obiekt `Compositor` są wyróżnione szarym tłem.

RYSUNEK 2.6.

Struktura obiektów odzwierciedlających podział na wiersze przeprowadzony przez obiekt Compositor



Każda podkلاza klasy `Compositor` może zawierać implementację innego algorytmu podziału na wiersze. Na przykład podklaza `SimpleCompositor` może przeprowadzać szybką analizę bez zagłębiania się w tak wymyślne zagadnienia jak „światło” dokumentu. Dobre „światło”

⁷ Obiekt `Compositor` musi otrzymać kody ze znaków glifów `Character`, aby móc obliczyć miejsca podziału wierszy. W podrozdziale 2.8 pokażemy, jak uzyskać te informacje polimorficznie, bez dodawania operacji specyficznych dla znaków do interfejsu `Glyph`.

oznacza równomierny rozkład tekstu i odstępów. W podklasie `TeXCompositor` można zaimplementować pełny algorytm `TEX` [Knu84]. Uwzględnia on kwestie takie jak „światło”, jednak odbywa się to kosztem dłuższego czasu formatowania.

Podział na klasy `Compositor` i `Composition` gwarantuje mocne oddzielenie kodu obsługującego fizyczną strukturę dokumentu od kodu poszczególnych algorytmów formatowania. Dzięki temu można dodawać nowe podklasy klasy `Compositor` bez modyfikowania klas glifów (i na odwrót). W rzeczywistości można zmienić algorytm podziału na wiersze w czasie wykonywania programu przez dodanie jednej operacji `SetCompositor` do podstawowego interfejsu glifów klasy `Composition`.

WZORZEC STRATEGIA

Do kapsułkowania algorytmów w obiektach służy wzorzec Strategia (s. 321). Kluczowe elementy tego wzorca to obiekty `Strategy` (kapsułkują różne algorytmy) i kontekst ich działania. Obiekty `Compositor` reprezentują strategie — kapsułkują różne algorytmy formatowania. Obiekt `Composition` to kontekst działania strategii z obiektu `Compositor`.

Kluczem do zastosowania wzorca Strategia jest zaprojektowanie wystarczająco ogólnych (umożliwiających zastosowanie różnorodnych algorytmów) interfejsów dla strategii i kontekstu. Zmienianie interfejsu strategii lub kontekstu w celu dodania obsługi nowego algorytmu nie powinno być konieczne. W omawianym przykładzie podstawowy interfejs klasy `Glyph` zapewnia wystarczająco ogólną obsługę wstawiania i usuwania elementów podrzędnych oraz dostępu do nich, aby podklasy klasy `Compositor` mogły zmieniać fizyczną strukturę dokumentu niezależnie od tego, jaki algorytm do tego stosują.

2.4. OZDABIANIE INTERFEJSU UŻYTKOWNIKA

Rozważmy dwa ozdobniki interfejsu użytkownika edytora Lexi. Pierwszy z nich to ramka wokół obszaru edycji tekstu wydzielająca jedną stronę. Drugi to pasek przewijania umożliwiający użytkownikom wyświetlanie różnych fragmentów stron. Aby ułatwić dodawanie i usuwanie takich mechanizmów (zwłaszcza w czasie wykonywania programu), nie należy przy dołączaniu ich do interfejsu użytkownika stosować dziedziczenia. Największą elastyczność można uzyskać, jeśli inne obiekty interfejsu użytkownika nie będą nawet wiedzieć o istnieniu ozdobników. Pozwoli to dodawać i usuwać mechanizmy bez modyfikowania innych klas.

NIEWIDOCZNA OTOCZKA

W kontekście programowania ozdabianie interfejsu użytkownika polega na rozbudowaniu istniejącego kodu. Zastosowanie do tego dziedziczenia uniemożliwia zmianę układu ozdobników w czasie wykonywania programu, a równie poważnym problemem jest specyficzny dla dziedziczenia gwałtowny wzrost liczby klas.

Można dodać obramowanie do klasy `Composition` przez utworzenie na jej podstawie podklasy `BorderedComposition`. W podobny sposób, przez utworzenie podklasy `ScrollableComposition`, można utworzyć interfejs do przewijania. Jeśli potrzebne są zarówno paski przewijania, jak i obramowanie, można przygotować podkласę `BorderedScrollableComposition` itd. W najgorszym przypadku trzeba będzie dodać jedną klasę dla każdej kombinacji ozdobników. Przy wzroście ich różnorodności rozwiązywanie to szybko stanie się niepraktyczne.

Składanie obiektów to potencjalnie bardziej praktyczny i elastyczny mechanizm rozbudowywania. Które jednak obiekty należy połączyć? Ponieważ wiemy, że ozdabiamy istniejący glif, same ozdobniki można utworzyć jako obiekty (na przykład egzemplarze klasy `Border`). Daje to dwóch kandydatów do składania — glif i obramowanie. W następnym kroku trzeba zdecydować, do którego obiektu należy dołączyć ten drugi. Obramowanie może obejmować glif, co ma sens, ponieważ ramka otacza glif na ekranie. Można też zastosować odwrotne rozwiązanie (umieścić ramkę w glifie), jednak wtedy trzeba wprowadzić zmiany w odpowiedniej podklasie klasy `Glyph` i zapisać w niej informacje o obramowaniu. Pierwsza propozycja, dołączenie glifu do ramki, pozwala umieścić cały kod do rysowania obramowania w klasie `Border` i uniknąć zmian w innych klasach.

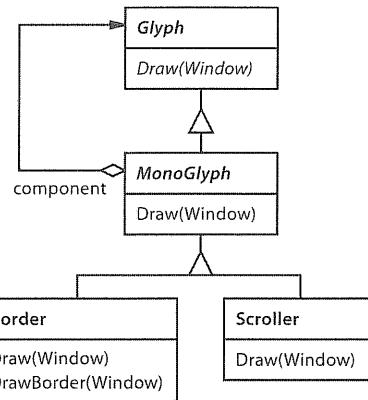
Jak wygląda klasa `Border`? Ponieważ ramki są widoczne na ekranie, powinny być glifami. Oznacza to, że `Border` powinna być podklassą klasy `Glyph`. Jest też ważniejszy powód zastosowania tego rozwiązania — dla klientów nie powinno być istotne, czy glify mają ramkę. Wszystkie glify należy traktować w taki sam sposób. Kiedy klient nakaże wyświetlić się zwykłemu glifowi bez ramki, glif powinien zrobić to bez używania ramki. Jeśli dany glif jest połączony z obramowaniem, klienci nie powinny traktować ramki obejmującej glif w specjalny sposób. Muszą jedynie nakazać jej się wyświetlić, podobnie jak w przypadku zwykłego glifu. Oznacza to, że interfejs klasy `Border` powinien być zgodny z interfejsem klasy `Glyph`. Aby zagwarantować tę relację, utworzymy klasę `Border` jako podkласę klasy `Glyph`.

Wszystko to prowadzi do zagadnienia *niewidocznej otoczki* (ang. *transparent enclosure*). Technika ta łączy (1) składanie z pojedynczym elementem podrzędnym (lub pojedynczym komponentem) i (2) zastosowanie zgodnych interfejsów. Klienci zwykle nie potrafią określić, czy używają komponentu czy otoczki (na przykład elementu nadzawanego w danym elemencie podrzędnym). Jest to prawda zwłaszcza wtedy, jeśli otoczka deleguje wszystkie operacje do komponentu. Jednak otoczka może też *wzbogacać* działanie komponentu przez samodzielne wykonywanie zadań przed delegowaniem operacji i (lub) po tej operacji. Ponadto otoczka może dodawać stan do komponentu. W następnym punkcie pokażemy, jak to zrobić.

KLASA MONOGLYPH

Niewidoczną otoczkę można zastosować do wszystkich glifów ozdabiających inne glify. Aby w konkretny sposób przedstawić to zagadnienie, zdefiniujemy na podstawie klasy `Glyph` podkласę `MonoGlyph`. Będzie to klasa abstrakcyjna „ozdobnych glifów”, takich jak `Border` (rysunek 2.7). Obiekty `MonoGlyph` przechowują referencję do komponentu i przekazują wszystkie skierowane do niego żądania.

RYSUNEK 2.7.
Relacje klasy
MonoGlyph



Sprawia to, że klasa MonoGlyph jest całkowicie niewidoczna dla klientów. Na przykład implementacja operacji Draw w klasie MonoGlyph wygląda tak:

```

void MonoGlyph::Draw (Window* w) {
    _component->Draw(w);
}
  
```

Podklasy klasy MonoGlyph obejmują nowe implementacje przynajmniej jednej z przekazywanych operacji. Na przykład operacja Border::Draw najpierw wywołuje dla komponentu operację MonoGlyph::Draw klasy nadzędnej, aby umożliwić komponentowi wykonanie zadanie, czyli narysowanie wszystkich elementów oprócz obramowania. Następnie operacja Border::Draw wyświetla ramkę przez wywołanie prywatnej operacji o nazwie DrawBorder (jej szczegóły pomijamy):

```

void Border::Draw (Window* w) {
    MonoGlyph::Draw(w);
    DrawBorder(w);
}
  
```

Warto zauważyć, że operacja Border::Draw to w istocie *rozszerzenie* operacji klasy nadzędnej umożliwiające narysowanie obramowania. Znacznie różni się to od *zastępowania* operacji klasy nadzędnej, co polegałoby na pominięciu wywołania MonoGlyph::Draw.

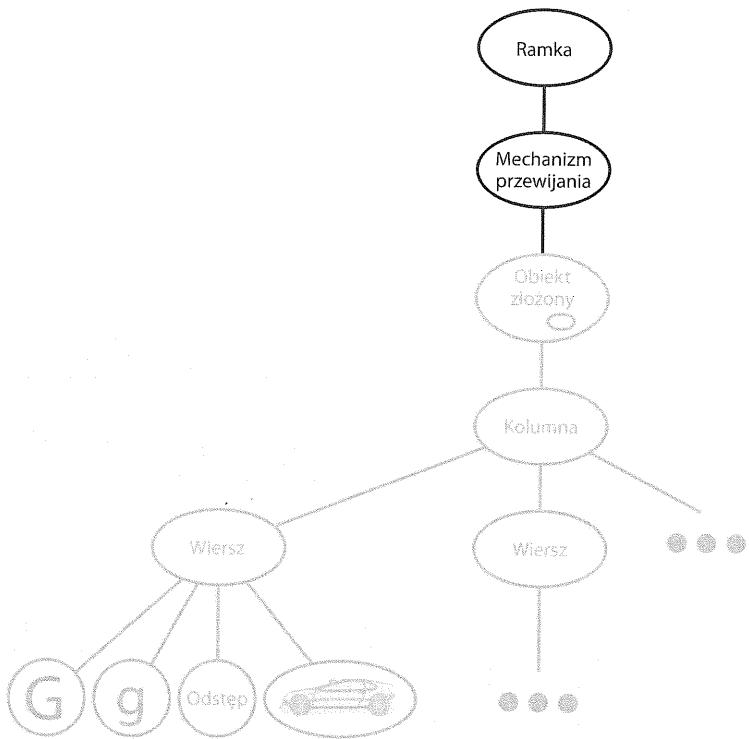
Na rysunku 2.7 znajduje się też inna podklasa klasy MonoGlyph — **Scroller**. Rysuje ona swój komponent w różnych miejscach w zależności od pozycji dwóch pasków przewijania (są to dołączone ozdobniki). Kiedy obiekt Scroller wyświetla swój komponent, informuje system graficzny, że należy przyciąć komponent do rozmiarów obiektu. Przycięcie fragmentów komponentu przewiniętych poza widoczny obszar powoduje, że nie pojawią się one na ekranie.

Teraz mamy już wszystkie elementy potrzebne do dodania obramowania i pasków przewijania do obszaru edycji tekstu w aplikacji Lexi. Dołączymy istniejący egzemplarz klasy **Composition** do egzemplarza klasy **Scroller** w celu dodania pasków przewijania, a następnie dołączymy całość do egzemplarza klasy **Border**. Ostateczną strukturę obiektów przedstawia rysunek 2.8.

Zauważmy, że można odwrócić kolejność składania i umieścić złożenie z obramowaniem w egzemplarzu klasy **Scroller**. Wtedy ramka będzie przewijana wraz z tekstem, co nie zawsze jest pożądane. Ważne jest to, że niewidoczna otoczka pozwala łatwo testować różne możliwości i tworzyć klienty pozbawione kodu związanego z ozdobnikami.

RYSUNEK 2.8.

Struktura obiektów z ozdobnikami



Ponadto warto zauważyć, że do obramowania dołączany jest tylko jeden glif. Jest to rozwiązańe odmienne od wcześniej zdefiniowanych złożień, w których obiekty nadrzędne mogły mieć dowolną liczbę elementów podrzędnych. Tu umieszczenie obramowania wokół czegoś oznacza, że to „coś” występuje pojedynczo. Można znaleźć uzasadnienie dla ozdabiania więcej niż jednego obiektu naraz, jednak takie podejście wymagałoby połączenie techniki ozdabiania z wieloma rodzajami złożień — ozdabianie wiersza, ozdabianie kolumny itd. Nie jest to pomocne, ponieważ mamy już klasy do tworzenia złożień tego typu. Dlatego lepiej będzie wykorzystać istniejące klasy w ramach składania i dodać nowe klasy do ozdabiania wynikowych struktur. Zachowanie niezależności ozdabiania od innych rodzajów składania pozwala uprościć klasy używane do ozdabiania i zmniejszyć ich liczbę. Ponadto nie trzeba wtedy powielać istniejących mechanizmów do obsługi składania.

WZORZEC DEKORATOR

Wzorzec Dekorator (s. 152) ujmuje relacje między klasami i obiektami służącymi do ozdabiania za pomocą niewidocznej otoczki. Pojęcie „ozdobnik” ma szersze znaczenie, niż przedstawiliśmy to w tym miejscu. We wzorcu Dekorator ozdobnik to wszystko, co dodaje obsługę nowych zadań do obiektu. Można na przykład wzbogacić drzewo składni abstrakcyjnej o operacje semantyczne, automat skończony o nowe przejścia lub sieć trwałych obiektów o oznaczenia dotyczące atrybutów. Wzorzec Dekorator uogólnia podejście zastosowane w edytorze Lexi i sprawia, że można wykorzystać je w wielu warunkach.

2.5. OBSŁUGA WIELU STANDARDÓW WYGLĄDU I DZIAŁANIA

Osiągnięcie przenośności między platformami sprzętowymi i programowymi to poważny problem w obszarze projektowania systemów. Przystosowanie edytora Lexi do nowej platformy nie powinno wymagać poważnych przekształceń. Jeśli są one konieczne, przystosowywanie może nie być warte zachodu. Przenoszenie systemu powinno być tak łatwe, jak to możliwe.

Jedną z przeszkód zmniejszających przenośność jest różnorodność standardów wyglądu i działania mających zapewniać jednolitość aplikacji. Te standardy wyznaczają wytyczne w zakresie wyglądu programów i sposobu ich reagowania na działania użytkowników. Choć istniejące standardy nie różnią się znacznie między sobą, trudno jest je ze sobą pomylić. Aplikacje zgodne ze standardem Motif nie wyglądają i nie działają w dokładnie taki sam sposób jak ich odpowiedniki na innych platformach (i na odwrót). Program uruchamiany na więcej niż jednej platformie musi być zgodny z wytycznymi z zakresu stylu interfejsu użytkownika każdej z tych platform.

Za cel projektowy postawiliśmy sobie sprawienie, aby edytor Lexi był zgodny z wieloma istniejącymi standardami wyglądu i działania oraz umożliwiał łatwe dodawanie obsługi nowych standardów po ich pojawienniu się (jest to nieuniknione). Chcemy też, aby projekt zapewniał najwyższy poziom elastyczności — umożliwiał zmianę wyglądu i działania edytora Lexi w czasie wykonywania programu.

ABSTRAKCYJNE UJĘCIE PROCESU TWORZENIA OBIEKTÓW

Wszystko, co widzimy i z czym wchodzimy w interakcje w interfejsie użytkownika edytora Lexi, to glify złożone z innymi, niewidocznymi glifami (takimi jak Row i Column). Niewidoczne glify zawierają w sobie glify widoczne, takie jak Button i Character, i zapewniają ich właściwy układ. W wytycznych z zakresu stylu dużo miejsca poświęca się wyglądom i działaniom tak zwanych „widgetów” (jest to inna nazwa widocznych glifów, takich jak przyciski, paski przewijania i menu, pełniących funkcję elementów sterujących w interfejsie użytkownika). W widżetach można używać do prezentacji danych prostszych glifów, na przykład znaków, okręgów, prostokątów i wielokątów.

Zakładamy, że istnieją dwa zestawy klas glifów-widgetów, które posłużą do implementacji obsługi wielu standardów wyglądu i działania:

1. Zestaw abstrakcyjnych podklas klasy `Glyph` dla każdej kategorii glifu-widgetu. Na przykład klasa abstrakcyjna `ScrollBar` będzie wzbogacać podstawowy interfejs glifu o ogólne operacje przewijania, klasa abstrakcyjna `Button` doda operacje związane z przyciskami itd.
2. Zestaw konkretnych podklas dla każdej abstrakcyjnej podklasty. Te konkretne podklasty będą obsługiwać różne standardy wyglądu i działania. Na przykład klasa `ScrollBar` może mieć podklasty `MotifScrollBar` i `PMScrollBar` obsługujące paski przewijania zgodne ze stylami Motif oraz Presentation Manager.

Edytor Lexi musi rozróżniać glify-widżety związane z poszczególnymi standardami wyglądu i działania. Na przykład kiedy zajdzie potrzeba umieszczenia przycisku w interfejsie, edytor będzie musiał utworzyć egzemplarz podklasy klasy `Glyph` odpowiadający przyciskowi o określonym stylu (`MotifButton`, `PButton`, `MacButton` itd.).

Oczywiste jest, że kod edytora Lexi nie może tego zrobić bezpośrednio, na przykład przez wywołanie konstruktora w języku C++. Spowodowałoby to zapisanie na stałe w kodzie przycisku o określonym stylu, co uniemożliwia wybór stylu w czasie wykonywania programu. Przy tym rozwiążaniu trzeba ponadto znaleźć i zmienić każde wywołanie konstruktora, aby przenieść edytor Lexi na inną platformę. Pamiętajmy, że przyciski to tylko jeden z wielu widżetów w interfejsie użytkownika edytora Lexi. Zaśmiecanie kodu wywołaniami konstruktorów klas powiązanych ze specyficzny standardem wyglądu i działania znacznie utrudnia konserwację systemu. Wystarczy zapomnieć o jednym takim konstruktorem, a w aplikacji dla systemu Mac pojawi się menu zgodne ze standardem Motif.

W edytorze Lexi potrzebny jest sposób na określanie stosowanego standardu wyglądu i działania. Pozwoli to utworzyć odpowiednie widżety. Unikanie jawnych wywołań konstruktorów to niejedyna rzecz, o której należy pamiętać. Potrzebna jest też możliwość łatwego zastępowania całych zestawów widżetów. Oba te cele można osiągnąć przez *abstrakcyjne ujęcie procesu tworzenia obiektów*. Na czym to polega? Wyjaśnimy to na przykładzie.

FABRYKI I KLASY PRODUKTÓW

Zwykle można utworzyć egzemplarz glifu w postaci paska przewijania zgodny ze standardem Motif za pomocą następującego kodu C++:

```
ScrollBar* sb = new MotifScrollBar;
```

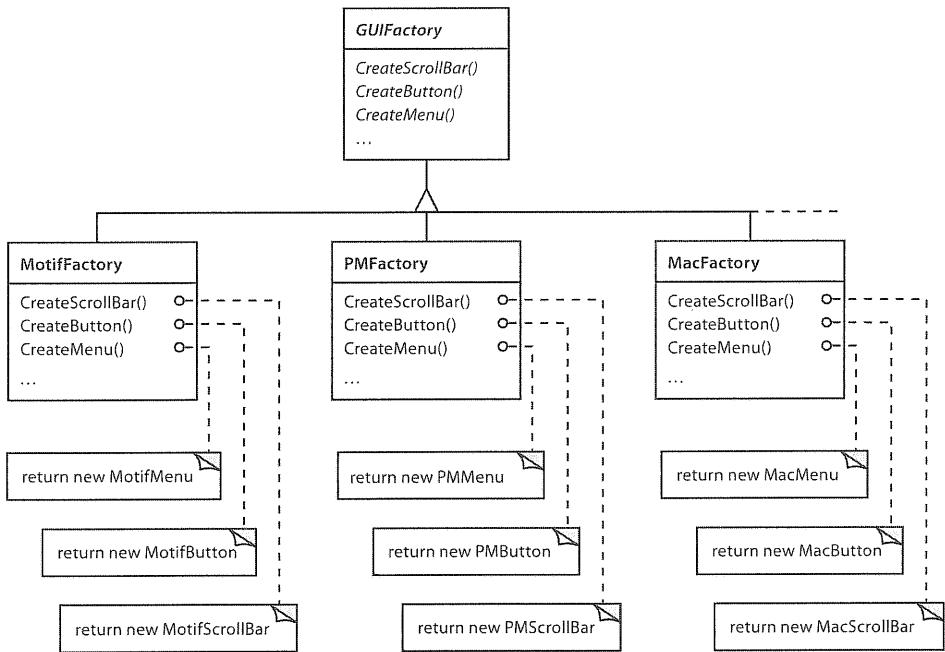
Jeśli celem jest zminimalizowanie zależności w zakresie wyglądu i działania edytora Lexi, kodu tego rodzaju należy unikać. Założymy jednak, że inicjujemy obiekt sb w poniższy sposób:

```
ScrollBar* sb = guiFactory->CreateScrollBar();
```

W tym kodzie `guiFactory` to egzemplarz klasy **MotifFactory**. Operacja `CreateScrollBar` zwraca nowy egzemplarz odpowiedniej podklasy klasy `ScrollBar` zgodny z wybranym standardem wyglądu i działania (tu jest to standard Motif). Z perspektywy klienta efekt jest taki sam jak przy jawnym wywołaniu konstruktora `MotifScrollBar`. Jednak nowe podejście znacznie różni się od poprzedniego — w kodzie nie ma żadnej wzmianki o standardzie Motif. Obiekt `guiFactory` abstrahuje proces tworzenia nie tylko pasków przewijania zgodnych ze standardem Motif, ale też pasków przewijania dla *dowolnego* standardu wyglądu i działania. Możliwości tego obiektu nie ograniczają się do generowania pasków przewijania. Potrafi on tworzyć różnorodne glify-widżety, w tym paski przewijania, przyciski, pola na dane wejściowe, menu itd.

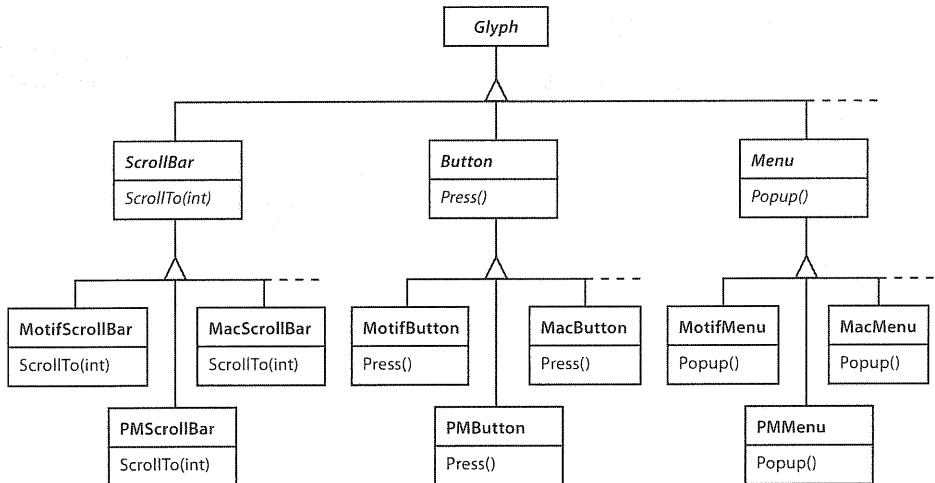
Wszystko to jest możliwe, ponieważ `MotifFactory` to podklasa klasy `GUIFactory` — klasy abstrakcyjnej definiującej ogólny interfejs do tworzenia glifów-widgetów. Obejmuje ona takie operacje jak `CreateScrollBar` i `CreateButton`. Służą one do tworzenia egzemplarzy różnych rodzajów glifów-widgetów. W podklasach klasy `GUIFactory` operacje te są zaimplementowane w taki sposób, aby zwracały glify zgodne z określonym standardem wyglądu i działania, na przykład obiekty `MotifScrollBar` lub `PButton`. Rysunek 2.9 przedstawia wynikową hierarchię klas obiektów `guiFactory`.

RYSUNEK 2.9.
Hierarchia
klasy
GUIFactory



Stwierdziliśmy, że fabryki tworzą obiekty będące **produktyami**. Ponadto produkty wytwarzane przez fabrykę są powiązane ze sobą. Tu wszystkie produkty to widgety o tym samym wyglądzie i działaniu. Rysunek 2.10 przedstawia wybrane klasy produktów potrzebne do tego, aby można zastosować fabryki do tworzenia glifów-widgetów.

RYSUNEK 2.10.
Abstrakcyjne
klasy
produktów
i konkretne
podklasy



Pozostaje nam udzielić odpowiedzi na ostatnie pytanie: „Skąd pochodzą egzemplarze klasy GUIFactory?”. Odpowiedź brzmi: „Z dowolnego dogodnego miejsca”. Zmienna `guiFactory` może być zmienną globalną, składową statyczną znanej klasy, a nawet zmienną lokalną, jeśli cały interfejs użytkownika działa w ramach jednej klasy lub funkcji. Istnieje nawet wzorzec

projektowy Singleton (s. 130) przeznaczony do zarządzania znymi niepowtarzalnymi obiektami tego rodzaju. Trzeba jednak pamiętać, aby zainicjować obiekt `guiFactory` przed użyciem go do tworzenia widgetów, ale po ustaleniu odpowiedniego standardu wyglądu i działania.

Jeśli standard wyglądu i działania jest znany w czasie komplikacji, obiekt `guiFactory` można zainicjować przez proste przypisanie do niego nowego egzemplarza fabryki w początkowej części programu:

```
GUIFactory* guiFactory = new MotifFactory;
```

Jeżeli użytkownik może określić standard wyglądu i działania za pomocą nazwy w momencie uruchamiania edytora, fabrykę można utworzyć w następujący sposób:

```
GUIFactory* guiFactory;
const char* styleName = getenv("LOOK_AND_FEEL");
// Użytkownik lub środowisko określa standard przy uruchamianiu programu.

if (strcmp(styleName, "Motif") == 0) {
    guiFactory = new MotifFactory;

} else if (strcmp(styleName, "Presentation_Manager") == 0) {
    guiFactory = new PMFactory;

} else {
    guiFactory = new DefaultGUIFactory;
}
```

Istnieją też bardziej zaawansowane sposoby wyboru fabryki w czasie wykonywania programu. Na przykład można przechowywać rejestr z odwzorowaniami łańcuchów znaków na obiekty reprezentujące fabryki. Umożliwia to rejestrowanie egzemplarzy nowych podklas fabryk bez modyfikowania istniejącego kodu, co nie jest możliwe w opisany wcześniej podejściu. Nie trzeba też dołączać do aplikacji wszystkich fabryk specyficznych dla platformy. Jest to ważne, ponieważ dołączenie fabryki `MotifFactory` w platformie nieobsługującej standardu Motif może okazać się niemożliwe.

Istotne jest to, że po skonfigurowaniu aplikacji przez wskazanie odpowiedniego obiektu fabryki standard wyglądu i działania jest już ustwiony. Jeśli zmienimy zdanie, możemy ponownie zainicjować obiekt `guiFactory` przez podanie fabryki specyficznej dla innego standardu oraz odtworzyć interfejs. Niezależnie od tego, jak i gdzie zechcemy zainicjować obiekt `guiFactory`, mamy pewność, że kiedy już to zrobimy, aplikacja zastosuje odpowiedni standard wyglądu i działania bez konieczności wprowadzania w niej zmian.

WZORZEC FABRYKA ABSTRAKCYJNA

Fabryki i produkty to kluczowe elementy wzorca Fabryka abstrakcyjna (s. 101). Wzorzec ten określa, jak generować rodziny powiązanych obiektów-produktów bez bezpośredniego tworzenia egzemplarzy klas. Jego przydatność jest największa, kiedy liczba i ogólne rodzaje obiektów-produktów nie zmieniają się, a poszczególne rodziny produktów różnią się między sobą. Aby wybrać jedną z tych rodzin, należy utworzyć egzemplarz określonej fabryki konkretnej, a następnie konsekwentnie korzystać z niego do tworzenia produktów. Można też

zastępować całe rodziny produktów przez zamianę jednej fabryki konkretnej na egzemplarz innej fabryki tego rodzaju. Specyficzny dla wzorca Fabryka abstrakcyjna nacisk na *rodziny* produktów odróżnia go od pozostałych wzorców konstrukcyjnych, które dotyczą tylko jednego rodzaju obiektów-produktów.

2.6. OBSŁUGA WIELU SYSTEMÓW OKIENKOWYCH

Standard wyglądu i działania to tylko jedno z wielu zagadnień związań z przenośnością. Następnym są środowiska okienkowe, w których działa edytor Lexi. System okienowy platformy tworzy złudzenie obecności wielu nachodzących na siebie okien na ekranie. Taki system zarządza obszarem zajmowanym przez okienka i kieruje do nich dane wejściowe z klawiatury i myszy. Obecnie istnieje kilka ważnych i w dużym stopniu niezgodnych ze sobą systemów okienkowych (na przykład Macintosh, Presentation Manager, Windows, X). Z tych samych przyczyn, dla których zapewniliśmy obsługę wielu standardów wyglądu i działania, chcemy, aby edytor Lexi funkcjonował w tak wielu systemach okienkowych, jak to możliwe.

CZY MOŻNA ZASTOSOWAĆ FABRYKĘ ABSTRAKCYJNĄ?

Początkowo może się wydawać, że jest to następna okazja do zastosowania wzorca Fabryka abstrakcyjna. Jednak ograniczenia związane z przenośnością edytora między systemami okienkowymi znacznie różnią się od ograniczeń dotyczących niezależności aplikacji od standardu wyglądu i działania.

Przy stosowaniu wzorca Fabryka abstrakcyjna założyliśmy, że zdefiniujemy konkretne klasy glifów-widgetów na potrzeby każdego standardu wyglądu i działania. Oznaczało to, że mogliśmy utworzyć konkretne produkty (na przykład MotifScrollBar i MacScrollBar) dla określonego standardu na podstawie abstrakcyjnej klasy produktu (takiej jak ScrollBar). Przypuszczamy jednak, że mamy już kilka hierarchii klas od różnych producentów — po jednej dla każdego standardu. Oczywiście bardzo mało prawdopodobne jest to, że hierarchie te będą zgodne ze sobą. Dlatego nie będzie można utworzyć wspólnej abstrakcyjnej klasy produktów dla widgetów każdego rodzaju (ScrollBar, Button, Menu itd.), a Fabryka abstrakcyjna nie będzie działała bez tych kluczowych klas. Trzeba dopasować różne hierarchie widgetów do wspólnego zestawu interfejsów produktów abstrakcyjnych. Dopiero wtedy będzie można poprawnie zadeklarować operacje Create... w interfejsie fabryki abstrakcyjnej.

Przy tworzeniu widgetów rozwiązaliśmy ten problem przez opracowanie własnych abstrakcyjnych i konkretnych klas produktów. Ponieważ systemy okienkowe mają niezgodne interfejsy programowania, teraz musimy zmierzyć się z podobnym problemem, aby umożliwić działanie edytora Lexi w istniejących systemach tego rodzaju. Jednak tym razem zadanie jest trudniejsze, ponieważ nie możemy sobie pozwolić na zaimplementowanie własnego niestandardowego systemu okienkowego.

Na szczęście istnieje rozwiązanie. Interfejsy systemów okienkowych — podobnie jak standardy wyglądu i działania — nie różnią się znacząco między sobą, ponieważ wszystkie mają w zasadzie tę samą funkcję. Potrzebny jest jednolity zestaw abstrakcji okienkowych, który umożliwi ukrycie implementacji różnych systemów okienkowych za wspólnym interfejsem.

KAPSUŁKOWANIE ZALEŻNOŚCI IMPLEMENTACYJNYCH

W podrozdziale 2.2 przedstawiliśmy klasę Window służącą do wyświetlania na ekranie glifu lub struktury glifu. Nie określiliśmy systemu okienkowego, z którym współdziałała ten obiekt, ponieważ tak naprawdę nie pochodzi on z żadnego konkretnego systemu tego rodzaju. Klasa Window kapsułkuje zadania, które okna wykonują w różnych systemach okienkowych. Okna te:

- ▶ udostępniają operacje do rysowania podstawowych kształtów geometrycznych;
- ▶ potrafią zmniejszać się do postaci ikony i ponownie przyjmować pierwotny rozmiar;
- ▶ mogą zmieniać rozmiar;
- ▶ potrafią na żądanie wyświetlić (lub odtworzyć) swoją zawartość, na przykład po przywróceniu ich pierwotnego rozmiaru lub po odsłonięciu niewidocznego fragmentu.

Klasa Window musi obejmować funkcje okien z różnych systemów okienkowych. Rozważmy dwa skrajne podejścia:

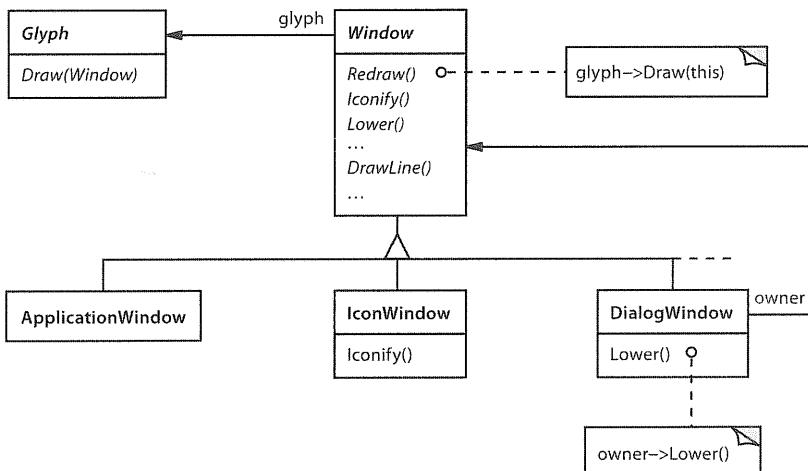
1. *Tylko wspólne funkcje.* Interfejs klasy Window może obejmować tylko funkcje wspólnie *wszystkim* systemom okienkowym. Problem z tym podejściem polega na tym, że interfejs będzie oferował tylko możliwości najmniej rozbudowanego systemu. Nie będzie można wykorzystać bardziej zaawansowanych funkcji, nawet jeśli obsługuje je większość systemów (choć nie wszystkie).
2. *Suma wszystkich funkcji.* Można utworzyć interfejs z uwzględnieniem możliwości *wszystkich* istniejących systemów. Tu problem polega na tym, że interfejs może stać się bardzo duży i nie-spójny. Ponadto trzeba będzie go zmieścić (podobnie jak zależny od niego edytor Lexi) za każdym razem, kiedy jeden z producentów zmodyfikuje interfejs swojego systemu okienkowego.

Żadne z tych skrajnych podejść nie stanowi dobrego rozwiązania, dlatego w projekcie zastosujemy opcję pośrednią. Klasa Window będzie udostępniać wygodny interfejs obsługujący większość popularnych funkcji systemów okienkowych. Ponieważ edytor Lexi będzie korzystał z tej klasy bezpośrednio, musi ona obsługiwać także elementy znane edytorowi, czyli glify. Oznacza to, że w interfejsie klasy Window trzeba umieścić podstawowy zestaw operacji graficznych umożliwiających glifom wyświetlanie swojego obrazu w oknie. Tabela 2.3 przedstawia wybrane operacje z interfejsu klasy Window.

TABELA 2.3. Interfejs klasy Window

Zadanie	
Zarządzanie oknem	<pre>virtual void Redraw() virtual void Raise() virtual void Lower() virtual void Iconify() virtual void Deiconify() ...</pre>
Obsługa grafiki	<pre>virtual void DrawLine() virtual void DrawRect() virtual void DrawPolygon() virtual void DrawText() ...</pre>

Window to klasa abstrakcyjna. Konkretnie podklasy klasy Window obsługują różne okna wykorzystywane przez użytkowników. Oknami są na przykład okna aplikacji, ikony i ostrzegawcze okna dialogowe, jednak każde z nich działa w nieco odmienny sposób. Można więc zdefiniować takie podklasy jak ApplicationWindow, IconWindow i DialogWindow oraz uwzględnić w nich te różnice. Uzyskana w ten sposób hierarchia klas zapewnia aplikacjom (na przykład edytorowi Lexi) jednolitą i intuicyjną abstrakcję obsługi okien niezależną od systemów okienkowych poszczególnych producentów.



Zdefiniowaliśmy już interfejs okna na potrzeby edytora Lexi. Skąd jednak pochodzić będą rzeczywiste okna specyficzne dla platform? Jeśli nie zaimplementujemy własnego systemu okienkowego, w pewnym miejscu abstrakcji okna trzeba będzie ją zaimplementować w kategoriach elementów udostępnianych przez docelowy system okienkowy. Gdzie znajduje się ta implementacja?

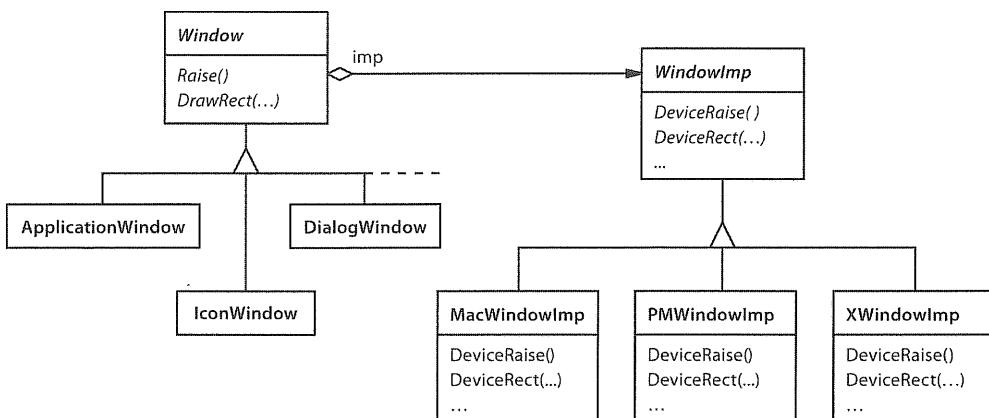
Jedną z możliwości jest zaimplementowanie wielu wersji (po jednej dla każdej platformy) klasy Window i jej podklas. Odpowiednią wersję trzeba wtedy wybrać w czasie kompilowania edytora Lexi na potrzeby danej platformy. Warto jednak wyobrazić sobie problemy z konserwacją tego rozwiązania. Wynikają one z zarządzania wieloma klasami, z których wszystkie mają nazwę „Window”, ale są zaimplementowane w różnych systemach okienkowych. Inna możliwość to utworzenie specyficznych dla implementacji podklas każdej klasy z hierarchii klasy Window. Jednak wtedy po raz kolejny natrafiamy na problem z nagłym wzrostem liczby podklas, podobnie jak przy tworzeniu ozdobników. Oba te rozwiązania mają też inną wadę — nie są na tyle elastyczne, aby umożliwiały zmianę systemu okienkowego po skompilowaniu programu. Dlatego trzeba wtedy przechowywać także kilka różnych plików wykonywalnych.

Żadna z wymienionych możliwości nie jest atrakcyjna, jednak cóż innego można zrobić? To samo, co w przypadku formatowania i ozdabiania — *zakapsułkować zmienne elementy*. Tu zmienia się implementacja systemu okienkowego. Jeśli ukryjemy funkcje takiego systemu w obiekcie, będziemy mogli zaimplementować klasę Window i jej podklasy w kategoriach interfejsu tego obiektu. Ponadto jeśli interfejs będzie zgodny z wszystkimi interesującymi nas systemami okienkowymi, nie będziemy musieli modyfikować klasy Window ani żadnej z jej podklas, aby zapewnić obsługę różnych takich systemów. Aby skonfigurować obiekty tej klasy na

potrzeby wybranego systemu okienkowego, wystarczy przekazać im odpowiedni obiekt kapsułkujący dany system. Dzięki temu okna można skonfigurować nawet w czasie wykonywania programu.

KLASY WINDOW I WINDOWIMP

Zdefiniujmy odrębną hierarchię klas **WindowImp**, w której ukryjemy implementacje różnych systemów okienkowych. **WindowImp** to klasa abstrakcyjna do tworzenia obiektów kapsułkujących kod zależny od systemu okienkowego. Aby edytor Lexi działał w określonym systemie okienkowym, należy skonfigurować każdy obiekt okna za pomocą egzemplarza podklasy klasy **WindowImp** odpowiadającego temu systemowi. Poniższy diagram przedstawia relacje między hierarchiami klas **Window** i **WindowImp**.



Przez ukrycie implementacji w klasach **WindowImp** unikamy uzależniania klas **Window** od systemów okienkowych, dzięki czemu hierarchia klasy **Window** jest stosunkowo mała i stabilna. Jednocześnie możemy łatwo rozbudować hierarchię implementacji, aby dodać obsługę nowych systemów okienkowych.

PODKLASY KLASY WINDOWIMP

Podklasy klasy **WindowImp** przekształcają żądania na operacje specyficzne dla danego systemu okienkowego. Rozważmy przykład z podrozdziału 2.2. Zdefiniowaliśmy w nim operację `Rectangle::Draw` w kategoriach operacji `DrawRect` egzemplarzy klasy **Window**:

```

void Rectangle::Draw (Window* w) {
    w->DrawRect(_x0, _y0, _x1, _y1);
}
  
```

W domyślnej implementacji operacji `DrawRect` użyto abstrakcyjnej operacji do rysowania prostokątów, zadeklarowanej w klasie **WindowImp**:

```

void Window::DrawRect (
    Coord x0, Coord y0, Coord x1, Coord y1
) {
    _imp->DeviceRect(x0, y0, x1, y1);
}
  
```

W tym kodzie `_imp` to zmienna składowa obiektu `Window` przechowująca obiekt `WindowImp` użyty do skonfigurowania danego obiektu `Window`. Implementacja okna jest określona przez egzemplarz podklasy klasy `WindowImp`, na który wskazuje zmienna `_imp`. W klasie `XWindowImp` (czyli w podklasie klasy `WindowImp` przeznaczonej do obsługi systemu okienkowego X) implementacja operacji `DeviceRect` może wyglądać tak:

```
void XWindowImp::DeviceRect (
    Coord x0, Coord y0, Coord x1, Coord y1
) {
    int x = round(min(x0, x1));
    int y = round(min(y0, y1));
    int w = round(abs(x0 - x1));
    int h = round(abs(y0 - y1));
    XDrawRectangle(_dpy, _winid, _gc, x, y, w, h);
}
```

Operacja `DeviceRect` ma taką definicję, ponieważ w `XDrawRectangle` (jest to interfejs systemu X służący do rysowania prostokątów) zdefiniowano prostokąt za pomocą współrzędnych jego lewego dolnego wierzchołka, szerokości i wysokości. Operacja `DeviceRect` musi obliczyć te wartości na podstawie otrzymanych danych. Najpierw określa współrzędne lewego dolnego wierzchołka (ponieważ para (x_0, y_0) może wyznaczać dowolny z czterech rogów), a następnie oblicza szerokość i wysokość prostokąta.

W `PMWindowImp` (jest to podklasa klasy `WindowImp` powiązana z systemem Presentation Manager) operacja `DeviceRect` może mieć inną definicję:

```
void PMWindowImp::DeviceRect (
    Coord x0, Coord y0, Coord x1, Coord y1
) {
    Coord left = min(x0, x1);
    Coord right = max(x0, x1);
    Coord bottom = min(y0, y1);
    Coord top = max(y0, y1);

    PPOINTL point[4];

    point[0].x = left;    point[0].y = top;
    point[1].x = right;   point[1].y = top;
    point[2].x = right;   point[2].y = bottom;
    point[3].x = left;    point[3].y = bottom;

    if (
        (GpiBeginPath(_hps, 1L) == false) ||
        (GpiSetCurrentPosition(_hps, &point[3]) == false) ||
        (GpiPolyLine(_hps, 4L, point) == GPI_ERROR) ||
        (GpiEndPath(_hps) == false)
    ) {
        // Zgłoś błąd.
    } else {
        GpiStrokePath(_hps, 1L, 0L);
    }
}
```

Dlaczego ten kod tak bardzo różni się od wersji dla systemu X? No cóż, system PM — w odróżnieniu od systemu X — nie udostępnia operacji do bezpośredniego rysowania prostokątów. W zamian oferuje ogólniejszy interfejs do określania wierzchołków wielosegmentowych kształtów (tak zwanej **ścieżki**) i rysowania obramowania lub wypełniania danego obszaru.

Implementacja operacji `DeviceRect` dla systemu PM oczywiście znacznie różni się od wersji dla systemu X, jednak nie ma to znaczenia. Klasa `WindowImp` ukrywa różnice między interfejsami systemów za potencjalnie rozbudowanym, ale stabilnym interfejsem. Pozwala to automatem podklas klasy `Window` skoncentrować się na abstrakcji okna, a nie na szczegółach działania systemu okienkowego. Ponadto umożliwia dodanie obsługi nowego systemu tego rodzaju bez modyfikowania klas `Window`.

KONFIGUROWANIE OBIEKTÓW WINDOW ZA POMOCĄ OBIEKTÓW WINDOWIMP

Kluczową kwestią, której jeszcze nie poruszyliśmy, jest sposób konfigurowania okna za pomocą odpowiedniej podklasy klasy `WindowImp`. Ujmijmy to inaczej — kiedy zmienna `_imp` jest inicjowana i kto określa, którego systemu okienkowego (a tym samym której podklasy klasy `WindowImp`) należy użyć? Okno potrzebuje jednego z obiektów `WindowImp`, zanim będzie mogło wykonać jakiekolwiek interesujące operacje.

Istnieje kilka możliwości, jednak tu skoncentrujemy się na rozwiązaniu opartym na wzorcu Fabryka abstrakcyjna (s. 101). Możemy zdefiniować klasę abstrakcyjną fabryki, `WindowSystemFactory`, udostępniającą interfejs do tworzenia różnych zależnych od systemów okienkowych obiektów z implementacją:

```
class WindowSystemFactory {
public:
    virtual WindowImp* CreateWindowImp() = 0;
    virtual ColorImp* CreateColorImp() = 0;
    virtual FontImp* CreateFontImp() = 0;

    // Operacje "Create..." dla wszystkich zasobów systemu okienkowego.
};
```

Teraz możemy zdefiniować konkretną klasę fabryki dla każdego systemu okienkowego:

```
class PMWindowSystemFactory : public WindowSystemFactory {
    virtual WindowImp* CreateWindowImp()
    { return new PMWindowImp; }
    // ...
};

class XWindowSystemFactory : public WindowSystemFactory {
    virtual WindowImp* CreateWindowImp()
    { return new XWindowImp; }
    // ...
};
```

W konstruktorze klasy bazowej Window można wykorzystać interfejs klasy WindowSystemFactory do zainicjowania składowej _imp za pomocą obiektu WindowImp odpowiedniego dla użytego systemu okienkowego:

```
Window::Window () {
    _imp = windowSystemFactory->CreateWindowImp ();
}
```

Zmienna windowSystemFactory to ogólnie dostępny egzemplarz podklasy WindowSystemFactory. Przypomina on ogólnie dostępnyą zmienną guiFactory, która określa standard wyglądu i działania. Zmienną windowSystemFactory można zainicjować w taki sam sposób jak zmienną guiFactory.

WZORZEC MOST

Klasa WindowImp określa interfejs dla standardowych mechanizmów systemów okienkowych, jednak jej projekt podlega innym ograniczeniom niż interfejs klasy Window. Programiści aplikacji nie będą bezpośrednio korzystać z interfejsu klasy WindowImp. Używać będą tylko obiektów Window. Dlatego interfejsu klasy WindowImp nie trzeba dostosowywać do punktu widzenia programisty, co było problemem przy projektowaniu hierarchii klas Window i ich interfejsów. Interfejs klasy WindowImp może ściślej odzwierciedlać to, co systemy okienkowe oczywiście udostępniają. Można w nim zastosować podejście oparte na funkcjach wspólnych lub na ich sumie (w zależności od tego, co lepiej pasuje do docelowego systemu okienkowego).

Ważne jest, aby zauważyc, że interfejs klasy Window jest przeznaczony dla programistów aplikacji, natomiast klasa WindowImp obsługuje systemy okienkowe. Podział możliwości systemów okienkowych na hierarchie klas Window i WindowImp pozwala niezależnie tworzyć implementacje i wyspecjalizowane wersje tych interfejsów. Obiekty z tych hierarchii współpracują ze sobą, aby umożliwić działanie edytora Lexi bez wprowadzania zmian w wielu systemach okienkowych.

Relacja między klasami Window i WindowImp to przykład zastosowania wzorca Most (s. 181). Umożliwia on współpracę odrębnym klasom, nawet jeśli można je modyfikować niezależnie od siebie. Kryteria projektowe doprowadziły w tym przykładzie do utworzenia dwóch hierarchii klas — jednej do obsługi logicznej reprezentacji okien i drugiej do przechowywania różnych implementacji okien. Wzorzec Most umożliwia konserwowanie i wzbogacanie logicznych abstrakcji okien bez konieczności modyfikowania kodu zależnego od systemów okienkowych (i na odwrót).

2.7. DZIAŁANIA UŻYTKOWNIKÓW

Niektóre funkcje edytora Lexi są dostępne poprzez reprezentację dokumentu w trybie WYSIWYG. Użytkownik może wpisywać i usuwać tekst, przenosić punkt wstawiania i zaznaczać fragmenty tekstu przez wskazywanie, klikanie oraz wprowadzanie znaków bezpośrednio w dokumencie. Dostęp do innych funkcji można uzyskać pośrednio — za pomocą operacji w menu rozwijanych edytora Lexi, przycisków i skrótów klawiaturowych. Możliwości edytora obejmują następujące operacje:

- ▶ tworzenie nowego dokumentu;
- ▶ otwieranie, zapisywanie i drukowanie istniejącego dokumentu;

- ▶ wycinanie zaznaczonego tekstu z dokumentu i ponowne wklejanie go;
- ▶ zmienianie czcionki i stylu zaznaczonego tekstu;
- ▶ zmienianie formatowania tekstu (na przykład wyrównania i wyjustowania);
- ▶ zamknięcie aplikacji;
- ▶ i inne.

Edytor Lexi udostępnia różne interfejsy użytkownika do wykonywania tych operacji. Jednak nie chcemy wiązać poszczególnych zadań z konkretnym interfejsem użytkownika, ponieważ możemy zdecydować się na obsługę tej samej operacji za pomocą kilku interfejsów (stronę można zmienić na przykład za pomocą przycisku wyboru strony lub polecenia z menu). Chcemy też zachować możliwość zmodyfikowania interfejsu w przyszłości.

Ponadto wymienione operacje są zaimplementowane w wielu różnych klasach. Jesteśmy autonomicznie implementacji, dlatego chcemy mieć dostęp do funkcji programu bez konieczności tworzenia wielu zależności między klasami implementacji i interfejsu użytkownika. W przeciwnym razie powstanie implementacja ze ścisłymi powiązaniemi, którą trudniej będzie zrozumieć, rozbudowywać i konserwować.

Sytuację dodatkowo komplikuje fakt, że chcemy, aby edytor Lexi udostępniał mechanizmy cofania i powtarzania⁸ operacji dla większości, ale nie dla wszystkich funkcji. W szczególności chcemy umożliwić anulowanie operacji związanych z modyfikacją dokumentu (takich jak usuwanie), których uruchomienie może doprowadzić do przypadkowego zniszczenia przez użytkownika wielu danych. Nie należy jednak próbować cofać takich operacji, jak zapisanie rysunku lub zamknięcie aplikacji. Te funkcje nie powinny być uwzględniane w procesie anulowania. Nie chcemy też ustalać arbitralnego limitu liczby wycofywanych i powtarzanych operacji.

Widać więc, że obsługę operacji wykonywanych przez użytkowników trzeba wbudować w różne części aplikacji. Zadanie polega na wymyśleniu prostego i rozszerzalnego mechanizmu, który spełni wszystkie wymienione wymogi.

KAPSUŁKOWANIE ŻĄDANIA

Z perspektywy projektantów menu rozwijane to po prostu rodzaj glifu zawierającego inne glify. Menu rozwijane od innych glifów zawierających elementy podrzędne odróżnia to, że większość glifów z menu w odpowiedzi na kliknięcie wykonuje pewne zadania.

Załóżmy, że te „robocze” glify to egzemplarze klasy **MenuItem** (jest to podklasa klasy **Glyph**), które wykonują zadania w odpowiedzi na zgłoszenie żądania przez klienta⁹. Obsłuszenie żądania może polegać na uruchomieniu operacji jednego obiektu, wykonaniu wielu zadań przez liczne obiekty lub zastosowaniu pośredniego rozwiązania.

⁸ Czyli ponownego wykonania operacji, która właśnie została cofnięta.

⁹ Teoretycznie klientem jest osoba korzystająca z edytora Lexi, jednak w rzeczywistości jest to inny obiekt (na przykład dyspozytor zdarzeń) zarządzający danymi wejściowymi wprowadzonymi przez użytkownika.

Moglibyśmy zdefiniować podkласę klasy `MenuItem` dla każdej operacji wykonywanej przez użytkownika, a następnie na stałe powiązać każdą z tych podklas z odpowiednim żądaniem. Jednak nie jest to właściwe rozwiązańe. Nie potrzebujemy podklasy klasy `MenuItem` dla każdego żądania, podobnie jak nie musimy tworzyć podklasy dla każdego łańcucha znaków z menu rozwijanego. Ponadto opisane podejście wiąże żądanie z określonym interfejsem użytkownika, co utrudnia obsługę żądania za pomocą innego interfejsu.

Zilustrujmy ten problem. Założymy, że można przejść do ostatniej strony dokumentu zarówno za pomocą obiektu `MenuItem` z menu rozwijanego, jak i przez wcisnięcie ikony strony w dolnej części interfejsu edytora Lexi (jeśli dokument jest krótki, to rozwiązanie może być wygodniejsze). Jeżeli powiążemy żądanie z obiektem `MenuItem` przez dziedziczenie, będziemy musieli zrobić to samo z ikoną strony i innymi widgetami zgłaszającymi to żądanie. Liczba klas, które trzeba utworzyć w tym podejściu, może sięgać iloczynu liczby typów widgetów i liczby żądań.

Brakuje mechanizmu umożliwiającego przekazanie elementom menu parametru określającego żądanie, które należy obsługiwać. Technika ta pozwoli uniknąć tworzenia dużej liczby podklas i zapewnia większą elastyczność w czasie wykonywania programu. Obiekt `MenuItem` można też sparametryzować na podstawie funkcji, którą należy wywołać, jednak to rozwiązanie nie jest kompletne przynajmniej z trzech powodów:

1. Nie rozwiązuje problemu cofania i powtarzania operacji.
2. Trudno jest powiązać stan z funkcją (na przykład funkcja zmieniająca czcionkę musi wiedzieć, której czcionkę ma zastosować).
3. Rozszerzanie funkcji jest trudne, podobnie jak powtórne wykorzystanie ich fragmentów.

Z tych przyczyn uważamy, że do sparametryzowania obiektów `MenuItem` należy zastosować obiekty, a nie funkcje. Następnie można wykorzystać dziedziczenie do rozszerzenia i powtórnego wykorzystania implementacji żądania. Dostępne jest też miejsce na przechowywanie stanu oraz zaimplementowanie funkcji cofania i powtarzania operacji. Oto następny przykład kapsułkowania zmiennych elementów, którymi w tym przypadku są żądania. Każde takie żądanie ukryjemy w obiekcie `Command`.

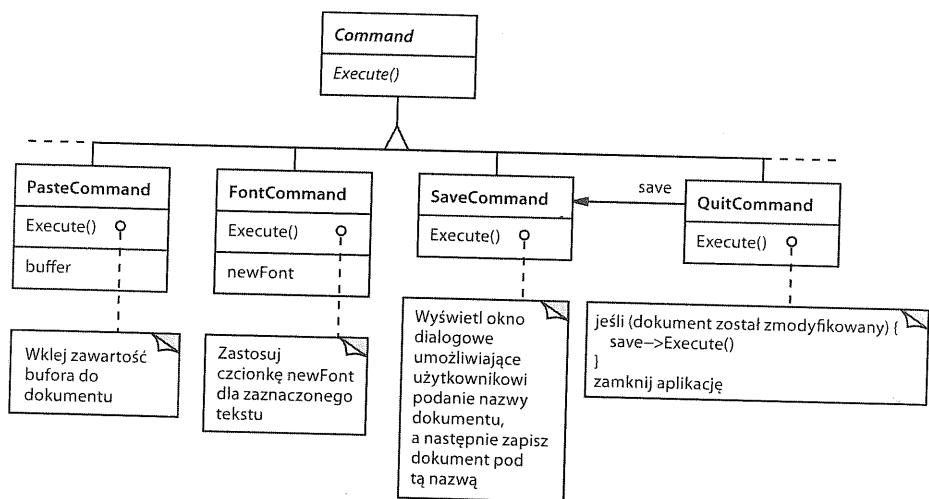
KLASA COMMAND I JEJ PODKLASY

Najpierw zdefiniujmy klasę abstrakcyjną `Command`, aby udostępnić interfejs do zgłaszania żądań. Podstawowy interfejs będzie składał się z pojedynczej operacji abstrakcyjnej o nazwie „Execute”. W podklasach klasy `Command` zaimplementujemy tę operację na różne sposoby w celu dodania obsługi różnych żądań. Niektóre podklasy mogą delegować część zadań (lub nawet wszystkie) do innych obiektów. Pozostałe podklasy mogą mieć możliwość samodzielnej obsługi żądań (rysunek 2.11). Jednak jednostka zgłaszająca żądanie traktuje każdy obiekt `Command` w taki sam sposób.

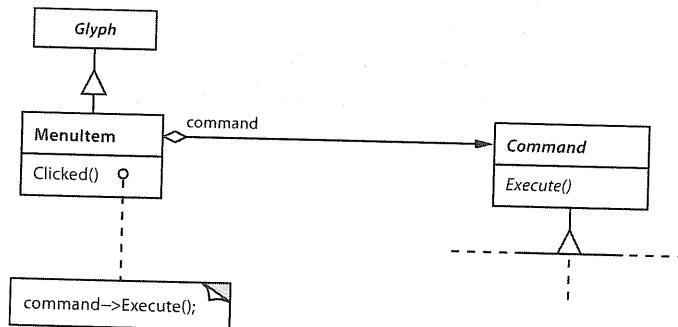
Teraz w obiektach `MenuItem` można umieścić obiekt `Command` kapsułkujący żądanie (rysunek 2.12). Każdemu obiekowi `MenuItem` przekazujemy egzemplarz podklasy klasy `Command` odpowiedni dla danego elementu menu (w podobny sposób określamy tekst wyświetlany w tym elemencie). Kiedy użytkownik wybierze konkretny element menu, obiekt `MenuItem` po prostu wywoła operację `Execute` obiektu `Command`, aby obsłużyć żądanie. Zauważmy, że przyciski i inne widgety mogą korzystać z obiektów `Command` w taki sam sposób, jak robią to elementy menu.

RYSUNEK 2.11.

Fragment
hierarchii klasy
Command

**RYSUNEK 2.12.**

Relacja między
klasami
MenuItem
i Command



MOŻLIWOŚĆ COFANIA OPERACJI

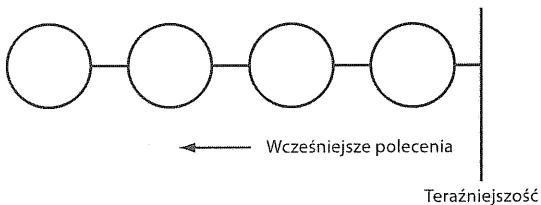
Cofanie i powtarzanie operacji to ważne funkcje interaktywnych aplikacji. Na potrzeby cofania i powtarzania poleceń dodajmy do interfejsu klasy Command operację Unexecute. Będzie ona odwracać efekty działania uruchomionej wcześniej operacji Execute na podstawie informacji zapisanych na potrzeby anulowania przez tę ostatnią. Na przykład dla obiektów FontCommand operacja Execute zapisuje zakres tekstu dotkniętego zmianą czcionki oraz pierwotnie zastosowaną czcionkę. Operacja Unexecute tych obiektów będzie przywracać poprzednią czcionkę w tekście z danego zakresu.

Czasem możliwość cofnięcia operacji trzeba ustalić w czasie wykonywania programu. Żądanie zmiany czcionki zaznaczonego tekstu nie spowoduje żadnych modyfikacji, jeśli do danego fragmentu już wcześniej przypisano wybraną czcionkę. Założymy, że użytkownik zaznacza tekst, a następnie żąda niepotrzebnej zmiany czcionki. Jaki powinien być efekt późniejszego żądania cofnięcia operacji? Czy nieznacząca zmiana powinna powodować wykonanie podobnie zbędnej operacji w odpowiedzi na żądanie anulowania? Prawdopodobnie nie. Jeśli użytkownik kilkakrotnie powtórzy żądanie niepotrzebnej zmiany czcionki, nie powinien musieć cofać operacji tyle samo razy, aby wrócić do ostatniej znaczącej zmiany. Jeśli ogólny efekt wykonania polecenia to brak modyfikacji, nie ma potrzeby obsługiwać odpowiadającego mu żądania anulowania operacji.

Dlatego aby można było ustalić, czy należy cofnąć polecenie, dodamy do interfejsu klasy Command abstrakcyjną operację `Reversible`. Zwraca ona wartość logiczną. W podklasach można ponownie zdefiniować tę operację, aby w zależności od warunków w czasie wykonywania programu zwracała wartość `true` lub `false`.

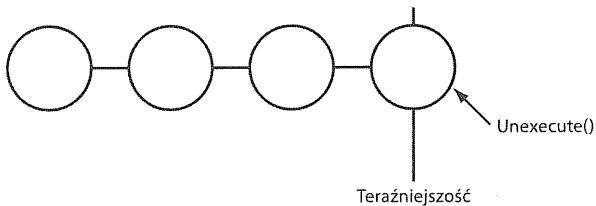
HISTORIA POLECEŃ

Ostatni etap dodawania obsługi cofania i powtarzania dowolnej liczby operacji wymaga zdefiniowania **historii poleceń**, czyli listy wykonanych (lub anulowanych, jeśli niektóre operacje cofnięto) instrukcji. W teorii historia poleceń wygląda tak:

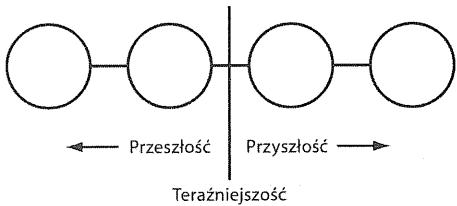


Każdy okrąg reprezentuje obiekt klasy Command. Tu użytkownik wywołał cztery polecenia. Na początku uruchomił polecenie pierwsze od lewej, następnie drugie i tak dalej do ostatnio wywołanej instrukcji (pierwsza od prawej). Linia z opisem „Teraźniejszość” pozwala wskazać ostatnio uruchomione (i cofnięte) polecenie.

Aby cofnąć ostatnią instrukcję, wystarczy wywołać dla niej operację `Unexecute`:

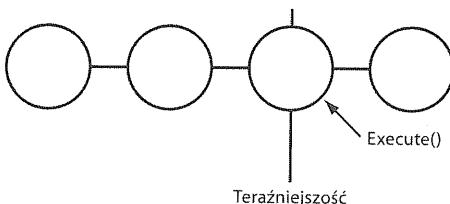


Po cofnięciu polecenia należy przenieść linię „terazniejszości” o jedną instrukcję w lewo. Jeśli użytkownik ponownie anuluje polecenie, w taki sam sposób cofnięta zostanie kolejna ostatnio wykonana instrukcja, a program wejdzie w stan przedstawiony poniżej:

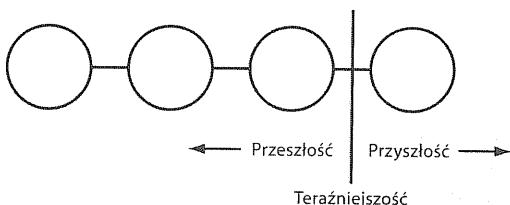


Zauważmy, że przez proste powtarzanie tej procedury można uzyskać wiele poziomów cofania instrukcji. Liczba tych poziomów jest ograniczona tylko długością historii poleceń.

Aby powtórzyć anulowaną instrukcję, należy wykonać te same operacje w odwrotnej kolejności. Polecenia na prawo od linii „terażniejszości” to instrukcje, które można w przyszłości powtórzyć. Aby ponownie wykonać ostatnie cofnięte polecenie, należy wywołać operację Execute dla instrukcji na prawo od linii „terażniejszości”:



Następnie trzeba przesunąć linię „terażniejszości”, aby przy następnych powtórzeniach ponownie wykonać kolejne polecenie.



Oczywiście jeśli kolejną operacją nie będzie następne powtórzenie, ale cofnięcie, polecenie na lewo od linii „terażniejszości” zostanie anulowane. Dlatego użytkownik może poruszać się w czasie do przodu i do tyłu (w zależności od potrzeby), aby naprawić błędy.

WZORZEC POLECENIE

Polecenia w edytorze Lexi to przykład zastosowania wzorca Polecenie (s. 302), który pokazuje, jak kapsułkować żądania. Wzorzec ten pozwala utworzyć jednolity interfejs do zgłoszenia żądań, umożliwiający konfigurację klientów pod kątem obsługi różnych żądań. Ten interfejs odziela klienty od implementacji obsługi żądania. Polecenie może delegować dowolną część implementacji obsługi żądania do innych obiektów lub samo wykonywać wszystkie operacje. Jest to doskonałe rozwiązanie dla aplikacji takich jak Lexi, w których potrzebny jest skoncentrowany dostęp do funkcji rozproszonych po programie. Wzorzec ten opisuje także mechanizmy cofania i powtarzania oparte na podstawowym interfejsie klasy Command.

2.8. SPRAWDZANIE PISOWNI I PODZIAŁ SŁÓW

Ostatni problem projektowy dotyczy analizy tekstu, a konkretnie sprawdzania błędów w pisowni i dodawania w odpowiednich miejscach punktów podziału słów, niezbędnych do poprawnego sformatowania tekstu.

Ograniczenia związane z tym zadaniem są podobne do tych dotyczących projektu mechanizmów formatowania (podrozdział 2.3). Sprawdzanie pisowni i podział słów można — podobnie jak w przypadku strategii podziału na wiersze — zrealizować na więcej niż jeden sposobów.

Dlatego także tu chcemy dodać obsługę wielu algorytmów. Zróżnicowany zestaw algorytmów może umożliwiać znalezienie równowagi między wykorzystaniem pamięci, czasem i jakością formatowania. Powinniśmy także sprawić, aby dodawanie nowych algorytmów było łatwe.

Ponadto chcemy uniknąć wiązania omawianych funkcji ze strukturą dokumentu. Ten cel jest tu jeszcze ważniejszy niż przy formatowaniu, ponieważ sprawdzanie pisowni i podział słów to tylko dwie z potencjalnie wielu analiz, które możemy wbudować w edytor Lexi. W przyszłości z pewnością zechcemy rozbudować możliwości analityczne aplikacji. Możemy dodać wyszukiwanie, zliczanie słów, mechanizm do sumowania wartości w tabelach, sprawdzanie gramatyki itd. Nie chcemy jednak modyfikować klasy `Glyph` i wszystkich jej podklas po dodaniu każdej nowej funkcji tego rodzaju.

Ta układanka składa się z dwóch części. Są to: (1) uzyskanie dostępu do analizowanych informacji rozproszonych po strukturze dokumentu w różnych glifach i (2) przeprowadzenie analiz. Rozważmy obie te kwestie osobno.

DOSTĘP DO ROZPROSZONYCH INFORMACJI

Wiele rodzajów analiz wymaga zbadania tekstu znak po znaku. Jest on rozproszony po hierarchicznej strukturze obiektów-glifów. Do analizy tekstu potrzebny jest mechanizm dostępu z informacjami o strukturach danych, w których zapisane są obiekty. Niektóre glify mogą przechowywać elementy podzielone w listach powiązanych, inne mogą wykorzystywać tablice lub mniej standardowe struktury danych. Mechanizm dostępu musi potrafić obsługiwać wszystkie takie możliwości.

Sytuację dodatkowo komplikuje to, że różne analizy wymagają odmiennych sposobów dostępu do danych. W większości analiz tekst jest sprawdzany od początku do końca. Jednak niektóre moduły działają w odwrotną stronę. Na przykład przy wyszukiwaniu „w górę” trzeba przejść przez tekst od końca, a nie od początku. Analiza wyrażeń algebraicznych może wymagać przechodzenia metodą inorder.

Dlatego w mechanizmie dostępu trzeba uwzględnić różne struktury danych, a także zapewnić obsługę wielu sposobów poruszania się po nich (na przykład metodą preorder, inorder i postorder).

KAPSUŁKOWANIE DOSTĘPU DO DANYCH I PRZECHODZENIA PO NICH

Obecnie w interfejsie klasy `Glyph` do wskazywania elementów podzielonych służy klientom indeks całkowitoliczbowy. Choć jest to uzasadnione dla klas z rodzinie `Glyph` przechowujących takie elementy w tablicy, rozwiązanie to jest niewydajne dla glifów z listami powiązanymi. Ważnym zadaniem abstrakcji glifu jest ukrywanie struktur danych przechowujących elementy podzielone. Dzięki temu w klasie z rodzinie `Glyph` można zmienić strukturę danych bez wpływu na inne klasy.

Dlatego tylko glify mogą znać używaną strukturę danych. Skutkiem ubocznym tego podejścia jest to, że interfejsu glifów nie należy rozwijać pod kątem tej lub innej struktury. Interfejs nie powinien być dostosowany w większym stopniu na przykład — jak ma to obecnie miejsce — do tablic niż do list powiązanych.

Możemy za pomocą jednej techniki rozwiązać ten problem i zapewnić obsługę różnych sposobów przechodzenia po danych. Można umieścić wiele możliwości dostępu i poruszania się bezpośrednio w klasach z rodziny `Glyph` oraz dodać sposób wyboru mechanizmu — na przykład przez podanie stałej wyliczeniowej jako parametru. Klasy powinny przekazywać ten parametr w czasie przechodzenia po danych, aby zagwarantować, że wszystkie stosują tę samą metodę. Klasy muszą też przekazywać wszystkie informacje zebrane w czasie poruszania się po danych.

Aby zapewnić obsługę tego rozwiązania, możemy dodać do interfejsu klasy `Glyph` następujące operacje abstrakcyjne:

```
void First(Traversal kind)
void Next()
bool IsDone()
Glyph* GetCurrent()
void Insert(Glyph*)
```

Operacje `First`, `Next` i `IsDone` kontrolują przechodzenie po danych. Operacja `First` inicjuje ten proces. Pobiera sposób poruszania się jako parametr typu `Traversal`. Typ ten obejmuje stałe wyliczeniowe przyjmujące wartości `CHILDREN` (przechodzenie tylko po bezpośrednich elementach podrzędnych glifu), `PREORDER` (przechodzenie po całej strukturze metodą preorder), `POSTORDER` i `INORDER`. Operacja `Next` powoduje przejście do następnego glifu, a `IsDone` informuje, czy przechodzenie zostało zakończone czy nie. Operacja `GetCurrent` zastępuje operację `Child` i zapewnia dostęp do obecnie wybranego glifu. Operacja `Insert` zastępuje dawną operację i wstawia otrzymany glif na bieżącej pozycji.

Na potrzeby analiz można wykorzystać poniższy kod w języku C++ do przechodzenia metodą preorder po strukturze glifów z elementem głównym g:

```
Glyph* g;

for (g->First(PREORDER); !g->IsDone(); g->Next()) {
    Glyph* current = g->GetCurrent();

    // Przeprowadzanie analiz.
}
```

Warto zauważyć, że usunęliśmy z interfejsu glifów indeks całkowitoliczbowy. Interfejs ten nie obejmuje już nic, co ułatwia korzystanie z tej lub innej kolekcji kosztem pozostałych. Ponadto wyeliminowaliśmy konieczność implementacji w klientach standardowych metod przechodzenia po elementach.

Jednak także to podejście nie jest pozbawione wad. Po pierwsze, nie będzie obsługiwać nowych metod przechodzenia po elementach, jeśli nie rozbudujemy zbioru wartości wyliczeniowych lub nie dodamy następnych operacji. Założymy, że chcemy zastosować odmianę poruszania się metodą preorder z automatycznym pominięciem glifów nietekstowych. Wymaga to zmiany wyliczenia `Traversal` przez umieszczenie w nim wartości w rodzaju `TEXTUAL_PREORDER`.

Chcemy uniknąć zmieniających istniejących deklaracji. Umieszczenie mechanizmu przechodzenia po elementach w całości w hierarchii klas **Glyph** utrudnia modyfikowanie lub rozbudowywanie go bez wprowadzania zmian w wielu innych klasach. Niełatwo jest też powtórnie wykorzystać ten mechanizm do poruszania się po innych strukturach obiektów. Ponadto nie można jednocześnie przechodzić po strukturze na więcej niż jeden sposobów.

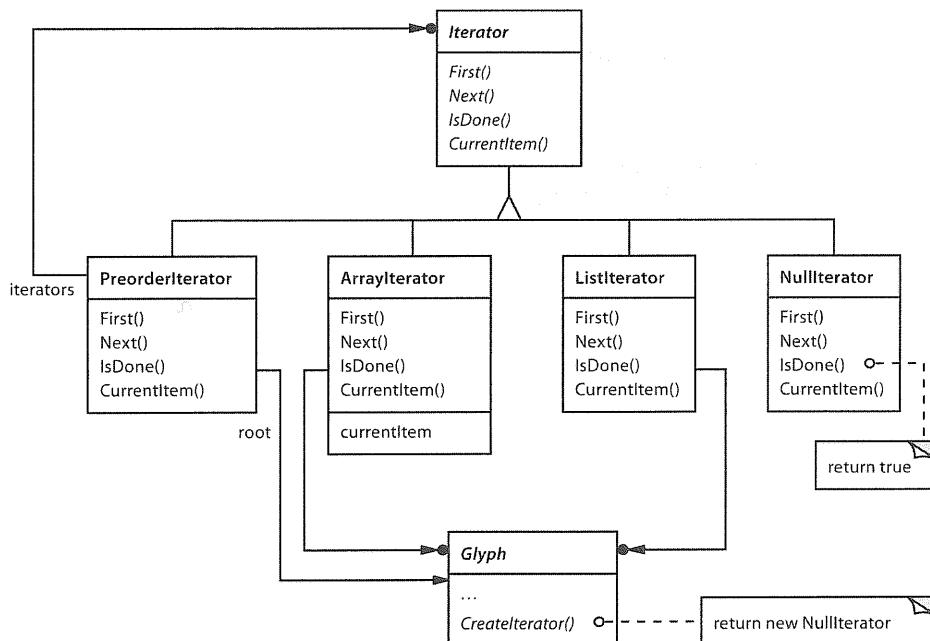
Także tu lepszym rozwiązaniem jest zakapsułkowanie zmiennych elementów. W tym przypadku są to mechanizmy dostępu do elementów i przechodzenia po nich. Można dodać klasę obiektów nazywanych **iteratorami**, której jedynym przeznaczeniem jest określanie różnych zestawów takich mechanizmów. Za pomocą dziedziczenia można uzyskać dostęp do różnych struktur danych w jednolity sposób, a także dodać obsługę nowych metod przechodzenia po elementach. Nie trzeba przy tym zmieniać interfejsów glifów lub modyfikować ich istniejących implementacji.

KLASA ITERATOR I JEJ PODKLASY

Do zdefiniowania ogólnego interfejsu na potrzeby dostępu do elementów i przechodzenia po nich użyjemy klasy abstrakcyjnej o nazwie **Iterator**. W klasach konkretnych, takich jak **ArrayIterator** i **ListIterator**, zaimplementujemy ten interfejs, aby zapewnić dostęp do tablic i list. W klasach **PreorderIterator**, **PostorderIterator** i podobnych zaimplementujemy różne sposoby poruszania się po określonych strukturach. Każda podklasa klasy **Iterator** obejmuje referencję do struktury, po której się porusza. Referencja ta jest używana do inicjowania egzemplarzy tych podklas. Rysunek 2.13 przedstawia klasę **Iterator** oraz kilka jej podklas. Warto zauważyć, że w interfejsie klasy **Glyph** umieściliśmy abstrakcyjną operację **CreateIterator()**, aby dodać obsługę iteratorów.

RYSUNEK 2.13.

Klasa Iterator
i jej podklasy



Interfejs klasy `Iterator` udostępnia operacje `First`, `Next` i `IsDone` służące do kontrolowania przechodzenia po elementach. W klasie `ListIterator` operację `First` zaimplementowaliśmy tak, aby wskazywała pierwszy element listy. Operacja `Next` powoduje przejście iteratora do następnego elementu listy, a operacja `IsDone` informuje, czy wskaźnik listy wskazuje pozycję za ostatnim jej elementem. Operacja `CurrentItem` przeprowadza dereferencję iteratora, aby zwrócić `glif`, na który wskazuje referencja. Klasa `ArrayIterator` wykonuje podobne zadania, ale na tablicy `glifów`.

Teraz można uzyskać dostęp do elementów podrzędnych struktury `glifu` bez znajomości jej reprezentacji:

```
Glyph* g;
Iterator<Glyph*>* i = g->CreateIterator();

for (i->First(); !i->IsDone(); i->Next()) {
    Glyph* child = i->CurrentItem();

    // Wykonywanie operacji na bieżącym elemencie podrzędnym.
}
```

Operacja `CreateIterator` zwraca domyślnie egzemplarz klasy `NullIterator`. Jest to okrojony iterator przeznaczony dla glifów pozbawionych elementów podrzędnych (czyli dla liści). Operacja `IsDone` klasy `NullIterator` zawsze zwraca wartość `true`.

W podklasach glifów mających elementy podrzędne operacja `CreatorIterator` jest przesłonięta i zwraca egzemplarz innej podklasy klasy `Iterator`. To, która podklasa to będzie, zależy od struktury przechowującej elementy podrzędne. Jeśli w podklasie `Row` klasy `Glyph` elementy podrzędne znajdują się w liście `_children`, operacja `CreateIterator` może wyglądać tak:

```
Iterator<Glyph*>* Row::CreateIterator () {
    return new ListIterator<Glyph*>(_children);
}
```

W iteratorach przechodzących po elementach w porządku `preorder` i `inorder` metody poruszania się są zaimplementowane za pomocą iteratorów specyficznych dla glifów. Iteratory potrzebne do obsługi tych metod są określone w glifie głównym struktury, po której porusza się iterator. Wywołują one operację `CreateIterator` dla glifów z tej struktury i wykorzystują stos do śledzenia utworzonych w ten sposób iteratorów.

Na przykład klasa `PreorderIterator` pobiera z glifu głównego iterator, inicjuje go, aby wskazywał pierwszy element, a następnie umieszcza iterator na stosie:

```
void PreorderIterator::First () {
    Iterator<Glyph*>* i = _root->CreateIterator();

    if (i) {
        i->First();
        _iterators.RemoveAll();
        _iterators.Push(i);
    }
}
```

Operacja `CurrentItem` po prostu wywołuje tę samą operację dla iteratora znajdującego się na szczytce stosu:

```
Glyph* PreorderIterator::CurrentItem () const {
    return
        _iterators.Size() > 0 ?
            _iterators.Top()->CurrentItem() : 0;
}
```

Operacja `Next` pobiera iterator ze szczytu stosu i żąda od bieżącego elementu utworzenia iteratora. Ma to służyć do maksymalnego zagłębiania się w strukturę glifu (w końcu jest to przechodzenie metodą `preorder`). Operacja `Next` ustawia nowy iterator na pierwszy element w kolejności przechodzenia i umieszcza ten iterator na stosie. Następnie operacja `Next` sprawdza ostatni iterator. Jeśli jego operacja `IsDone` zwróci `true`, przechodzenie po obecnym poddrzewie (lub liściu) zakończyło się. Wtedy operacja `Next` zdejmuję iterator ze szczytu stosu i powtarza cały proces do czasu znalezienia następnych elementów, po których iterator jeszcze nie przeszedł (jeśli takie istnieją). Jeżeli takich elementów nie ma, oznacza to, że zakończono przechodzenie po strukturze.

```
void PreorderIterator::Next () {
    Iterator<Glyph*>* i =
        _iterators.Top()->CreateIterator();

    i->First();
    _iterators.Push(i);

    while {
        _iterators.Size() > 0 && _iterators.Top()->IsDone()
    } {
        delete _iterators.Pop();
        _iterators.Top()->Next();
    }
}
```

Warto zauważyć, że hierarchia klasy `Iterator` umożliwia dodawanie nowych metod przechodzenia po elementach bez konieczności modyfikowania klas glifów. Wystarczy utworzyć podkласę klasy `Iterator` i dodać nowy sposób poruszania się, jak zrobiliśmy to w przypadku podklasy `PreorderIterator`. Podklasy klasy `Glyph` stosują ten sam interfejs, aby zapewnić klientom dostęp do elementów podległych bez ujawniania struktury danych, w której elementy te są zapisane. Ponieważ iteratory przechowują własną kopię stanu procesu przechodzenia, równolegle może działać wiele procesów poruszania się — nawet po tej samej strukturze. Ponadto choć w tym przykładzie iteratory przechodzą po strukturach glifów, nic nie przeszkadza w sparametryzowaniu klasy w rodzaju `PreorderIterator` za pomocą typu obiektu z takiej struktury (w języku C++ użylibyśmy do tego szablonów). Następnie można powtórnie wykorzystać mechanizmy z klasy `PreorderIterator` do poruszania się po innych strukturach.

WZORZEC ITERATOR

Wzorzec Iterator (s. 230) obejmuje techniki dostępu do struktur obiektów i poruszania się po nich. Można go stosować nie tylko do struktur złożonych, ale też do kolekcji. Wzorzec ten wymaga utworzenia abstrakcji algorytmu przechodzenia po elementach i oddziela klienty od wewnętrznej struktury obiektów, po których należy przejść. Wzorzec Iterator ponownie pokazuje, że kapsułkowanie zmiennych elementów pomaga zapewnić elastyczność rozwiązania i umożliwia jego powtórne wykorzystanie. Problem iteracji jest jednak zaskakująco złożony, a ze wzorcem Iterator wiąże się o wiele więcej niuansów oraz korzyści i kosztów, które należy zrównoważyć, niż opisaliśmy to w tym miejscu.

PRZECHODZENIE I DZIAŁANIA WYKONYWANE W JEGO TRAKCIE

Skoro określiliśmy już sposób poruszania się po strukturze glifu, pora utworzyć metody do sprawdzania pisowni i podziału słów. Obie te analizy wymagają rejestrowania informacji w czasie przechodzenia po elementach.

Najpierw musimy zdecydować, które obiekty powinny odpowiadać za analizy. Zadanie to można przydzielić klasom `Iterator` i sprawić w ten sposób, że analizy będą integralną częścią poruszania się po danych. Jednak większą elastyczność i potencjał w zakresie powtórnego wykorzystania rozwiązania zapewnia oddzielenie procesu przechodzenia po elementach od działań wykonywanych w jego trakcie. Dzieje się tak, ponieważ różne analizy często wymagają tego samego procesu poruszania się. Dlatego możemy wielokrotnie wykorzystać ten sam zestaw iteratorów na potrzeby różnych analiz. Na przykład przechodzenie metodą `preorder` jest stosowane w wielu analizach, w tym przy sprawdzaniu pisowni, podziale słów, wyszukiwaniu „w dół” i zliczaniu słów.

Dlatego analizy i poruszanie się po elementach należy rozdzielić. W których jeszcze obiektach można umieścić przeprowadzanie analiz? Wiemy, że potrzebnych może być wiele rodzajów analiz. Każdy z nich wymaga wykonania innych operacji w różnych miejscach procesu przechodzenia po elementach. W poszczególnych analizach niektóre glify są ważniejsze od pozostały. W czasie sprawdzania pisowni lub podziału słów edytor powinien uwzględnić glify znaków, a nie glify graficzne (na przykład linie lub bitmapy). Przy separacji kolorów ważne są widoczne glify. Z pewnością w różnych analizach uwzględnić będziemy odmienne glify.

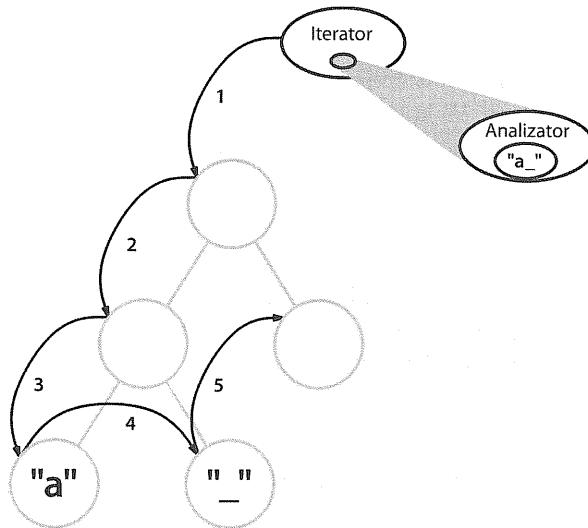
Dlatego w poszczególnych analizach musi istnieć możliwość rozróżniania glifów. Oczywistym rozwiązaniem jest umieszczenie mechanizmów analitycznych w samych klasach glifów. Na potrzeby każdej analizy można wtedy dodać jedną lub kilka operacji abstrakcyjnych do klasy `Glyph` i zaimplementować je w podklasach zgodnie z rolą, jaką operacje te pełnią w poszczególnych analizach.

Wadą tego podejścia jest konieczność zmodyfikowania każdej klasy glifu po dodaniu analiz nowego rodzaju. W niektórych sytuacjach można złagodzić ten problem. Jeśli analizy dotyczą niewielu klas (lub w większości klas działają w ten sam sposób), można w klasie `Glyph` umieścić domyślną implementację abstrakcyjnej operacji. Ta domyślna wersja powinna obsługiwać standardowe przypadki. W ten sposób ograniczymy zmiany do klasy `Glyph` i nietypowych podklas.

Jednak choć implementacja domyślna zmniejsza liczbę koniecznych zmian, do rozwiązania pozostaje trudny do zauważenia problem — każda nowa funkcja analityczna powoduje powiększenie interfejsu klasy `Glyph`. Z czasem operacje analityczne spowodują, że podstawowy interfejs tej klasy przestanie być czytelny. Trudno będzie dostrzec, że głównym przeznaczeniem klasy `Glyph` jest definiowanie i określanie struktury obiektów mających wygląd i kształt. Ten aspekt interfejsu nie będzie widoczny wśród wielu innych elementów.

KAPSUŁKOWANIE ANALIZ

Wszystko wskazuje na to, że musimy ukryć analizy w odrębnym obiekcie, podobnie jak robiliśmy to już wielokrotnie. Możemy umieścić mechanizmy do obsługi poszczególnych analiz w osobnych klasach i wykorzystać egzemplarze tych klas w połączeniu z odpowiednim iteratorem. Iteratorem „przenosiły” wtedy wspomniane egzemplarze do każdego glifu w strukturze, aby obiekt obsługujący analizę mógł przeprowadzić jej fragment dla każdego elementu podczas poruszania się po nich. W tym rozwiążaniu jednostka analizująca powinna rejestrować potrzebne jej informacje (tu są to znaki) w czasie przechodzenia po elementach:



Podstawowe pytanie związane z tym podejściem dotyczy tego, jak obiekt obsługujący analizę ma rozróżniać glify bez uciekania się do sprawdzania typów lub rzutowania ich w dół. Nie chcemy, aby klasa `SpellingChecker` obejmowała (pseudo)kod podobny do poniższego:

```
void SpellingChecker::Check (Glyph* glyph) {
    Character* c;
    Row* r;
    Image* i;

    if (c = dynamic_cast<Character*>(glyph)) {
        // Analiza znaku.

    } else if (r = dynamic_cast<Row*>(glyph)) {
        // Przygotowanie do analizy elementów podrzędnych obiektu r.
```

```

    } else if (i = dynamic_cast<Image*>(glyph)) {
        // Nie wykonyje żadnych operacji.
    }
}

```

Ten kod jest nieelegancki. Zastosowano w nim dość wymyślne operacje, takie jak rzutowanie bezpieczne ze względu na typ. Ponadto trudno go rozszerzać. Przy wprowadzeniu każdej zmiany w hierarchii klasy Glyph trzeba będzie pamiętać o zmodyfikowaniu ciała tej funkcji. Języki obiektowe powstały właśnie po to, aby wyeliminować kod tego rodzaju.

Chcemy uniknąć siłowego podejścia, ale jak to zrobić? Zastanówmy się, co się stanie, kiedy dodamy do klasy Glyph poniższą operację abstrakcyjną:

```
void CheckMe(SpellingChecker&)
```

Operację CheckMe należy zdefiniować w każdej podklasie klasy Glyph w następujący sposób:

```

void GlyphSubclass::CheckMe (SpellingChecker& checker) {
    checker.CheckGlyphSubclass(this);
}

```

W tym kodzie fragment GlyphSubclass należy zastąpić nazwą właściwej podklasy klasy Glyph. Warto zauważyc, że w momencie wywołania operacji CheckMe wiadomo, jakiej podklasy klasy Glyph użyto. W końcu CheckMe to jedna z operacji tej podklasy. Z kolei interfejs klasy SpellingChecker obejmuje operację do sprawdzania każdej podklasy klasy Glyph¹⁰:

```

class SpellingChecker {
public:
    SpellingChecker();

    virtual void CheckCharacter(Character*) ;
    virtual void CheckCharacter(Character*) ;
    virtual void CheckImage(Image*) ;

    // ...i tak dalej.

    List<char*>& GetMisspellings();

protected:
    virtual bool IsMisspelled(const char*);

private:
    char _currentWord[MAX_WORD_SIZE];
    List<char*> _misspellings;
};

```

¹⁰ Ponieważ parametry odróżniają użyte tu funkcje składowe, mogliśmy zastosować przeciążanie, aby każda z nich miała taką samą nazwę. Nadaliśmy im odmienne nazwy, aby ułatwić ich odróżnienie (zwłaszcza w miejscach ich wywołania).

W klasie SpellingChecker operacja sprawdzania glifów klasy Character może wyglądać tak:

```
void SpellingChecker::CheckCharacter (Character* c) {
    const char ch = c->GetCharCode();

    if (isalpha(ch)) {
        // Dodać znaku z alfabetu do zmiennej _currentWord.

    } else {
        // Dojście do znaku spoza alfabetu.

        if (IsMisspelled(_currentWord)) {
            // Dodawanie zmiennej _currentWord do kolekcji _misspellings.
            _misspellings.Append(strdup(_currentWord));
        }

        _currentWord[0] = '\0';
        // Zerowanie zmiennej _currentWord w celu sprawdzenia następnego słowa.
    }
}
```

Warto zauważyć, że specjalną operację GetCharCode zdefiniowaliśmy tylko dla klasy Character. Obiekt SpellingChecker potrafi obsługiwać operacje specyficzne dla podklas bez sprawdzania lub rzutowania typów. Pozwala to traktować obiekty w specjalny sposób.

Operacja CheckCharacter łączy litery w buforze _currentWord. Kiedy natrafi na znak spoza alfabetu, na przykład na podkreślenie, korzysta z operacji IsMisspelled do sprawdzenia pisowni słowa z bufora _currentWord¹¹. Jeśli wyraz jest błędnie napisany, operacja CheckCharacter dodaje go do innych takich słów. Następnie opróżnia bufor _currentWord, aby przygotować go na tworzenie nowego wyrazu. Po zakończeniu przechodzenia po tekście można za pomocą operacji GetMisspellings pobrać listę błędnie napisanych słów.

Teraz można przejść po strukturze glifów i wywołać dla każdego z nich operację CheckMe z obiektem SpellingChecker jako argumentem. W efekcie powoduje to wskazanie każdego glifu obiektowi SpellingChecker i prowadzi do wykonania przez niego następnego kroku w procesie sprawdzania pisowni.

```
SpellingChecker spellingChecker;
Composition* c;

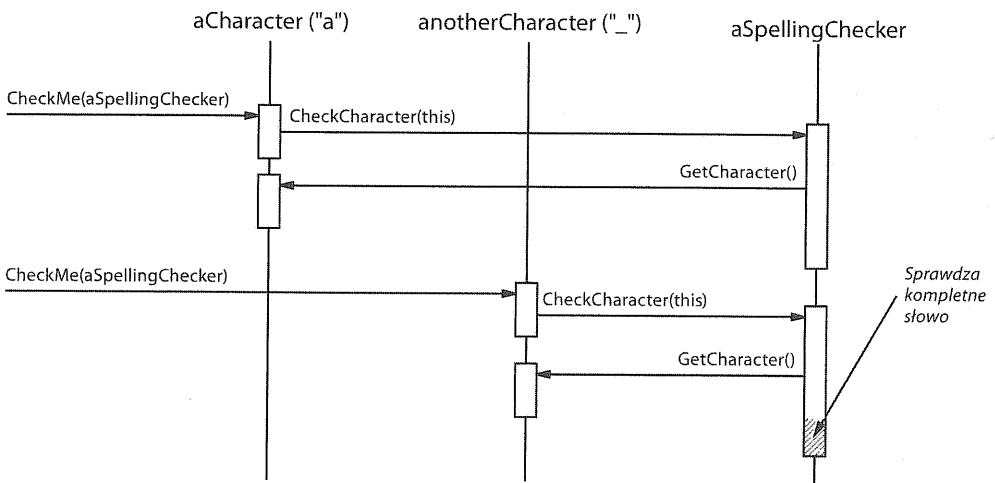
// ...

Glyph* g;
PreorderIterator i(c);

for (i.First(); !i.IsDone(); i.Next()) {
    g = i.CurrentItem();
    g->CheckMe(spellingChecker);
}
```

¹¹ Operacja IsMisspelled obejmuje implementację algorytmu do sprawdzania pisowni, którego nie omawiamy szczegółowo w tym miejscu, ponieważ jest niezależny od projektu edytora Lexi. Obsługę różnych algorytmów można dodać przez utworzenie podklas klasy SpellingChecker. Inna możliwość to zastosowanie wzorca Strategia (s. 315), jak zrobiliśmy to w przypadku formatowania w podrozdziale 2.3.

Poniższy diagram interakcji ilustruje współdziałanie glifów Character i obiektu SpellingChecker.



To podejście pozwala znaleźć błędy w pisowni, jak jednak ma pomóc nam w obsłudze wielu rodzajów analiz? Wygląda na to, że dla każdego nowego typu analiz trzeba dodać do klasy Glyph i jej podklas operację podobną do CheckMe(SpellingChecker&). Jest to prawda, jeśli zdecydujemy się na tworzenie *niezależnych* klas dla poszczególnych analiz. Jednak można też przygotować wspólny interfejs dla *wszystkich* klas obsługujących analizy. Pozwoli to korzystać z tych klas polimorficznie, co oznacza, że można zastąpić specyficzne dla analiz operacje (na przykład CheckMe(SpellingChecker&)) niezależną operacją przyjmującą ogólniejszy parametr.

KLASA VISITOR I JEJ PODKLASY

Będziemy używać nazwy **visitor** (czyli odwiedzający) do określania klas obiektów, które „odwiedzają” inne obiekty w czasie przechodzenia po nich i wykonują odpowiednie operacje¹². Tu możemy utworzyć klasę Visitor z definicją abstrakcyjnego interfejsu służącego do „odwiedzania” glifów z danej struktury.

```

class Visitor {
public:
    virtual void VisitCharacter(Character*) { }
    virtual void VisitRow(Row*) { }
    virtual void VisitImage(Image*) { }

    // ... i tak dalej.
};
  
```

Konkretnie podklasy klasy Visitor wykonują różne analizy. Można na przykład utworzyć klasę SpellingCheckingVisitor do sprawdzania pisowni i podklassę HyphenationVisitor do podziału słów. Implementacja klasy SpellingCheckingVisitor może przypominać wcześniejszą

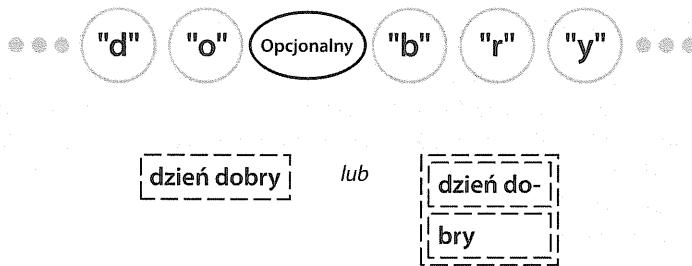
¹² „Odwiedzanie” to tylko nieco ogólniejsza nazwa na „analizowanie”. Jest to wstęp do terminologii stosowanej we wzorcu projektowym, do którego zmierzamy.

implementację klasy SpellingChecker. Różnicą będą nazwy operacji, które w nowej klasie powinny odzwierciedlać ogólniejszy interfejs klasy Visitor. Na przykład operacja CheckCharacter zostanie nazwana VisitCharacter.

Ponieważ określenie CheckMe (czyli sprawdź mnie) nie jest właściwe dla odwiedzających, którzy niczego nie sprawdzają, nadajmy tej operacji ogólniejszą nazwę — Accept. Trzeba też zmienić argument operacji. Powinna ona przyjmować argument Visitor&, co odzwierciedla fakt, że operacja potrafi obsługiwać każdego nowego odwiedzającego. Teraz dodanie nowej analizy wymaga tylko zdefiniowania nowej podklasy klasy Visitor — nie trzeba modyfikować żadnych klas glifów. Obsługę wszystkich analiz dodanych w przyszłości zapewnimy przez umieszczenie tej jednej operacji w klasie Glyph i jej podkласach.

Pokazaliśmy już, jak przebiega sprawdzanie pisowni. W klasie HyphenationVisitor rejestrowanie tekstu odbywa się podobnie. Jednak kiedy operacja VisitCharacter klasy HyphenationVisitor złoży całe słowo, wykona inne działania. Zamiast sprawdzać pisownię, uruchomi algorytm podziału słów, aby ustalić potencjalne miejsca podziału w danym wyrazie. Następnie w każdym z tych punktów wstawi opcjonalny glif. Takie glify to egzemplarze klasy Discretionary (jest to podkلاsa klasy Glyph).

Opcjonalny glif w zależności od tego, czy jest ostatnim znakiem w wierszu czy nie, przyjmuje jedną z dwóch możliwych reprezentacji wizualnych. Jeżeli jest znakiem końcowym, wygląda jak kreska. Jeśli znajduje się w innym miejscu, w ogóle nie jest widoczny. Takie glify sprawdzają w elemencie nadzrędnym (obiekcie Row), czy są jego ostatnim elementem podrzędnym. Operację tę wykonują za każdym razem, kiedy otrzymają żądanie wyświetlenia się lub sprawdzenia wymiarów. Strategia formatowania traktuje glify tego rodzaju tak samo jak odstępy i uznaje je za możliwe zakończenie wiersza. Poniższy diagram pokazuje, jak może wyglądać zagnieżdżony glif opcjonalny.



WZORZEC ODWIEDZAJĄCY

Opisaliśmy właśnie wzorzec Odwiedzający (s. 280). Omówione wcześniej klasa Visitor i jej podklasy to kluczowe elementy tego wzorca. Ujmuje on technikę zastosowaną tu do umożliwienia obsługi dowolnej liczby rodzajów analizy struktur glifów bez konieczności modyfikowania samych klas glifów. Inną cenną cechą obiektów odwiedzających jest to, że można je zastosować nie tylko do obiektów złożonych, takich jak struktury glifów, ale też do *dowolnej* struktury obiektowej. Może to być na przykład zbiór, lista, a nawet acykliczny graf skierowany. Ponadto klasy, które może odwiedzić odwiedzający, nie muszą być powiązane ze sobą za pomocą wspólnej klasy nadzrędnej. Oznacza to, że odwiedzający mogą współdziałać z różnymi hierarchiami klas.

Przed zastosowaniem wzorca Odwiedzający należy zadać sobie ważne pytanie: „Które hierarchie klas zmieniają się najczęściej?”. Wzorzec ten jest najprzydatniejszy, kiedy programista chce wykonywać różne operacje na obiektach ze stabilnej struktury klas. Dodanie nowego odwiedzającego nie wymaga wprowadzania zmian w tej strukturze, co jest szczególnie istotne, jeśli jest ona rozbudowana. Jednak za każdym razem, kiedy do struktury dodawana jest nowa podklasa, trzeba zaktualizować wszystkie interfejsy z rodziny *Visitor* przez dołączenie do nich operacji *Visit...* dla tej podklasy. W omawianym przykładzie oznacza to, że dodanie nowej podklasy klasy *Glyph*, *Foo*, wymaga dołączenia do klasy *Visitor* i wszystkich jej podklas operacji *VisitFoo*. Jednak w kontekście projektu edytora Lexi dużo prawdopodobniejsze jest to, że dodamy nowy rodzaj analiz, a nie nową podkласę klasy *Glyph*. Dlatego wzorzec Odwiedzający dobrze pasuje do potrzeb tego programu.

2.9. PODSUMOWANIE

W projekcie edytora Lexi zastosowaliśmy osiem różnych wzorców:

1. wzorzec Kompozyt (s. 170) do reprezentacji fizycznej struktury dokumentu;
2. wzorzec Strategia (s. 321), aby umożliwić zastosowanie różnych algorytmów formatowania;
3. wzorzec Dekorator (s. 152) do ozdabiania interfejsu użytkownika;
4. wzorzec Fabryka abstrakcyjna (s. 101), aby dodać obsługę wielu standardów wyglądu i działania;
5. wzorzec Most (s. 181), aby umożliwić obsługę wielu platform okienkowych;
6. wzorzec Polecanie (s. 302) na potrzeby cofania operacji użytkownika;
7. wzorzec Iterator (s. 230) do obsługi dostępu do struktur obiektów i poruszania się po nich;
8. wzorzec Odwiedzający (s. 280), aby umożliwić obsługę dowolnej liczby mechanizmów analitycznych bez komplikowania implementacji struktury dokumentu.

Żaden z wymienionych problemów projektowych nie występuje tylko w edytorsach dokumentów, takich jak aplikacja Lexi. W większości choć trochę skomplikowanych programów można wykorzystać wiele wzorców, choć możliwe, że do osiągnięcia innych niż tu celów. W aplikacji do przeprowadzania analiz finansowych można użyć wzorca Kompozyt do zdefiniowania portfeli inwestycyjnych składających się z różnych portfeli podrzędnych i kont. W kompilatorze można wykorzystać wzorzec Strategia, aby umożliwić stosowanie odmiennych systemów przydziału rejestrów dla różnych maszyn docelowych. W aplikacjach z graficznym interfejsem użytkownika zwykle stosowane są przynajmniej wzorce Dekorator i Polecanie, które i my wykorzystaliśmy.

Choć omówiliśmy kilka istotnych problemów związanych z projektem edytora Lexi, jest też wiele innych trudności, o których nie wspomnialiśmy. Jednak w książce opisaliśmy więcej niż osiem zastosowanych tu wzorców. Dlatego w czasie poznawania pozostałych z nich pomyśl o tym, jak można je wykorzystać w edytorze Lexi, lub — co jeszcze lepsze — jak zastosować je we własnych projektach!

Rozdział 3.

Wzorce konstrukcyjne

Konstrukcyjne wzorce projektowe pozwalają ująć w abstrakcyjnej formie proces tworzenia egzemplarzy klas. Pomagają zachować niezależność systemu od sposobu tworzenia, składania i reprezentowania obiektów. Klasowe wzorce konstrukcyjne są oparte na dziedziczeniu i służą do modyfikowania klas, których egzemplarze są tworzone. W obiektowych wzorcach konstrukcyjnych tworzenie egzemplarzy jest delegowane do innego obiektu.

Wzorce konstrukcyjne zyskują na znaczeniu wraz z coraz częstszym zastępowaniem w systemach dziedziczenia klas składaniem obiektów. Powoduje to, że programiści kładą mniejszy nacisk na trwałe zapisywanie w kodzie określonego zestawu zachowań, a większy — na definiowanie mniejszego zbioru podstawowych działań, które można połączyć w dowolną liczbę bardziej złożonych zachowań. Dlatego tworzenie obiektów o określonych zachowaniach wymaga czegoś więcej niż prostego utworzenia egzemplarza klasy.

We wzorcach z tego rozdziału powtarzają się dwa motywy. Po pierwsze, wszystkie te wzorce kapsulkują informacje o tym, z których klas konkretnych korzysta system. Po drugie, ukrywają proces tworzenia i składania egzemplarzy tych klas. System zna tylko interfejsy obiektów zdefiniowane w klasach abstrakcyjnych. Oznacza to, że wzorce konstrukcyjne dają dużą elastyczność w zakresie tego, co jest tworzone, kto to robi, jak przebiega ten proces i kiedy ma miejsce. Umożliwiają skonfigurowanie systemu z obiektemi-produktami o bardzo zróżnicowanych strukturach i funkcjach. Konfigurowanie może przebiegać statycznie (w czasie komplikacji) lub dynamicznie (w czasie wykonywania programu).

Niektóre wzorce konstrukcyjne są dla siebie konkurencją. Na przykład w niektórych warunkach można z pożytkiem zastosować zarówno wzorzec Prototyp (s. 120), jak i Fabryka abstrakcyjna (s. 101). W innych przypadkach wzorce się uzupełniają. We wzorcu Budowniczy (s. 92) można wykorzystać jeden z pozostałych wzorców do określenia, które komponenty zostaną zbudowane, a do zaimplementowania wzorca Prototyp (s. 120) można użyć wzorca Singleton (s. 130).

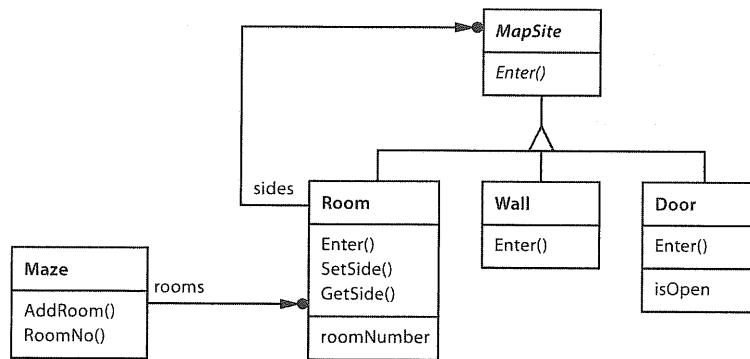
Ponieważ wzorce konstrukcyjne są mocno powiązane ze sobą, przeanalizujemy całą ich piątkę razem, aby podkreślić podobieństwa i różnice między nimi. Wykorzystamy też jeden przykład do zilustrowania implementacji tych wzorców — tworzenie labiryntu na potrzeby gry komputerowej. Labirynt i gra będą nieco odmienne w poszczególnych wzorcach. Czasem celem gry będzie po prostu znalezienie wyjścia z labiryntu. W tej wersji gracz prawdopodobnie będzie

widział tylko lokalny fragment labiryntu. Czasem w labiryntach trzeba będzie rozwiązać problemy i poradzić sobie z zagrożeniami. W tych odmianach można udostępnić mapę zbadanego już fragmentu labiryntu.

Pominiemy wiele szczegółów dotyczących tego, co może znajdować się w labiryncie i czy gra jest jedno-, czy wieloosobowa. Zamiast tego skoncentrujemy się na tworzeniu labiryntów. Labirynt definiujemy jako zbiór pomieszczeń. Każde z nich ma informacje o sąsiadach. Mogą to być następne pokoje, ściana lub drzwi do innego pomieszczenia.

Klasy `Room`, `Door` i `Wall` reprezentują komponenty labiryntu używane we wszystkich przykładach. Definiujemy tylko fragmenty tych klas potrzebne do utworzenia labiryntu. Ignorujemy graczy, operacje wyświetlania labiryntu i poruszania się po nim oraz inne ważne funkcje nieistotne przy generowaniu labiryntów.

Poniższy diagram ilustruje relacje między wspomnianymi klasami:



Każde pomieszczenie ma cztery strony. W implementacji w języku C++ do określania stron północnej, południowej, wschodniej i zachodniej służy typ wyliczeniowy `Direction`:

```
enum Direction {North, South, East, West};
```

W implementacji w języku Smalltalk kierunki te są reprezentowane za pomocą odpowiednich symboli.

`MapSite` to klasa abstrakcyjna wspólna dla wszystkich komponentów labiryntu. Aby uprościć przykład, zdefiniowaliśmy w niej tylko jedną operację — `Enter`. Jej działanie zależy od tego, gdzie gracz wchodzi. Jeśli jest to pomieszczenie, zmienia się lokalizacja gracza. Jeżeli są to drzwi, mogą zajść dwa zdarzenia — jeśli są otwarte, gracz przejdzie do następnego pokoju, a o zamknięte drzwi użytkownik rozbije sobie nos.

```
class MapSite {
public:
    virtual void Enter() = 0;
};
```

`Enter` to prosty podstawowy element bardziej złożonych operacji gry. Na przykład jeśli gracz znajduje się w pomieszczeniu i zechce pójść na wschód, gra może ustalić, który obiekt `MapSite` znajduje się w tym kierunku, i wywołać operację `Enter` tego obiektu. Operacja `Enter` specyficzna

dla podklasy określi, czy gracz zmienił lokalizację czy rozbił sobie nos. W prawdziwej grze operacja Enter mogłaby przyjmować jako argument obiekt reprezentujący poruszającego się gracza.

Room to podkلاsa konkretnej klasy MapSite określająca kluczowe relacje między komponentami labiryntu. Przechowuje referencje do innych obiektów MapSite i numer pomieszczenia (numery te służą do identyfikowania pokojów w labiryncie).

```
class Room : public MapSite {
public:
    Room(int roomNo);

    MapSite* GetSide(Direction) const;
    void SetSide(Direction, MapSite*);

    virtual void Enter();

private:
    MapSite* _sides[4];
    int _roomNumber;
};
```

Poniższe klasy reprezentują ścianę i drzwi umieszczone po dowolnej stronie pomieszczenia.

```
class Wall : public MapSite {
public:
    Wall();

    virtual void Enter();
};

class Door : public MapSite {
public:
    Door(Room* = 0, Room* = 0);

    virtual void Enter();
    Room* OtherSideFrom(Room*);

private:
    Room* _room1;
    Room* _room2;
    bool _isOpen;
};
```

Potrzebne są informacje nie tylko o częściach labiryntu. Zdefiniujemy też klasę Maze reprezentującą kolekcję pomieszczeń. Klasa ta udostępnia operację RoomNo, która znajduje określony pokój po otrzymaniu jego numeru.

```
class Maze {
public:
    Maze();

    void AddRoom(Room*);
```

```

    Room* RoomNo(int) const;
private:
    // ...
};

```

Operacja RoomNo może znajdować pomieszczenia za pomocą wyszukiwania liniowego, tablicy haszującej lub prostej tablicy. Nie będziemy jednak zajmować się takimi szczegółami. Zamiast tego skoncentrujmy się na tym, jak określić komponenty obiektu Maze.

Następną klasą, jaką zdefiniujemy, jest MazeGame. Służy ona do tworzenia labiryntu. Prostym sposobem na wykonanie tego zadania jest użycie serii operacji dodających komponenty do labiryntu i łączących je. Na przykład poniższa funkcja składowa utworzy labirynt składający się z dwóch pomieszczeń rozdzielonych drzwiami:

```

Maze* MazeGame::CreateMaze () {
    Maze* aMaze = new Maze;
    Room* r1 = new Room(1);
    Room* r2 = new Room(2);
    Door* theDoor = new Door(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, new Wall);
    r1->SetSide(East, theDoor);
    r1->SetSide(South, new Wall);
    r1->SetSide(West, new Wall);

    r2->SetSide(North, new Wall);
    r2->SetSide(East, new Wall);
    r2->SetSide(South, new Wall);
    r2->SetSide(West, theDoor);

    return aMaze;
}

```

Funkcja ta jest stosunkowo skomplikowana, jeśli weźmiemy pod uwagę, że jedyne, co robi, to tworzy labirynt składający się z dwóch pomieszczeń. Można łatwo wymyślić sposób na uproszczenie tej funkcji. Na przykład konstruktor klasy Room mógłby inicjować pokój przez przypisanie ścian do jego stron. Jednak to rozwiązań powoduje jedynie przeniesienie kodu w inne miejsce. Prawdziwy problem związany z tą funkcją składową nie jest związany z jej rozmiarem, ale z *brakiem elastyczności*. Powoduje ona zapisanie na stałe układu labiryntu. Zmiana tego układu wymaga zmodyfikowania omawianej funkcji składowej. Można to zrobić albo przez jej przesłonięcie (co oznacza ponowną implementację całego kodu), albo przez zmodyfikowanie jej fragmentów (to podejście jest narażone na błędy i nie sprzyja ponownemu wykorzystaniu rozwiązania).

Wzorce konstrukcyjne pokazują, jak zwiększyć *elastyczność* projektu. Nie zawsze oznacza to zmniejszenie samego projektu. Wzorce te przede wszystkim ułatwiają modyfikowanie klas definiujących komponenty labiryntu.

Założmy, że chcemy powtórnie wykorzystać układ labiryntu w nowej grze obejmującej (między innymi) magiczne labirynty. Potrzebne będą w niej nowe rodzaje komponentów, takie jak *DoorNeedingSpell* (drzwi, które można zamknąć i następnie otworzyć tylko za pomocą czaru) i *EnchantedRoom* (pokój z niezwykłymi przedmiotami, na przykład magicznymi kluczami lub czarami). Jak można w łatwy sposób zmodyfikować operację *CreateMaze*, aby tworzyła labirynty z obiektami nowych klas?

W tym przypadku największa przeszkoda związana jest z zapisaniem na stałe klas, których egzemplarze tworzy opisywana operacja. Wzorce konstrukcyjne udostępniają różne sposoby usuwania bezpośrednich referencji do klas konkretnych z kodu, w którym trzeba tworzyć egzemplarze takich klas:

- ▶ Jeśli operacja *CreateMaze* przy tworzeniu potrzebnych pomieszczeń, ścian i drzwi wywołuje funkcje wirtualne zamiast konstruktora, można zmienić klasy, których egzemplarze powstają, przez utworzenie podklasy klasy *MazeGame* i ponowne zdefiniowanie funkcji wirtualnych. To rozwiązanie to przykład zastosowania wzorca Metoda twytwórcza (s. 110).
- ▶ Jeśli operacja *CreateMaze* otrzymuje jako parametr obiekt, którego używa do tworzenia pomieszczeń, ścian i drzwi, można zmienić klasy tych komponentów przez przekazanie nowych parametrów. Jest to przykład zastosowania wzorca Fabryka abstrakcyjna (s. 101).
- ▶ Jeśli operacja *CreateMaze* otrzymuje obiekt, który potrafi utworzyć cały nowy labirynt za pomocą operacji dodawania pomieszczeń, drzwi i ścian, można zastosować dziedziczenie do zmodyfikowania fragmentów labiryntu lub sposobu jego powstawania. W ten sposób działa wzorzec Budowniczy (s. 92).
- ▶ Jeśli operacja *CreateMaze* jest sparametryzowana za pomocą różnych prototypowych obiektów reprezentujących pomieszczenia, drzwi i ściany, które kopiuje i dodaje do labiryntu, można zmienić układ labiryntu przez zastąpienie danych obiektów prototypowych innymi. Jest to przykład zastosowania wzorca Prototyp (s. 120).

Ostatni wzorzec konstrukcyjny, *Singleton* (s. 130), pozwala zagwarantować, że w grze powstanie tylko jeden labirynt, a wszystkie obiekty gry będą mogły z niego korzystać (bez uciekania się do stosowania zmiennych lub funkcji globalnych). Wzorzec ten ułatwia też rozbudowywanie lub zastępowanie labiryntów bez modyfikowania istniejącego kodu.

BUDOWNICZY (BUILDER)

obiektowy, konstrukcyjny

PRZEZNACZENIE

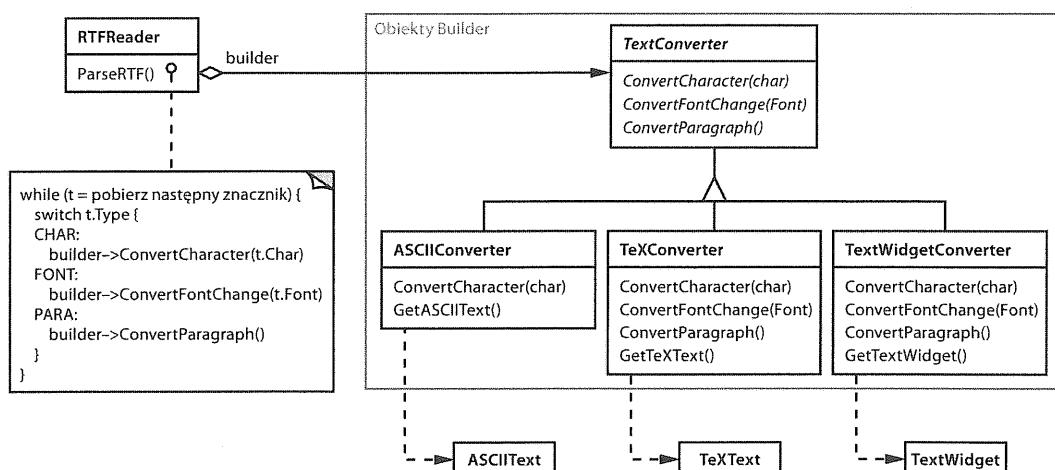
Oddziela tworzenie złożonego obiektu od jego reprezentacji, dzięki czemu ten sam proces konstrukcji może prowadzić do powstawania różnych reprezentacji.

UZASADNIENIE

Czytnik dokumentów w formacie RTF (ang. *Rich Text Format*) powinien móc przekształcać takie dokumenty na wiele formatów tekstowych. Takie narzędzie mogłoby przeprowadzać konwersję dokumentów RTF na zwykły tekst w formacie ASCII lub na widget tekstowy, który można interaktywnie edytować. Jednak problem polega na tym, że liczba możliwych przekształceń jest nieokreślona. Dlatego należy zachować możliwość łatwego dodawania nowych metod konwersji bez konieczności modyfikowania czytnika.

Rozwiążanie polega na skonfigurowaniu klasy RTFReader za pomocą obiektu TextConverter przekształcającego dokumenty RTF na inną reprezentację tekstową. Klasa RTFReader w czasie analizowania dokumentu RTF korzysta z obiektu TextConverter do przeprowadzania konwersji. Kiedy klasa RTFReader wykryje znacznik formatu RTF (w postaci zwykłego tekstu lub słowa sterującego z tego formatu), przekaże do obiektu TextConverter żądanie przekształcenia znacznika. Obiekty TextConverter odpowiadają zarówno za przeprowadzanie konwersji danych, jak i zapisywanie znacznika w określonym formacie.

Podklasy klasy TextConverter są wyspecjalizowane pod kątem różnych konwersji i formatów. Na przykład klasa ASCIIConverter ignoruje żądania związane z konwersją elementów innych niż zwykły tekst. Z kolei klasa TeXConverter obejmuje implementację operacji obsługujących wszystkie żądania, co umożliwia utworzenie reprezentacji w formacie T.X, uwzględniającej wszystkie informacje na temat stylu tekstu. Klasa TextWidgetConverter generuje złożony obiekt interfejsu użytkownika umożliwiający oglądanie i edytowanie tekstu.



Każda klasa konwertująca przyjmuje mechanizm tworzenia i składania obiektów złożonych oraz ukrywa go za abstrakcyjnym interfejsem. Konwerter jest oddzielony od czytnika odpowiadającego za analizowanie dokumentów RTF.

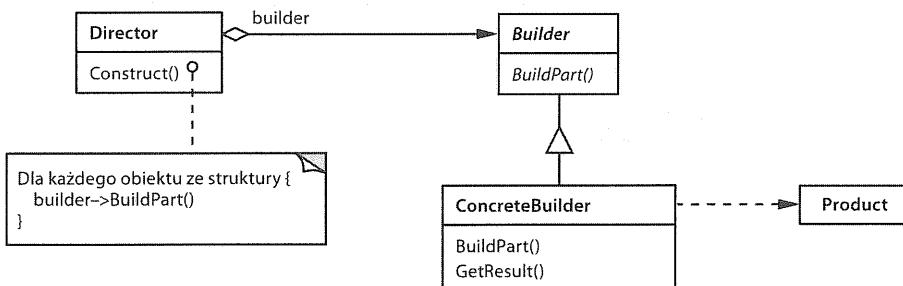
Wzorzec Budowniczy ujmuje wszystkie te relacje. W tym wzorcu każda klasa konwertująca nosi nazwę **builder** (czyli budowniczy), a klasa czytnika to **director** (czyli kierownik). Zastosowanie wzorca Budowniczy w przytoczonym przykładzie powoduje oddzielenie algorytmu interpretującego format tekstowy (czyli parsera dokumentów RTF) od procesu tworzenia i reprezentowania przekształconego dokumentu. Umożliwia to powtórne wykorzystanie algorytmu analizującego z klasy RTFReader do przygotowania innych reprezentacji tekstu z dokumentów RTF. Aby to osiągnąć, wystarczy skonfigurować klasę RTFReader za pomocą innej podklasy klasy TextConverter.

WARUNKI STOSOWANIA

Wzorca Budowniczy należy używać w następujących sytuacjach:

- ▶ Jeśli algorytm tworzenia obiektu złożonego powinien być niezależny od składników tego obiektu i sposobu ich łączenia.
- ▶ Kiedy proces konstrukcji musi umożliwiać tworzenie różnych reprezentacji generowanego obiektu.

STRUKTURA



ELEMENTY

- ▶ **Builder** (TextConverter), czyli budowniczy:
 - określa interfejs abstrakcyjny do tworzenia składników obiektu Product.
- ▶ **ConcreteBuilder** (ASCIIConverter, TeXConverter, TextWidgetConverter), czyli budowniczy konkretny:
 - tworzy i łączy składniki produktu w implementacji interfejsu klasy Builder;
 - definiuje i śledzi generowane reprezentacje;
 - udostępnia interfejs do pobierania produktów (na przykład operacje GetASCIIText i GetTextWidget).

► **Director** (RTFReader), czyli kierownik:

- tworzy obiekt za pomocą interfejsu klasy Builder.

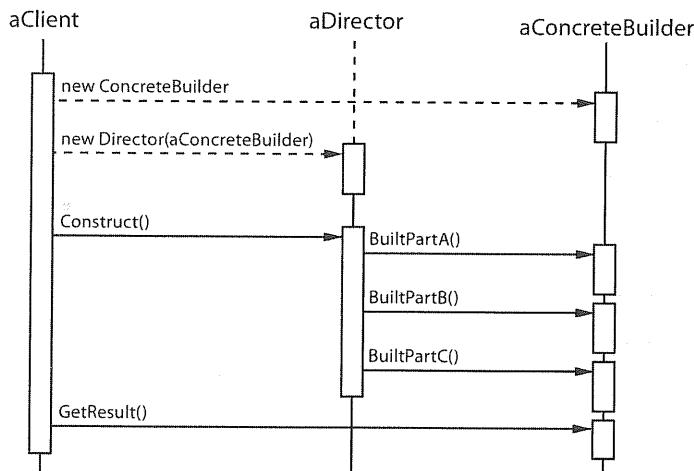
► **Product** (ASCIIText, TeXText, TextWidget):

- reprezentuje generowany obiekt złożony; klasa `ConcreteBuilder` tworzy wewnętrzną reprezentację produktu i definiuje proces jej składania;
- obejmuje klasy definiujące składowe elementy obiektu, w tym interfejsy do łączenia składowych w ostateczną postać obiektu.

WSPÓŁDZIAŁANIE

- Klient tworzy obiekt Director i konfiguruje go za pomocą odpowiedniego obiektu Builder.
- Kiedy potrzebne jest utworzenie części produktu, obiekt Director wysyła powiadomienie do obiektu Builder.
- Obiekt Builder obsługuje żądania od obiektu Director i dodaje części do produktu.
- Klient pobiera produkt od obiektu Builder.

Poniższy diagram interakcji pokazuje, w jaki sposób klasy Builder i Director współpracują z klientem.



KONSEKWENCJE

Oto kluczowe konsekwencje zastosowania wzorca Budowniczy:

1. *Możliwość modyfikowania wewnętrznej reprezentacji produktu.* Obiekt Builder udostępnia obiektowi Director interfejs abstrakcyjny do tworzenia produktu. Interfejs ten umożliwia obiektowi Builder ukrycie reprezentacji i wewnętrznej struktury produktu, a także sposobu jego składania. Ponieważ do tworzenia produktu służy interfejs abstrakcyjny, zmiana wewnętrznej reprezentacji produktu wymaga jedynie zdefiniowania obiektu Builder nowego rodzaju.

2. *Odzizłowanie reprezentacji od kodu służącego do tworzenia produktu.* Wzorzec Budowniczy pomaga zwiększyć modularność, ponieważ kapsułkuje sposób tworzenia i reprezentowania obiektu złożonego. Klienci nie potrzebują żadnych informacji o klasach definiujących wewnętrzną strukturę produktu, ponieważ klasy te nie występują w interfejsie obiektu Builder.

Każdy obiekt `ConcreteBuilder` obejmuje cały kod potrzebny do tworzenia i składania produktów określonego rodzaju. Kod ten wystarczy napisać raz. Następnie można wielokrotnie wykorzystać go w różnych obiektach `Director` do utworzenia wielu odmian obiektu `Product` za pomocą tych samych składników. W przykładzie dotyczącym dokumentów RTF moglibyśmy zdefiniować czytnik dokumentów o formacie innym niż RTF, na przykład klasę `SGMLReader`, i użyć tych samych podklas klasy `TextConverter` do wygenerowania reprezentacji dokumentów SGML w postaci obiektów `ASCIIText`, `TeXText` i `TextWidget`.

3. *Większa kontrola nad procesem tworzenia.* Wzorzec Budowniczy — w odróżnieniu od wzorców konstrukcyjnych tworzących produkty w jednym etapie — polega na generowaniu ich krok po kroku pod kontrolą obiektu `Director`. Dopiero po ukończeniu produktu obiekt `Director` odbiera go od obiektu `Builder`. Dlatego interfejs klasy `Builder` w większym stopniu niż inne wzorce konstrukcyjne odzwierciedla proces tworzenia produktów. Zapewnia to pełniejszą kontrolę nad tym procesem, a tym samym i wewnętrzną strukturą gotowego produktu.

IMPLEMENTACJA

Zwykle w implementacji znajduje się klasa abstrakcyjna `Builder` obejmująca definicję operacji dla każdego komponentu, którego utworzenia może zażądać obiekt `Director`. Domyslnie operacje te nie wykonują żadnych działań. W klasie `ConcreteBuilder` przesłonięte są operacje komponentów, które klasa ta ma generować.

Oto inne związane z implementacją kwestie, które należy rozważyć:

1. *Interfejs do składania i tworzenia obiektów.* Obiekty `Builder` tworzą produkty krok po kroku. Dlatego interfejs klasy `Builder` musi być wystarczająco ogólny, aby umożliwiał konstruowanie produktów każdego rodzaju przez konkretne podklasy klasy `Builder`.

Kluczowa kwestia projektowa dotyczy modelu procesu tworzenia i składania obiektów. Zwykle wystarczający jest model, w którym efekty zgłoszenia żądania konstrukcji są po prostu dodawane do produktu. W przykładzie związanym z dokumentami RTF obiekt `Builder` przekształca i dodaje następny znacznik do wcześniejszej skonwertowanego tekstu.

Jednak czasem potrzebny jest dostęp do wcześniejszej utworzonych części produktu. W przykładzie dotyczącym labiryntów, który prezentujemy w punkcie Przykładowy kod, interfejs klasy `MazeBuilder` umożliwia dodanie drzwi między istniejącymi pomieszczeniami. Następnym przykładem, w którym jest to potrzebne, są budowane od dołu do góry struktury drzewiaste, takie jak drzewa składni. Wtedy obiekt `Builder` zwraca węzły podzielone obiektowi `Director`, który następnie przekazuje je ponownie do obiektu `Builder`, aby ten utworzył węzły nadzędne.

2. *Dlaczego nie istnieje klasa abstrakcyjna produktów?* W typowych warunkach produkty tworzone przez obiekty `ConcreteBuilder` mają tak odmienną reprezentację, że udostępnienie wspólnej klasy nadrzędnej dla różnych produktów przynosi niewielkie korzyści. W przykładzie dotyczącym dokumentów RTF obiekty `ASCIIText` i `TextWidget` prawdopodobnie nie będą miały wspólnego interfejsu ani też go nie potrzebują. Ponieważ klienci zwykle konfigurują obiekt `Director` za pomocą odpowiedniego obiektu `ConcreteBuilder`, klient potrafi określić, która podklasa konkretnej klasy `Builder` jest używana, i na tej podstawie obsługuje dostępne produkty.
3. *Zastosowanie pustych metod domyślnych w klasie Builder.* W języku C++ metody służące do tworzenia obiektów celowo nie są deklarowane jako czysto wirtualne funkcje składowe. W zamian definiuje się je jako puste metody, dzięki czemu w klientach trzeba przesłonić tylko potrzebne operacje.

PRZYKŁADOWY KOD

Zdefiniujmy nową wersję funkcji składowej `CreateMaze` (s. 90). Będzie ona przyjmować jako argument obiekt budujący klasy `MazeBuilder`.

Klasa `MazeBuilder` definiuje poniższy interfejs służący do tworzenia labiryntów:

```
class MazeBuilder {
public:
    virtual void BuildMaze() { }
    virtual void BuildRoom(int room) { }
    virtual void BuildDoor(int roomFrom, int roomTo) { }
    virtual Maze* GetMaze() { return 0; }
protected:
    MazeBuilder();
};
```

Ten interfejs pozwala utworzyć trzy elementy: (1) labirynt, (2) pomieszczenia o określonym numerze i (3) drzwi między ponumerowanymi pokojami. Operacja `GetMaze` zwraca labirynt klientowi. W podkласach klasy `MazeBuilder` należy ją przesłonić, aby zwracały one generowany przez siebie labirynt.

Wszystkie związane z budowaniem labiryntu operacje klasy `MazeBuilder` domyślnie nie wykonują żadnych działań. Jednak nie są zadeklarowane jako czysto wirtualne, dzięki czemu w klasach pochodnych wystarczy przesłonić tylko potrzebne metody.

Po utworzeniu interfejsu klasy `MazeBuilder` można zmodyfikować funkcję składową `CreateMaze`, aby przyjmowała jako parametr obiekt tej klasy:

```
Maze* MazeGame::CreateMaze (MazeBuilder& builder) {
    builder.BuildMaze();

    builder.BuildRoom(1);
    builder.BuildRoom(2);
    builder.BuildDoor(1, 2);

    return builder.GetMaze();
}
```

Porównajmy tę wersję operacji `CreateMaze` z jej pierwotną wersją. Warto zauważyć, w jaki sposób w budowniczym ukryto wewnętrzną reprezentację labiryntu — czyli klasy z definicjami pomieszczeń, drzwi i ścian — i jak elementy te są składane w gotowy labirynt. Można się domyślić, że istnieją klasy reprezentujące pomieszczenia i drzwi, jednak w kodzie nie ma wskazówek dotyczących klasy związanej ze ścianami. Ułatwia to zmianę reprezentacji labiryntu, ponieważ nie trzeba modyfikować kodu żadnego z klientów używających klasy `MazeBuilder`.

Wzorzec Budowniczy — podobnie jak inne wzorce konstrukcyjne — kapsułkuje tworzenie obiektów. Tutaj służy do tego interfejs zdefiniowany w klasie `MazeBuilder`. Oznacza to, że możemy wielokrotnie wykorzystać tę klasę do tworzenia labiryntów różnego rodzaju. Przykładem na to jest operacja `CreateComplexMaze`:

```
Maze* MazeGame::CreateComplexMaze (MazeBuilder& builder) {
    builder.BuildRoom(1);
    // ...
    builder.BuildRoom(1001);

    return builder.GetMaze();
}
```

Warto zauważyć, że klasa `MazeBuilder` nie tworzy labiryntu. Służy ona głównie do definiowania interfejsu do generowania labiryntów. Puste implementacje znajdują się w niej dla wygody programisty, natomiast potrzebne działania wykonują podklasy klasy `MazeBuilder`.

Podklasa `StandardMazeBuilder` to implementacja służąca do tworzenia prostych labiryntów. Zapisuje ona budowany labirynt w zmiennej `_currentMaze`.

```
class StandardMazeBuilder : public MazeBuilder {
public:
    StandardMazeBuilder();

    virtual void BuildMaze();
    virtual void BuildRoom(int);
    virtual void BuildDoor(int, int);

    virtual Maze* GetMaze();
private:
    Direction CommonWall(Room*, Room*);
    Maze* _currentMaze;
};
```

`CommonWall` to operacja narzędziowa określająca kierunek standardowej ściany pomiędzy dwoma pomieszczeniami.

Konstruktor `StandardMazeBuilder` po prostu inicjuje zmienną `_currentMaze`.

```
StandardMazeBuilder::StandardMazeBuilder () {
    _currentMaze = 0;
}
```

Operacja `BuildMaze` tworzy egzemplarz klasy `Maze`, który pozostałe operacje składają i ostatecznie zwracają do klienta (za to odpowiada operacja `GetMaze`).

```

void StandardMazeBuilder::BuildMaze () {
    _currentMaze = new Maze;
}

Maze* StandardMazeBuilder::GetMaze () {
    return _currentMaze;
}

```

Operacja BuildRoom tworzy pomieszczenie i ściany wokół niego.

```

void StandardMazeBuilder::BuildRoom (int n) {
    if (!_currentMaze->RoomNo(n)) {
        Room* room = new Room(n);
        _currentMaze->AddRoom(room);

        room->SetSide(North, new Wall);
        room->SetSide(South, new Wall);
        room->SetSide(East, new Wall);
        room->SetSide(West, new Wall);
    }
}

```

Aby utworzyć drzwi między dwoma pomieszczeniami, obiekt StandardMazeBuilder wyszukuje w labiryncie odpowiednie pokoje i łączącą je ścianę.

```

void StandardMazeBuilder::BuildDoor (int n1, int n2) {
    Room* r1 = _currentMaze->RoomNo(n1);
    Room* r2 = _currentMaze->RoomNo(n2);
    Door* d = new Door(r1, r2);

    r1->SetSide(CommonWall(r1,r2), d);
    r2->SetSide(CommonWall(r2,r1), d);
}

```

Klienci mogą teraz użyć do utworzenia labiryntu operacji CreateMaze wraz z obiektem StandardMazeBuilder.

```

Maze* maze;
MazeGame game;
StandardMazeBuilder builder;

game.CreateMaze(builder);
maze = builder.GetMaze();

```

Moglibyśmy umieścić wszystkie operacje klasy StandardMazeBuilder w klasie Maze i pozwolić każdemu obiektowi Maze, aby samodzielnie utworzył swój egzemplarz. Jednak zmniejszenie klasy Maze sprawia, że łatwiej będzie ją zrozumieć i zmodyfikować, a wyodrębnienie z niej klasy StandardMazeBuilder nie jest trudne. Co jednak najważniejsze, rozdzielenie tych klas pozwala utworzyć różnorodne obiekty z rodziny MazeBuilder, z których każdy używa innych klas do generowania pomieszczeń, ścian i drzwi.

CountingMazeBuilder to bardziej wymyślna podklasa klasy MazeBuilder. Budowniczowie tego typu w ogóle nie tworzą labiryntów, a jedynie zliczają utworzone komponenty różnych rodzajów.

```
class CountingMazeBuilder : public MazeBuilder {
public:
    CountingMazeBuilder();

    virtual void BuildMaze();
    virtual void BuildRoom(int);
    virtual void BuildDoor(int, int);
    virtual void AddWall(int, Direction);

    void GetCounts(int&, int&) const;
private:
    int _doors;
    int _rooms;
};
```

Konstruktor inicjuje liczniki, a przesłonięte operacje klasy MazeBuilder w odpowiedni sposób powiększają ich wartość.

```
CountingMazeBuilder::CountingMazeBuilder () {
    _rooms = _doors = 0;
}

void CountingMazeBuilder::BuildRoom (int) {
    _rooms++;
}

void CountingMazeBuilder::BuildDoor (int, int) {
    _doors++;
}

void CountingMazeBuilder::GetCounts (
    int& rooms, int& doors
) const {
    rooms = _rooms;
    doors = _doors;
}
```

Klient może korzystać z klasy CountingMazeBuilder w następujący sposób:

```
int rooms, doors;
MazeGame game;
CountingMazeBuilder builder;

game.CreateMaze(builder);
builder.GetCounts(rooms, doors);

cout << "Liczba pomieszczeń w labiryncie to "
    << rooms << ", a liczba drzwi wynosi "
    << doors << "." << endl;
```

ZNANE ZASTOSOWANIA

Aplikacja do konwersji dokumentów RTF pochodzi z platformy ET++ [WGM88]. Jej część służąca do obsługi tekstu wykorzystuje budowniczego do przetwarzania tekstu zapisanego w formacie RTF.

Wzorzec Budowniczy jest często stosowany w języku Smalltalk-80 [Par90]:

- ▶ Klasa `Parser` w podsystemie odpowiedzialnym za komplikację pełni funkcję kierownika i przyjmuje jako argument obiekt `ProgramNodeBuilder`. Obiekt `Parser` za każdym razem, kiedy rozpozna daną konstrukcję składniową, wysyła do powiązanego z nim obiektu `ProgramNodeBuilder` powiadomienie. Kiedy parser kończy działanie, żąda od budowniczego utworzenia drzewa składni i przekazuje je klientowi.
- ▶ `ClassBuilder` to budowniczy, którego klasy używają do tworzenia swoich podklas. W tym przypadku klasa jest zarówno kierownikiem, jak i produktem.
- ▶ `ByteCodeStream` to budowniczy, który tworzy skompilowaną metodę w postaci tablicy bajtów. Klasa `ByteCodeStream` to przykład niestandardowego zastosowania wzorca Budowniczy, ponieważ generowany przez nią obiekt złożony jest kodowany jako tablica bajtów, a nie jako zwykły obiekt języka Smalltalk. Jednak interfejs klasy `ByteCodeStream` jest typowy dla budowniczych i łatwo można zastąpić tę klasą inną, reprezentującą programy jako obiekty składowe.

Platforma Service Configurator wchodząca w skład środowiska Adaptive Communications Environment korzysta z budowniczych do tworzenia komponentów usług sieciowych dołączanych do serwera w czasie jego działania [SS94]. Komponenty te są opisane w języku konfiguracyjnym analizowanym przez parser LALR(1). Akcje semantyczne parsera powodują wykonanie operacji na budowniczym, który dodaje informacje do komponentu usługowego. W tym przykładzie parser pełni funkcję kierownika.

POWIĄZANE WZORCE

Fabryka abstrakcyjna (s. 101) przypomina wzorzec Budowniczy, ponieważ też może służyć do tworzenia obiektów złożonych. Główna różnica między nimi polega na tym, że wzorzec Budowniczy opisuje przede wszystkim tworzenie obiektów złożonych krok po kroku. We wzorcu Fabryka abstrakcyjna nacisk położony jest na rodzinę obiektów-produktów (zarówno prostych, jak i złożonych). Budowniczy zwraca produkt w ostatnim kroku, natomiast we wzorcu Fabryka abstrakcyjna produkt jest udostępniany natychmiast.

Budowniczy często służy do tworzenia kompozytów (s. 170).

FABRYKA ABSTRAKCYJNA (ABSTRACT FACTORY) obiektowy, konstrukcyjny

PRZEZNACZENIE

Udostępnia interfejs do tworzenia rodzin powiązanych ze sobą lub zależnych od siebie obiektów bez określania ich klas konkretnych.

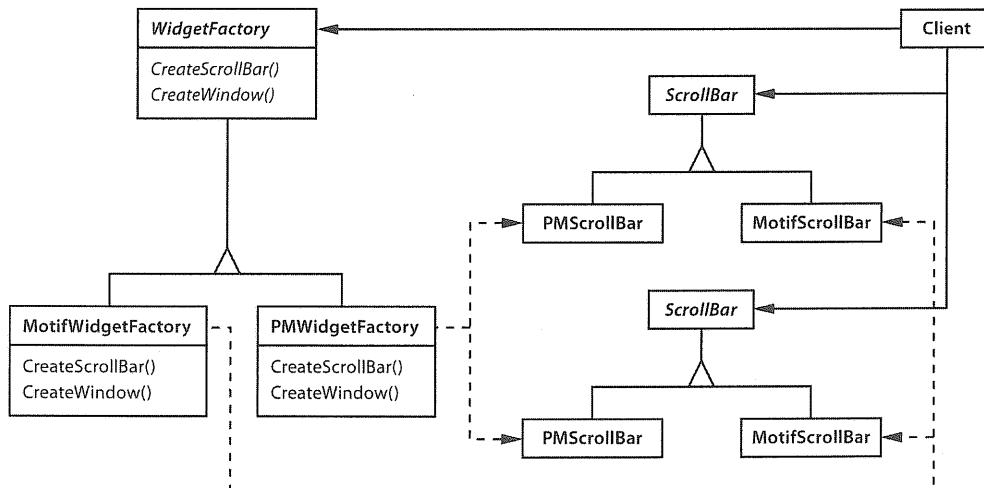
INNE NAZWY

Zestaw (ang. *kit*).

UZASADNIENIE

Zastanówmy się nad pakietem narzędziowym do tworzenia interfejsów użytkownika, obsługującym różne standardy wyglądu i działania (na przykład Motif i Presentation Manager). Poszczególne standardy wyznaczają różnygląd i inne zachowanie widgetów interfejsu użytkownika, takich jak paski przewijania, okna i przyciski. Aby aplikacja była przenośna między różnymi standardami, nie należy zapisywać w niej na stałe określonego wyglądu i sposobu działania widgetów. Tworzenie określających te aspekty egzemplarzy klas w różnych miejscach aplikacji utrudnia późniejszą zmianę jej wyglądu i zachowania.

Możemy rozwiązać ten problem przez zdefiniowanie klasy abstrakcyjnej *WidgetFactory* i zadeklarowanie w niej interfejsu do tworzenia podstawowych widgetów. Należy przygotować też klasy abstrakcyjne dla poszczególnych rodzajów widgetów oraz podklasy konkretne z implementacją określonych standardów wyglądu i działania. Interfejs klasy *WidgetFactory* obejmuje operacje, które zwracają nowe obiekty dla klas abstrakcyjnych reprezentujących poszczególne widgety. Klienci wywołują te operacje, aby otrzymać egzemplarze widgetów, ale nie wiedzą, której klasy konkretnej używają. Dlatego klienci pozostają niezależne od stosowanego standardu wyglądu i działania.



Dla każdego standardu wyglądu i działania istnieje podklasa konkretna klasy `WidgetFactory`. W każdej takiej podklasie zaimplementowane są operacje do tworzenia widgetów odpowiednich dla danego standardu. Na przykład operacja `CreateScrollBar` klasy `MotifWidgetFactory` tworzy i zwraca egzemplarz paska przewijania zgodnego ze standardem Motif, natomiast odpowiadająca jej operacja klasy `PMWidgetFactory` tworzy pasek przewijania dla standardu Presentation Manager. Klienci tworzą widgety wyłącznie za pośrednictwem interfejsu klasy `WidgetFactory` i nie znają klas z implementacjami widgetów dla określonych standardów wyglądu i działania. Oznacza to, że klienci muszą być zgodne tylko z interfejsem klasy abstrakcyjnej, a nie z konkretnymi klasami konkretnymi.

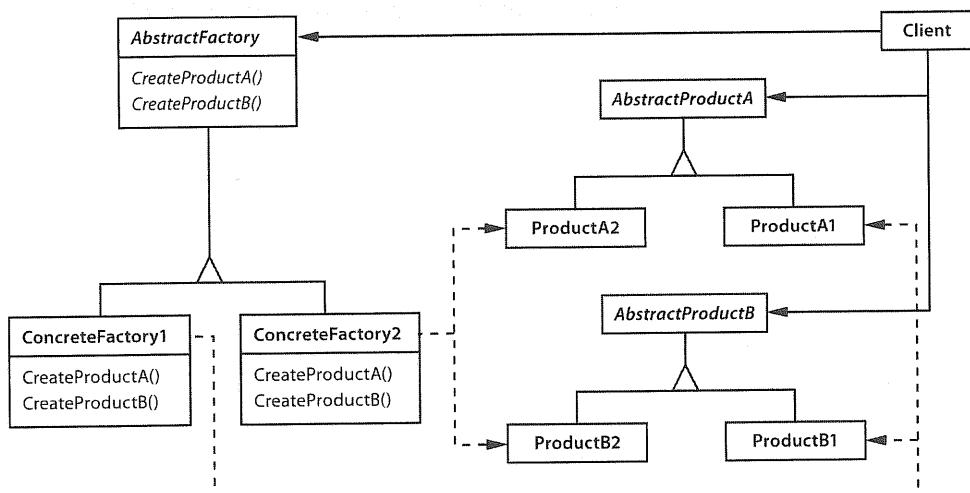
Klasa `WidgetFactory` wymusza ponadto zależności między klasami konkretnymi widgetów. Paska przewijania standardu Motif należy używać wraz z przyciskiem i edytorem tekstu zgodnymi z tym standardem. Ograniczenie to jest wymuszane automatycznie (jest to skutek zastosowania klasy `MotifWidgetFactory`).

WARUNKI STOSOWANIA

Wzorzec Fabryka abstrakcyjna należy stosować w następujących warunkach:

- ▶ Kiedy system powinien być niezależny od sposobu tworzenia, składania i reprezentowania jego produktów.
- ▶ Jeśli system należy skonfigurować za pomocą jednej z wielu rodzin produktów.
- ▶ Jeżeli powiązane obiekty-produkty z jednej rodziny są zaprojektowane do wspólnego użytku i trzeba wymusić jednoczesne korzystanie z tych obiektów.
- ▶ Kiedy programista chce udostępnić klasę biblioteczną produktów i ujawnić jedynie ich interfejsy, a nie implementacje.

STRUKTURA



ELEMENTY

- ▶ **AbstractFactory (WidgetFactory)**, czyli fabryka abstrakcyjna:
 - obejmuje deklarację interfejsu z operacjami tworzącymi produkty abstrakcyjne.
- ▶ **ConcreteFactory (MotifWidgetFactory, PMWidgetFactory)**, czyli fabryka konkretna:
 - obejmuje implementację operacji tworzących produkty konkretne.
- ▶ **AbstractProduct (Window, ScrollBar)**, czyli produkt abstrakcyjny:
 - obejmuje deklarację interfejs dla produktów określonego typu.
- ▶ **ConcreteProduct (MotifWindow, MotifScrollBar)**, czyli produkt konkretny:
 - definiuje obiekt-produkt tworzony przez odpowiadającą mu fabrykę konkretną;
 - obejmuje implementację interfejsu klasy **AbstractProduct**.
- ▶ **Client**:
 - korzysta jedynie z interfejsów zadeklarowanych w klasach **AbstractFactory** i **AbstractProduct**.

WSPÓŁDZIAŁANIE

- ▶ W czasie wykonywania programu powstaje zwykle jeden egzemplarz klasy **ConcreteFactory**. Ta fabryka konkretna tworzy obiekty-produkty o określonej implementacji. Aby wygenerować różne obiekty-produkty, klienci muszą użyć odmiennych fabryk konkretnych.
- ▶ Klasa **AbstractFactory** przekazuje tworzenie obiektów-produktów do swojej podklasy **ConcreteFactory**.

KONSEKWENCJE

Wzorzec Fabryka abstrakcyjna ma następujące zalety i wady:

4. *Izoluje klasy konkretne.* Wzorzec Fabryka abstrakcyjna pomaga kontrolować klasy obiektów tworzonych przez aplikację. Ponieważ fabryka kapsuluje zadanie i proces tworzenia obiektów-produktów, izoluje klienty od klas zawierających implementację. Klienci manipulują egzemplarzami tych klas za pośrednictwem interfejsów abstrakcyjnych. Nazwy klas produktów są odizolowane w implementacji fabryki konkretnej i nie pojawiają się w kodzie klienckim.
5. *Ułatwia zastępowanie rodzin produktów.* Klasa fabryki konkretnej pojawia się w aplikacji tylko raz — w miejscu tworzenia jej egzemplarza. Dlatego łatwo jest zmienić fabrykę konkretną wykorzystywaną przez aplikację. Aby użyć w programie innego zestawu produktów, wystarczy podać inną fabrykę konkretną. Ponieważ fabryka abstrakcyjna tworzy kompletną rodzinę produktów, jednocześnie zmieniana jest cała taka rodzina. W przykładowym interfejsie użytkownika można zastąpić widżety standardu Motif widgetami standardu Presentation Manager w prosty sposób — przez podmianę odpowiednich obiektów-fabryk i odtworzenie interfejsu.

6. Ułatwia zachowanie spójności między produktami. Jeśli obiekty-produkty z danej rodziny są zaprojektowane tak, aby używać ich razem, ważne jest, aby aplikacja w danym momencie korzystała z obiektów z tylko jednej rodziny. Klasa `AbstractFactory` pozwala w łatwy sposób wymusić to ograniczenie.
7. Utrudnia dodawanie obsługi produktów nowego rodzaju. Rozszerzanie fabryk abstrakcyjnych w celu tworzenia produktów nowego typu nie jest proste. Wynika to z tego, że w interfejsie klasy `AbstractFactory` na stałe zapisany jest zestaw produktów, które można utworzyć. Aby dodać obsługę produktów nowego rodzaju, trzeba rozszerzyć interfejs fabryki, co wymaga zmodyfikowania klasy `AbstractFactory` i wszystkich jej podklas. Jedno z rozwiązań tego problemu omawiamy w punkcie Implementacja.

IMPLEMENTACJA

Oto kilka technik przydatnych przy implementowaniu wzorca Fabryka abstrakcyjna.

1. *Fabryki jako singletony.* W aplikacji zwykle potrzebny jest tylko jeden egzemplarz klasy `ConcreteFactory` na każdą rodzinę produktów. Dlatego zazwyczaj najlepiej jest implementować takie klasy zgodnie ze wzorcem Singleton (s. 130).
2. *Tworzenie produktów.* Klasa `AbstractFactory` obejmuje jedynie deklarację *interfejsu* do tworzenia produktów. To podklasy `ConcreteProduct` odpowiadają za ich generowanie. Najczęściej definiowana jest w tym celu metoda wytwarzca (zobacz wzorzec Metoda wytwarzca, s. 110) dla każdego produktu. W fabryce konkretnej generowane produkty są określane przez przesłonięcie metody fabrycznej dla każdego z tych produktów. Choć taka implementacja jest prosta, wymaga przygotowania dla każdej rodziny produktów nowej podklasy konkretnej reprezentującej fabrykę, nawet jeśli różnice między poszczególnymi rodzinami są niewielkie.

Jeśli aplikacja może obejmować wiele rodzin produktów, fabrykę konkretną można zaimplementować za pomocą wzorca Prototyp (s. 120). Fabryka konkretna jest wtedy inicjowana za pomocą prototypowego egzemplarza każdego produktu z rodziny i tworzy nowe produkty przez klonowanie ich prototypów. Podejście oparte na wzorcu Prototyp pozwala wyeliminować konieczność tworzenia dla każdej rodziny produktów nowej klasy konkretnej reprezentującej fabrykę.

Oto sposób na zaimplementowanie fabryki opartej na wzorcu Prototyp w języku Smalltalk. Fabryka konkretna przechowuje klonowane prototypy w słowniku o nazwie `partCatalog`. Metoda `make:` pobiera prototyp i klonuje go:

```
make: partName
  ^ (partCatalog at: partName) copy
```

Fabryka konkretna obejmuje metodę do dodawania elementów do katalogu:

```
addPart: partTemplate named: partName
  partCatalog at: partName put: partTemplate
```

Prototypy są dodawane do fabryki przez wskazanie ich za pomocą symbolu:

```
aFactory addPart: aPrototype named: #ACMEWidget
```

W językach, w których klasy są traktowane jak standardowe obiekty (na przykład w językach Smalltalk i Objective C), można zastosować pewną odmianę podejścia opartego na wzorcu Prototyp. W tych językach klasy można uznać za uproszczone fabryki tworzące produkty tylko jednego rodzaju. W tworzącej produkty fabryce konkretnej można przypisać do zmiennych *klasy* (podobnie jak prototypy). Te klasy będą tworzyć nowe egzemplarze na rzecz fabryki konkretnej. Aby zdefiniować nową fabrykę, należy zainicjować egzemplarz fabryki konkretnej za pomocą *klas* produktów, zamiast tworzyć podklassę. To podejście pozwala wykorzystać specyficzne cechy języków, natomiast podstawowe rozwiązańe oparte na wzorcu Prototyp jest niezależne od języka.

Wersja oparta na klasach — podobnie jak opisane właśnie fabryki oparte na wzorcu Prototyp napisane w języku Smalltalk — ma jedną zmienną egzemplarza (*partCatalog*). Jest to słownik, którego kluczami są nazwy poszczególnych elementów. Zmienna *partCatalog* nie przechowuje przeznaczonych do sklonowania prototypów, ale klasy produktów. Nowa wersja metody *make*: wygląda tak:

```
make: partName
    ^ (partCatalog at: partName) new
```

3. *Definiowanie rozszerzalnych fabryk.* W klasie *AbstractFactory* zwykle zdefiniowane są różne operacje dla wszystkich rodzajów produktów generowanych przez tę klasę. Rodzaje produktów są określone w sygnaturach operacji. Dodanie produktu nowego rodzaju wymaga zmodyfikowania interfejsu klasy *AbstractFactory* i wszystkich klas od niego zależnych.

Elastyczniejszy (choć mniej bezpieczny) projekt wymaga dodania parametru do operacji tworzących obiekty. Ten parametr określa rodzaj generowanego obiektu. Jako parametru można użyć identyfikatora klasy, liczby całkowitej, łańcucha znaków lub dowolnego innego elementu identyfikującego rodzaj produktu. W tym podejściu klasa *AbstractFactory* potrzebuje jedynie pojedynczej operacji *Make* z parametrem określającym rodzaj tworzonego obiektu. Tej techniki użyliśmy w omówionych wcześniej fabrykach abstrakcyjnych opartych na wzorcu Prototyp lub klasie.

Tę wersję łatwiej jest stosować w językach z dynamiczną kontrolą typów (takich jak Smalltalk) niż w językach ze statyczną kontrolą typów (na przykład C++). W języku C++ rozwiązania tego można użyć tylko wtedy, jeśli wszystkie obiekty mają tą samą abstrakcyjną klasę bazową lub gdy klient, który zażądał produktów, może bezpiecznie przekształcić ich typ na właściwy. W punkcie Implementacja poświęconym wzorcowi Metoda wytwarzająca (s. 110) pokazujemy, jak zaimplementować takie sparametryzowane operacje w języku C++.

Jednak nawet kiedy przekształcanie na właściwy typ nie jest konieczne, pozostaje do rozwiązania pewien problem — wszystkie produkty przekazywane do klienta mają *ten sam* abstrakcyjny interfejs określony przez zwracany typ. Dlatego klient nie może rozróżnić klas produktów ani dokonywać bezpiecznych założeń na ich temat. Jeśli klient musi wykonać operacje specyficzne dla podklasy, nie będzie mógł uzyskać dostępu do nich za pośrednictwem abstrakcyjnego interfejsu. Choć klient może przeprowadzić rzutowanie w dół (na przykład za pomocą instrukcji *dynamic_cast* w języku C++), nie zawsze jest to wykonalne lub bezpieczne, ponieważ operacja ta może zakończyć się niepowodzeniem. Jest to typowy koszt utworzenia wysoce elastycznego i rozszerzalnego interfejsu.

PRZYKŁADOWY KOD

Zastosujmy wzorzec Fabryka abstrakcyjna do utworzenia labiryntów opisanych w początkowej części rozdziału.

Klasa MazeFactory służy do tworzenia elementów labiryntów — pomieszczeń, ścian i drzwi między pokojami. Można użyć jej w programie, który wczytuje plany labiryntów z pliku i tworzy odpowiednie labirynty. Ponadto można wykorzystać ją w aplikacji generującej labirynty w sposób losowy. Programy, które tworzą labirynty, przyjmują obiekt MazeFactory jako argument, dzięki czemu programista może określić generowane pomieszczenia, ściany i drzwi.

```
class MazeFactory {
public:
    MazeFactory();

    virtual Maze* MakeMaze() const
    { return new Maze; }
    virtual Wall* MakeWall() const
    { return new Wall; }
    virtual Room* MakeRoom(int n) const
    { return new Room(n); }
    virtual Door* MakeDoor(Room* r1, Room* r2) const
    { return new Door(r1, r2); }
};
```

Przypomnijmy, że funkcja składowa CreateMaze (s. 90) tworzy mały labirynt składający się z dwóch pomieszczeń i drzwi między nimi. W tej funkcji nazwy klas zapisane są na stałe, co utrudnia generowanie labiryntów o różnych elementach.

Oto wersja operacji CreateMaze, w której rozwiązałismy ten problem przez zastosowanie obiektu MazeFactory jako parametru:

```
Maze* MazeGame::CreateMaze (MazeFactory& factory) {
    Maze* aMaze = factory.MakeMaze();
    Room* r1 = factory.MakeRoom(1);
    Room* r2 = factory.MakeRoom(2);
    Door* aDoor = factory.MakeDoor(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, factory.MakeWall());
    r1->SetSide(East, aDoor);
    r1->SetSide(South, factory.MakeWall());
    r1->SetSide(West, factory.MakeWall());

    r2->SetSide(North, factory.MakeWall());
    r2->SetSide(East, factory.MakeWall());
    r2->SetSide(South, factory.MakeWall());
    r2->SetSide(West, aDoor);

    return aMaze;
}
```

Możemy utworzyć klasę `EnchantedMazeFactory` (fabrykę magicznych labiryntów) jako podklasę klasy `MazeFactory`. Klasa `EnchantedMazeFactory` powinna przesłaniać kilka funkcji składowych i zwracać różne podklasy klas `Room`, `Wall` itd.

```
class EnchantedMazeFactory : public MazeFactory {
public:
    EnchantedMazeFactory();

    virtual Room* MakeRoom(int n) const
    { return new EnchantedRoom(n, CastSpell()); }

    virtual Door* MakeDoor(Room* r1, Room* r2) const
    { return new DoorNeedingSpell(r1, r2); }

protected:
    Spell* CastSpell() const;
};
```

Teraz założymy, że chcemy utworzyć grę z labiryntem, w której w pomieszczeniach mogą znajdować się bomby. Jeśli bomba wybucha, uszkodzi co najmniej ściany. Możemy dodać podkласę klasy `Room` służącą do rejestrowania, czy w pokoju znajduje się bomba i czy już wybuchła. Potrzebna będzie też podklasa klasy `Wall` do śledzenia uszkodzeń ścian. Nazwijmy te klasy `RoomWithABomb` i `BombedWall`.

Ostatnia klasa, którą zdefiniujemy, to `BombedMazeFactory`. Jest to podklasa klasy `MazeFactory` gwarantująca, że ściany to obiekty `BombedWall`, a pomieszczenia to obiekty `RoomWithABomb`. W klasie `BombedMazeFactory` trzeba przesłonić tylko dwie funkcje:

```
Wall* BombedMazeFactory::MakeWall () const {
    return new BombedWall;
}

Room* BombedMazeFactory::MakeRoom(int n) const {
    return new RoomWithABomb(n);
}
```

Aby zbudować prosty labirynt zawierający bomby, wystarczy wywołać operację `CreateMaze` i przekazać do niej obiekt klasy `BombedMazeFactory`.

```
MazeGame game;
BombedMazeFactory factory;

game.CreateMaze(factory);
```

Operacja `CreateMaze` może przyjmować także egzemplarz klasy `EnchantedMazeFactory`, jeśli ma utworzyć magiczny labirynt.

Zauważmy, że klasa `MazeFactory` jest jedynie kolekcją metod wytwarzających. Jest to najczęściej stosowany sposób implementowania wzorca Fabryka abstrakcyjna. Ponadto warto zwrócić uwagę na to, że klasa `MazeFactory` nie jest abstrakcyjna. Dlatego pełni jednocześnie funkcje klas `AbstractFactory` oraz `ConcreteFactory`. Jest to następna często używana implementacja w prostych zastosowaniach wzorca Fabryka abstrakcyjna. Ponieważ `MazeFactory` to klasa konkretna składająca się w całości z metod wytwarzających, łatwo jest utworzyć nową klasę tego rodzaju przez utworzenie podklasy i przesłonięcie operacji, które trzeba zmodyfikować.

W operacji CreateMaze wykorzystaliśmy operację SetSide obiektów Room do określenia stron w tych obiektach. Jeśli operacja CreateMaze tworzy pomieszczenia za pomocą klasy BombedMazeFactory, labirynt będzie składał się z obiektów RoomWithABomb ze stronami typu BombedWall. Jeśli obiekt RoomWithABomb będzie musiał uzyskać dostęp do specyficznej dla podklasy składowej obiektu BombedWall, konieczne będzie zrzutowanie referencji do ścian z typu Wall* na BombedWall*. To rzutowanie w dół jest bezpieczne, jeśli argument rzeczywiście ma typ BombedWall. Jest to pewne, jeżeli ściany są zbudowane wyłącznie za pomocą klasy BombedMazeFactory.

Języki z dynamiczną kontrolą typu, na przykład Smalltalk, oczywiście nie wymagają rzutowania w dół, jednak mogą generować błędy czasu wykonania, jeśli natrafią na obiekt Wall w miejscu, gdzie oczekują podklasy klasy Wall. Wykorzystanie przy tworzeniu ścian wzorca Fabryka abstrakcyjna pomaga zapobiec podobnym błędom czasu wykonania, ponieważ mamy wtedy pewność, że program utworzy ściany określonego typu.

Rozważmy wersję klasy MazeFactory w języku Smalltalk. Obejmuje ona jedną operację make, która przyjmuje jako parametr rodzaj generowanego obiektu. Ponadto fabryka konkretna przechowuje klasy tworzonych produktów.

Najpierw należy napisać odpowiednik operacji CreateMaze w języku Smalltalk:

```
CreateMaze: aFactory
| room1 room2 aDoor |
room1 := (aFactory make: #room) number: 1.
room2 := (aFactory make: #room) number: 2.
aDoor := (aFactory make: #door) from: room1 to: room2.
room1 atSide: #tnorth put: (aFactory make: #wall).
room1 atSide: #east put: aDoor.
room1 atSide: #tsouth put: (aFactory make: #wall).
room1 atSide: #twest put: (aFactory make: #wall).
room2 atSide: #north put: (aFactory make: #wall).
room2 atSide: #east put: (aFactory make: #wall).
room2 atSide: #tsouth put: (aFactory make: #wall).
room2 atSide: #twest put: aDoor.
^ Maze new addRoom: room1; addRoom: room2; yourself
```

Klasa MazeFactory — jak opisaliśmy to w punkcie Implementacja — wymaga tylko jednej zmiennej egzemplarza, partCatalog, aby udostępnić katalog, którego kluczami są klasy komponentów labiryntu. Przypomnijmy też, jak zaimplementowaliśmy metodę make::

```
make: partName
^ (partCatalog at: partName) new
```

Teraz można utworzyć obiekt MazeFactory i wykorzystać go do zaimplementowania operacji createMaze. Do utworzenia fabryki posłuży metoda createMazeFactory klasy MazeGame.

```
createMazeFactory
^ (MazeFactory new
    addPart: Wall named: #wall;
    addPart: Room named: #room;
    addPart: Door named: #door;
    yourself)
```

Obiekty BombedMazeFactory i EnchantedMazeFactory są tworzone przez powiązanie różnych klas z odpowiednimi kluczami. Na przykład obiekt EnchantedMazeFactory można utworzyć tak:

```
createMazeFactory
^ (MazeFactory new
  addPart: Wall named: #wall;
  addPart: EnchantedRoom named: #room;
  addPart: DoorNeedingSpell named: #door;
  yourself)
```

ZNANE ZASTOSOWANIA

W pakiecie InterViews do określania klas `AbstractFactory` służy przyrostek `Kit` [Lin92]. Pakiet ten obejmuje definicje fabryk abstrakcyjnych `WidgetKit` i `DialogKit` generujących obiekty interfejsu użytkownika specyficzne dla danego standardu wyglądu i działania. Pakiet InterViews obejmuje też klasę `LayoutKit`, która w zależności od wybranego układu tworzy różne obiekty złożone. Na przykład układ poziomy może wymagać zastosowania odmiennych obiektów złożonych w zależności od orientacji dokumentu (pionowej lub poziomej).

W platformie ET++ [WGM88] wzorzec Fabryka abstrakcyjna zastosowano do zapewnienia przenośności rozwiązań między różnymi systemami okienkowymi (na przykład X Windows i SunView). Abstrakcyjna klasa bazowa `WindowSystem` definiuje interfejs do tworzenia obiektów reprezentujących zasoby systemów okienkowych (interfejs ten obejmuje na przykład operacje `MakeWindow`, `SetFont`, `SetColor` itd.). W podklasach konkretnych interfejs ten jest zaimplementowany dla określonych systemów okienkowych. W czasie wykonywania programu platforma ET++ tworzy egzemplarz podklasy konkretnej klasy `WindowSystem`, a egzemplarz ten generuje obiekty konkretne reprezentujące zasoby systemowe.

POWIĄZANE WZORCE

Fabryki abstrakcyjne często są implementowane za pomocą metod wytwarzających (Metoda wytwarzająca, s. 110), jednak można wykorzystać do tego także wzorzec Prototyp (s. 120).

Fabryki konkretne są często singletonami (Singleton, s. 130).

METODA WYTWÓRCZA (FACTORY METHOD)

klasowy, konstrukcyjny

PRZEZNACZENIE

Określa interfejs do tworzenia obiektów, przy czym umożliwia podklasom wyznaczenie klasy danego obiektu. Metoda wytwórcza umożliwia klasom przekazanie procesu tworzenia egzemplarzy podklasom.

INNE NAZWY

Konstruktor wirtualny (ang. *virtual constructor*).

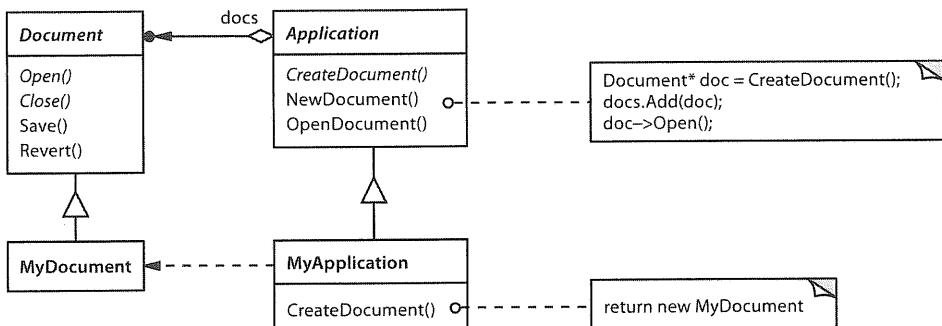
UZASADNIENIE

W platformach klasy abstrakcyjne służą do definiowania i podtrzymywania relacji między obiektemi. Platforma często odpowiada także za tworzenie obiektów.

Zastanówmy się nad platformą dla aplikacji potrafiących wyświetlać wiele dokumentów. Dwie kluczowe abstrakcje w tej platformie to klasy *Application* i *Document*. Obie te klasy są abstrakcyjne, a w klientach trzeba utworzyć ich podklasy i umieścić tam implementacje specyficzne dla aplikacji. Aby utworzyć aplikację do rysowania, należy zdefiniować klasy *DrawingApplication* i *DrawingDocument*. Klasa *Application* odpowiada za zarządzanie obiektami *Document* i tworzy je na żądanie (na przykład kiedy użytkownik wybierze z menu opcję *Otwórz* lub *Nowy*).

Ponieważ określona podkласa klasy *Document*, której egzemplarz należy utworzyć, jest specyficzna dla aplikacji, w klasie *Application* nie można z góry ustalić rodzaju tej podklasy. Klasa *Application* potrafi jedynie określić, kiedy należy utworzyć nowy dokument, a nie *jakiego rodzaju* powinien on być. Stawia nas to przed dilemma — platforma musi tworzyć egzemplarze klas, ale ma informacje tylko o klasach abstrakcyjnych, których egzemplarzy wygenerować nie może.

Rozwiązaniem jest zastosowanie wzorca Metoda wytwórcza. Pozwala on zakapsułkować informacje o tym, która podklaśe klasy *Document* należy utworzyć, i zapisać te dane poza platformą.



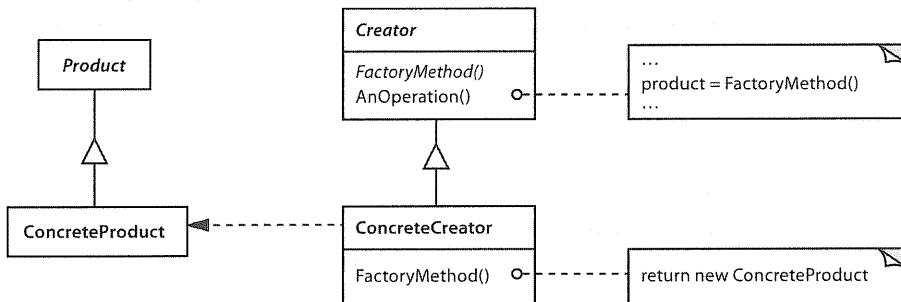
W podklasach klasy Application należy przedefiniować operację CreateDocument klasy Application, tak aby nowa wersja operacji zwracała odpowiednią podkласę klasy Document. Egzemplarz podklasy klasy Application może następnie generować specyficzne dla aplikacji egzemplarze klasy Document bez znajomości ich klasy. Operację CreateDocument nazywamy **metodą wytwarzającą**, ponieważ odpowiada za tworzenie obiektów.

WARUNKI STOSOWANIA

Wzorca Metoda wytwarzająca należy używać w następujących warunkach:

- ▶ Kiedy w danej klasie nie można z góry ustalić klasy obiektów, które trzeba utworzyć.
- ▶ Jeśli programista chce, aby to podklasy danej klasy określały tworzone przez nią obiekty.
- ▶ Jeżeli klasy delegują zadania do jednej z kilku podklas pomocniczych, a programista chce zapisać w określonym miejscu informacje o tym, która z tych podklas jest delegatem.

STRUKTURA



ELEMENTY

- ▶ **Product (Document)**, czyli produkt:
 - definiuje interfejs obiektów generowanych przez metodę wytwarzającą.
- ▶ **ConcreteProduct (MyDocument)**, czyli produkt konkretny:
 - obejmuje implementację interfejsu klasy Product.
- ▶ **Creator (Application)**, czyli wytwarzca:
 - obejmuje deklarację metody wytwarzającej zwracającej obiekty typu Product; w obiekcie Creator można też zdefiniować implementację domyślnej metody fabrycznej, zwracającą domyślny obiekt CreateProduct;
 - może wywoływać metodę wytwarzającą w celu wygenerowania obiektu Product.
- ▶ **ConcreteCreator (MyApplication)**, czyli wytwarzca konkretny:
 - przesyłania metodę wytwarzającą, tak aby zwracała egzemplarz klasy ConcreteProduct.

WSPÓŁDZIAŁANIE

- Klasa `Creator` działa na podstawie założenia, że w jej podklasach zdefiniowana jest metoda wytwórcza zwracająca egzemplarz odpowiedniej klasy `ConcreteProduct`.

KONSEKWENCJE

Metoda wytwórcza eliminuje konieczność wiązania klas specyficznych dla aplikacji z kodem. W kodzie używany jest tylko interfejs klasy `Product`, dlatego działać w nim będzie dowolna zdefiniowana przez użytkownika klasa `ConcreteProduct`.

Potencjalną wadą metody wytwórczej jest to, że klienci czasem muszą tworzyć podklasy klasy `Creator` tylko w celu wygenerowania określonego obiektu `ConcreteProduct`. Nie ma nic złego w tworzeniu podklas, jeśli w kliencie i tak trzeba dodać takie podklasy dla klasy `Creator`. Jednak jeżeli jest inaczej, w kliencie trzeba wprowadzić dodatkowe zmiany.

Oto dwie następne konsekwencje zastosowania wzorca Metoda wytwórcza:

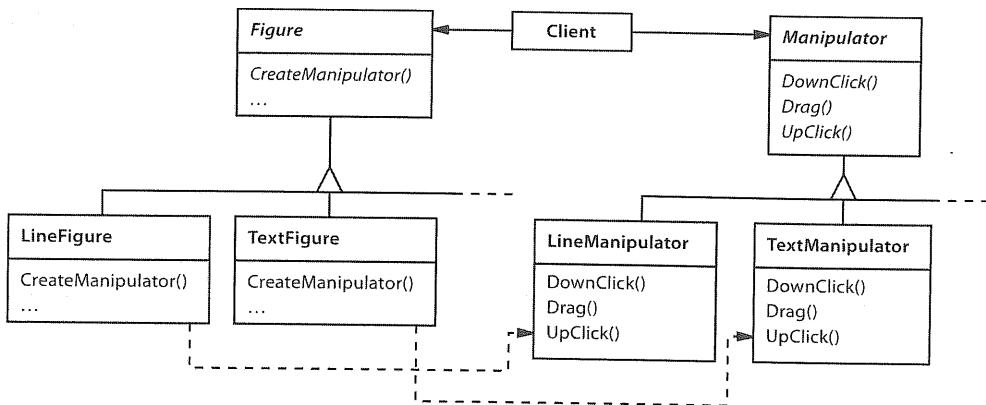
1. *Zapewnienie punktów zaczepienia dla podklas.* Tworzenie obiektów w klasie za pomocą metody wytwórczej zawsze daje większą elastyczność niż bezpośrednie ich generowanie. Wzorzec Metoda wytwórcza zapewnia punkty zaczepienia dla podklas na potrzeby tworzenia wzbo-gaconej wersji obiektu.

W przykładzie dotyczącym klasy `Document` moglibyśmy zdefiniować w niej metodą wytwórczą o nazwie `CreateFileDialog` generującą domyślny obiekt okna dialogowego służący do otwierania istniejących dokumentów. W podklasie klasy `Document` można zdefiniować specyficzne dla aplikacji okno dialogowe przez przesłonięcie wspomnianej metody wytwórczej. W tym przykładzie metoda wytwórcza nie jest abstrakcyjna, ponieważ udo-stępnia przydatną implementację domyślną.

2. *Połączenie równoległych hierarchii klas.* W przykładach omówionych do tej pory metodę fabryczną wywołują tylko obiekty `Creator`. Jednak nie zawsze musi tak być. Metoda wytwórcza może okazać się przydatna także dla klientów (zwłaszcza w systemach z równole-głymi hierarchiami klas).

Równoległe hierarchie klas powstają, kiedy klasa deleguje część zadań do odrębnej klasy. Rozważmy figury graficzne, którymi można interaktywnie manipulować — rozciągać je, przenosić lub rotować za pomocą myszy. Zaimplementowanie takich interakcji nie zawsze jest łatwe. Często wymaga to zapisywania i aktualizowania informacji o stanie zmian w danym momencie. Ten stan jest niezbędny tylko w czasie manipulowania elementem, dlatego nie trzeba go przechowywać w obiekcie reprezentującym figurę. Ponadto poszczególne figury zachowują się inaczej w czasie manipulowania nimi przez użytkownika. Na przykład rozciąganie odcinka może doprowadzić do przesunięcia jego punktu końcowego, a rozciąganie tekstu — do zmiany wysokości interlinii.

Z uwagi na te ograniczenia lepiej jest użyć odrębnego obiektu `Manipulator` i zaimple-mentować w nim obsługę interakcji oraz przechowywać tam stan specyficzny dla mani-pulacji. Poszczególne figury będą korzystać z różnych podklas klasy `Manipulator` do ob-sługi określonych interakcji. Ostateczna hierarchia klasy `Manipulator` jest równoległą (przynajmniej w części) do hierarchii klasy `Figure`.



Klasa **Figure** udostępnia metodę wytwarzającą **CreateManipulator**. Umożliwia ona klientom tworzenie obiektów **Manipulator** odpowiadających podklasom klasy **Figure**. W tych podklasach omawiana metoda jest przesłonięta, tak aby zwracała egzemplarz odpowiedniego dla nich podklassy klasy **Manipulator**. Inną możliwość to zaimplementowanie w klasie **Figure** metody **CreateManipulator** w taki sposób, żeby zwracała domyślny egzemplarz klasy **Manipulator**. Wtedy podklasy klasy **Figure** mogą odziedziczyć tę domyślną implementację. Takie podklasy nie potrzebują powiązanych z nimi podklass klasy **Manipulator**, dlatego hierarchie są tylko częściowo równoległe.

Zauważmy, w jaki sposób metoda wytwarzająca łączy dwie hierarchie klas. Obejmuje ona informacje o tym, które klasy są powiązane ze sobą.

IMPLEMENTACJA

Przy stosowaniu wzorca Metoda wytwórcza należy uwzględnić następujące zagadnienia:

1. *Dwie główne odmiany.* Dwa podstawowe warianty wzorca Metoda wytwórcza to: (1) utworzenie klasy **Creator** jako klasy abstrakcyjnej i pominięcie w niej implementacji zadeklarowanej metody wytwórczej oraz (2) utworzenie klasy **Creator** jako klasy konkretnej i umieszczenie w niej domyślnej implementacji metody wytwórczej. Można też utworzyć klasę abstrakcyjną z definicją implementacji domyślnej, jednak jest to rzadziej stosowane rozwiązanie.

W pierwszym przypadku w podklasie *trzeba* zdefiniować implementację, ponieważ nie istnieje przydatna implementacja domyślna. Pozwala to rozwiązać problem tworzenia egzemplarzy nieprzewidzianych klas. W drugiej sytuacji umieszczenie metody wytwórczej w konkretnej klasie **Creator** służy przede wszystkim zwiększeniu elastyczności. Podejście to jest zgodne z następującą zasadą: „Twórz obiekty za pomocą odrębnej operacji, aby można przesłonić sposób ich generowania w podklassach”. Ta reguła gwarantuje, że projektanci podklas będą mogli w razie potrzeby zmienić klasę obiektów generowanych przez klasę nadziedzinną.

2. *Sparametryzowane metody wytwórcze.* Inna odmiana wzorca umożliwia metodom wytwarzającym generowanie produktów *wielu* rodzajów. Metoda wytwórcza przyjmuje wtedy parametr określający rodzaj generowanego obiektu. Wszystkie obiekty tworzone przez taką

metodę wytwórczą będą miały wspólny interfejs klasy `Product`. W przykładzie dotyczącym klasy `Document` klasa `Application` może obsługiwać różne rodzaje obiektów `Document`. Aby określić specyficzny typ dokumentu, należy przekazać do operacji `CreateDocument` dodatkowy parametr.

W platformie Unidraw [VL90] (służy ona do tworzenia aplikacji z funkcją edycji w trybie graficznym) podejście to zastosowano do odtwarzania obiektów zapisanych na dysku. Platforma ta obejmuje definicję klasy `Creator` z metodą wytwórczą `Create` przyjmującą jako argument identyfikator klasy. Ten identyfikator określa klasę, której egzemplarz należy utworzyć. Kiedy platforma zapisuje obiekt na dysku, najpierw rejestruje identyfikator klasy, a następnie zmienne egzemplarza. W czasie odtwarzania obiektu najpierw wczytuje identyfikator klasy.

Po wczytaniu identyfikatora klasy platforma wywołuje operację `Create` i przekazuje do niej identyfikator jako parametr. Operacja `Create` wyszukuje konstruktor odpowiedniej klasy i wykorzystuje go do utworzenia egzemplarza danej klasy. W ostatnim kroku `Create` wywołuje operację `Read` obiektu, co powoduje wczytanie pozostałych informacji z dysku i zainicjowanie zmiennych egzemplarza.

Sparametryzowana metoda wytwórcza ma następującą ogólną postać (`MyProduct` i `YourProduct` są tu podklasami klasy `Product`).

```
class Creator {
public:
    virtual Product* Create(ProductId);
};

Product* Creator::Create (ProductId id) {
    if (id == MINE) return new MyProduct;
    if (id == YOURS) return new YourProduct;
    // Powtarzane dla pozostałych produktów.

    return 0;
}
```

Przesłonięcie spараметryzowanej metody wytwórczej pozwala łatwo i wybiorczo rozszerzać lub modyfikować produkty tworzone przez klasę `Creator`. Można wprowadzić nowe identyfikatory dla produktów nowego rodzaju lub powiązać istniejące identyfikatory z innymi produktami.

Na przykład w podklasie `MyCreator` można zastąpić miejscami klasy `MyProduct` i `YourProduct` oraz dodać obsługę nowej podklasy `TheirProduct`.

```
Product* MyCreator::Create (ProductId id) {
    if (id == YOURS) return new MyProduct;
    if (id == MINE) return new YourProduct;
    // Uwaga — identyfikatory YOURS i MINE zamieniono miejscami.

    if (id == THEIRS) return new TheirProduct;

    return Creator::Create(id); // Wywoływana, jeśli żaden z warunków nie jest spełniony.
}
```

Zauważmy, że ostatnim zadaniem wykonywanym przez tę operację jest wywołanie operacji `Create` z klasy nadzędnej. Dzieje się tak, ponieważ operacja `MyCreator::Create` obsługuje w specyficzny sposób (inaczej niż klasa nadzędna) jedynie identyfikatory `YOUR`, `MINE` i `THEIRS`. Inne klasy nie są tu uwzględniane. Dlatego klasa `MyCreator` rozszerza listę tworzonych produktów i przekazuje zadanie generowania większości z nich klasie nadzędnej.

3. *Varianty i problemy specyficzne dla języka.* Z poszczególnymi językami programowania związane są inne ciekawe odmiany i zastrzeżenia.

W programach w języku Smalltalk często używana jest metoda zwracająca klasę, której egzemplarz należy utworzyć. W metodzie wytwórczej w klasie `Creator` można wykorzystać tę wartość do utworzenia produktu, a klasa `ConcreteCreator` może przechowywać, a nawet obliczać tę wartość. W efekcie określanie typu tworzonego egzemplarza podklasy klasy `ConcreteProduct` ma miejsce jeszcze później.

W napisanej w języku Smalltalk wersji przykładu dotyczącego klasy `Document` można w klasie `Application` zdefiniować metodę `documentClass`. Metoda ta powinna zwracać odpowiednią klasę `Document`, której egzemplarz należy utworzyć. Implementacja metody `documentClass` w klasie `MyApplication` zwraca klasę `MyDocument`. Dlatego w klasie `Application` należy umieścić następujący kod:

```
clientMethod
    document := self documentClass new.

documentClass
    self subclassResponsibility
```

Klasa `MyApplication` obejmuje poniższy kod:

```
documentClass
    ^ MyDocument
```

Ten fragment zwraca do klasie `Application` klasę `MyDocument`, której egzemplarz należy utworzyć.

Jeszcze elastyczniejsze rozwiązanie, zbliżone do sparametryzowanych metod wytwórczych, polega na przechowywaniu klasy tworzonych obiektów w zmiennej statycznej klasy `Application`. Pozwala to uniknąć tworzenia podklasy klasy `Application` w celu zmodyfikowania produktu.

Metody wytwórcze w języku C++ zawsze są funkcjami wirtualnymi (często czysto wirtualnymi). Należy jednak zachować ostrożność i nie wywoływać metod wytwórczych w konstruktorze klasy `Creator`, ponieważ metoda wytwórcza klasy `ConcreteCreator` nie będzie wtedy jeszcze dostępna.

Można uniknąć tego problemu dzięki zachowaniu staranności i korzystaniu z produktów wyłącznie za pośrednictwem akcesora tworzącego dany produkt na żądanie. W konstruktorze zamiast generować konkretny produkt, należy zainicjować go za pomocą wartości 0. Do zwrócenia produktu posłuży akcesor. Najpierw jednak sprawdzi, czy produkt istnieje, a jeśli nie — utworzy go. Ta technika jest czasem nazywana **leniwym inicjowaniem**. Poniższy kod ilustruje typową implementację tego rozwiązania.

```

class Creator {
public:
    Product* GetProduct();
protected:
    virtual Product* CreateProduct();
private:
    Product* _product;
};

Product* Creator::GetProduct () {
    if (_product == 0) {
        _product = CreateProduct();
    }
    return _product;
}

```

4. Wykorzystanie szablonów w celu uniknięcia tworzenia podklas. Wspomnieliśmy już, że następującym potencjalnym problemem związanym z metodami wytwórczymi jest to, iż czasem trzeba utworzyć podklasę tylko w celu utworzenia odpowiednich obiektów Product. Inny sposób na poradzenie sobie z tą niedogodnością w języku C++ polega na udostępnieniu szablonu podklasy klasy Creator sparametryzowanego za pomocą klasy Product.

```

class Creator {
public:
    virtual Product* CreateProduct() = 0;
};

template <class TheProduct>
class StandardCreator: public Creator {
public:
    virtual Product* CreateProduct();
};

template <class TheProduct>
Product* StandardCreator<TheProduct>::CreateProduct () {
    return new TheProduct;
}

```

Dzięki temu szablonowi klient może podać samą klasę produktu — tworzenie podklasy klasy Creator nie jest konieczne.

```

class MyProduct : public Product {
public:
    MyProduct();
    // ...
};

StandardCreator<MyProduct> myCreator;

```

5. Konwencje nazewnicze. Dobrym zwyczajem jest stosowanie konwencji nazewniczych wyraźnie wskazujących na zastosowanie metod wytwórczych. Na przykład w platformie MacApp [App89] (służy ona do tworzenia aplikacji na komputery Macintosh) operacja abstrakcyjna definiująca metodą wytwórczą zawsze deklarowana jest w postaci Class* DoMakeClass(), gdzie Class to nazwa klasy produktu.

PRZYKŁADOWY KOD

Funkcja `CreateMaze` (s. 90) tworzy i zwraca labirynt. Jeden ze związków z nią problemów polega na tym, że zapisano w niej na stałe klasy labiryntu, pomieszczeń, drzwi i ścian. Zastosujemy metodę wytwarzającą, aby umożliwić zmodyfikowanie tych komponentów w podklasach.

Najpierw zdefiniujmy metody wytwarzające w klasie `MazeGame`. Posłużą one do tworzenia obiektów reprezentujących labirynt, pomieszczenie, ścianę i drzwi.

```
class MazeGame {
public:
    Maze* CreateMaze();

// Metody wytwarzające:

    virtual Maze* MakeMaze() const
        { return new Maze; }
    virtual Room* MakeRoom(int n) const
        { return new Room(n); }
    virtual Wall* MakeWall() const
        { return new Wall; }
    virtual Door* MakeDoor(Room* r1, Room* r2) const
        { return new Door(r1, r2); }
};
```

Każda metoda wytwarzająca zwraca komponent określonego rodzaju. Klasa `MazeGame` udostępnia implementację domyślną zwracającą labirynt, pomieszczenia, ściany i drzwi najprostszego rodzaju.

Teraz można zmodyfikować operację `CreateMaze` z wykorzystaniem metod wytwarzających.

```
Maze* MazeGame::CreateMaze () {
    Maze* aMaze = MakeMaze();

    Room* r1 = MakeRoom (1);
    Room* r2 = MakeRoom(2);
    Door* theDoor = MakeDoor(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, MakeWall());
    r1->SetSide(East, theDoor);
    r1->SetSide(South, MakeWall());
    r1->SetSide(West, MakeWall());

    r2->SetSide(North, MakeWall());
    r2->SetSide(East, MakeWall());
    r2->SetSide(South, MakeWall());
    r2->SetSide(West, theDoor);

    return aMaze;
}
```

W różnych grach można utworzyć podklasy klasy `MazeGame`, aby dodać wyspecjalizowane części labiryntu. W tych podklasach można przedefiniować niektóre (lub wszystkie) metody twórcze w celu określenia odmian produktów. Na przykład w klasie `BombedMazeGame` można umieścić nowe definicje produktów `Room` i `Wall`, tak aby metody zwracały wersje specyficzne dla labiryntu z bombami.

```
class BombedMazeGame : public MazeGame {
public:
    BombedMazeGame();

    virtual Wall* MakeWall() const
    { return new BombedWall; }

    virtual Room* MakeRoom(int n) const
    { return new RoomWithABomb(n); }
};
```

Podklaę `EnchantedMazeGame` można zdefiniować w następujący sposób:

```
class EnchantedMazeGame : public MazeGame {
public:
    EnchantedMazeGame();

    virtual Room* MakeRoom(int n) const
    { return new EnchantedRoom(n, CastSpell()); }

    virtual Door* MakeDoor(Room* r1, Room* r2) const
    { return new DoorNeedingSpell(r1, r2); }
protected:
    Spell* CastSpell() const;
};
```

ZNANE ZASTOSOWANIA

Metody twórcze są powszechnie stosowane w pakietach narzędziowych i platformach. Wcześniej przykład dotyczący dokumentu ilustruje typowe zastosowanie wzorca w platformach MacApp i ET++ [WGM88]. Przykład opisujący manipulatory pochodzi z platformy Unidraw.

Klasa `View` w architekturze MVC języka Smalltalk-80 obejmuje metodę `defaultController`. Tworzy ona kontroler, dlatego można traktować ją jak metodę twórczą [Par90]. Jedna w podklasach klasy `View` klasa ich kontrolera domyślnego jest określana za pomocą metod `defaultControllerClass` (zwraca ona klasę, której egzemplarz tworzy metoda `defaultController`). Dlatego prawdziwą metodą twórczą, czyli tą, którą należy przesłonić w podklasach, jest `defaultControllerClass`.

Bardziej wymyślny przykład z języka Smalltalk-80 to metoda twórcza `parserClass` zdefiniowana w klasie `Behavior` (jest to nadklasa wszystkich obiektów reprezentujących klasy). To rozwiązanie umożliwia klasom wykorzystanie niestandardowego parsera do analizy kodu źródłowego. W kliencie można na przykład zdefiniować klasę `SQLParser` do analizowania kodu źródłowego klasy z zagnieżdżonymi instrukcjami w języku SQL. Metoda `parserClass`

zaimplementowana w klasie `Behavior` zwraca standardową klasę `Parser` języka Smalltalk. W klasie z zagnieżdżonymi instrukcjami w języku SQL należy przesłonić tę metodę (jako metodę statyczną), tak aby zwracała klasę `SQLParser`.

W systemie Orbix ORB firmy IONA Technologies [ION94] wzorzec Metoda wytwórcza wykorzystano do generowania pełnomocników odpowiedniego typu (zobacz wzorzec Pełnomocnik, s. 191) w odpowiedzi na żądanie referencji do zdalnego obiektu. Metoda wytwórcza sprawia, że można łatwo zastąpić domyślnego pełnomocnika inną wersją, na przykład używającą pamięci podręcznej po stronie klienta.

POWIĄZANE WZORCE

Wzorzec Fabryka abstrakcyjna (s. 101) jest często implementowany za pomocą metod wytwórczych. Przykład w punkcie „Uzasadnienie” w opisie wzorca Fabryka abstrakcyjna ilustruje także zastosowania wzorca Metoda wytwórcza.

Metody wytwórcze zwykle wywołuje się w metodach szablonowych (s. 264). We wcześniejszym przykładzie dotyczącym dokumentu `NewDocument` to metoda szablonowa.

Prototypy (s. 120) nie wymagają tworzenia podklas klasy `Creator`, jednak często konieczne jest wtedy umieszczenie operacji `Initialize` w klasie `Product`. W klasie `Creator` operacja `Initialize` służy do inicjowania obiektu. Metoda wytwórcza nie wymaga stosowania takich operacji.

PROTOTYP (PROTOTYPE)

obiektowy, konstrukcyjny

PRZEZNACZENIE

Określa na podstawie prototypowego egzemplarza rodzaje tworzonych obiektów i generuje nowe obiekty przez kopiowanie tego prototypu.

UZASADNIENIE

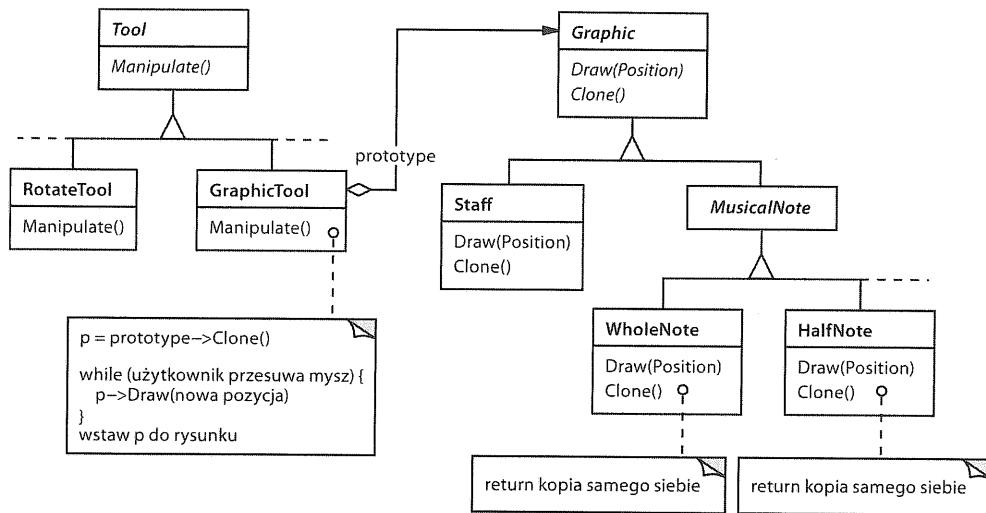
Można zbudować edytor partytur przez dostosowanie ogólnej platformy do tworzenia edytorów graficznych i dodanie nowych obiektów reprezentujących nuty, pauzy i pięciolinie. Platforma do tworzenia edytorów może udostępniać paletę narzędzi do dodawania takich obiektów muzycznych do partytury. W palecie mogą znaleźć się też narzędzia do zaznaczania i przenoszenia tych obiektów oraz manipulowania nimi. Użytkownik mógłby na przykład kliknąć narzędzie związane z ćwierćnutami i użyć go do dodania ćwierćnuta do partytury. Mógłby też wykorzystać narzędzie do przenoszenia, aby przesunąć nutę w górę lub w dół na pięciolinii i zmienić w ten sposób wysokość dźwięku.

Załóżmy, że omawiana platform udostępnia klasę abstrakcyjną `Graphic` reprezentującą komponenty graficzne, takie jak nuty i pięciolinie. Ponadto obejmuje klasę abstrakcyjną `Tool`, która służy do definiowania w palecie narzędzi podobnych do tych omówionych wcześniej. W platformie należy ponadto zdefiniować podklasę `GraphicTool` reprezentującą narzędzia do tworzenia egzemplarzy obiektów graficznych i dodawania ich do dokumentu.

Jednak klasa `GraphicTool` może sprawić problemy projektantowi platformy. Klasy reprezentujące nuty i pięciolinie są specyficzne dla aplikacji, natomiast klasa `GraphicTool` należy do platformy, dlatego nie potrafi tworzyć egzemplarzy klas muzycznych dodawanych do partytury. Moglibyśmy przygotować podkласę klasy `GraphicTool` dla obiektu muzycznego każdego rodzaju, jednak prowadzi to do utworzenia wielu podklas różniących się jedynie rodzajem generowanego obiektu. Wiemy, że składanie obiektów to elastyczna alternatywa dla tworzenia podklas. Pozostaje jednak pytanie, jak wykorzystać tę technikę w platformie, aby sparametryzować egzemplarze klasy `GraphicTool` za pomocą *klasy* z rodziny `Graphic`, której egzemplarz należy utworzyć.

Rozwiążanie polega na tworzeniu przez klasę `GraphicTool` nowego obiektu `Graphic` przez kopiowanie (lub klonowanie) egzemplarza podklasy klasy `Graphic`. Taki egzemplarz nazywamy **prototypem**. Klasa `GraphicTool` jest sparametryzowana za pomocą prototypu, który powinna sklonować i dodać do dokumentu. Jeśli wszystkie podklasy klasy `Graphic` obsługują operację `Clone`, klasa `GraphicTool` może sklonować dowolny obiekt z rodziny klas `Graphic`.

Dlatego w edytorze utworów muzycznych każde narzędzie do tworzenia obiektów muzycznych jest egzemplarzem klasy `GraphicTool` zainicjowanym za pomocą innego prototypu. Każdy egzemplarz klasy `GraphicTool` tworzy obiekt muzyczny przez klonowanie jego prototypu, a następnie dodaje kopię do partytury.



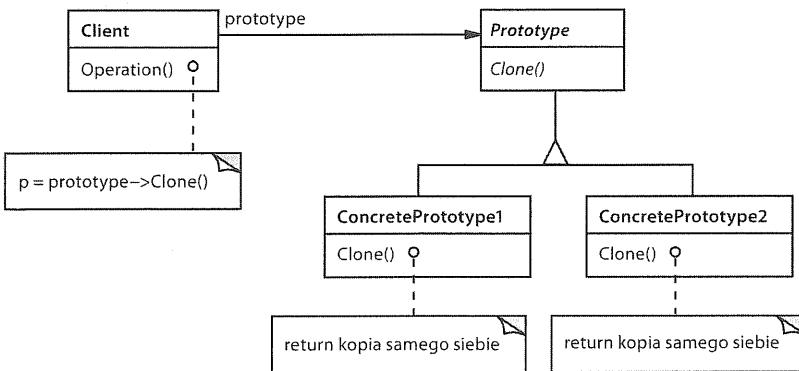
Możemy użyć wzorca Prototyp, aby jeszcze bardziej zmniejszyć liczbę klas. Odrębne klasy reprezentują całe nuty i półnuty, choć prawdopodobnie nie jest to konieczne. Zamiast tego można użyć egzemplarzy tej samej klasy zainicjowanych za pomocą różnych bitmap i czasu trwania dźwięku. Narzędzie do tworzenia całych nut będzie obiektem `GraphicTool`, którego prototyp to obiekt `MusicalNote` zainicjowany w taki sposób, aby reprezentował całą nutę. Pozwala to znacznie ograniczyć liczbę klas w systemie, a ponadto ułatwia dodawanie do edytora nut nowego rodzaju.

WARUNKI STOSOWANIA

Ze wzorca Prototyp należy korzystać, jeśli system powinien być niezależny od sposobu tworzenia, składania i reprezentowania produktów *orz*

- ▶ klasy tworzonych egzemplarzy są określone w czasie wykonywania programu (na przykład przez dynamiczne wczytywanie) *lub*
- ▶ programista chce uniknąć tworzenia hierarchii klas fabryk odpowiadającej hierarchii klas produktów *lub*
- ▶ egzemplarze klasy mogą przyjmować jeden z niewielu stanów; wygodniejsze może wtedy okazać się dodanie odpowiedniej liczby prototypów i klonowanie ich zamiast ręcznego tworzenia egzemplarzy klasy (za każdym razem z właściwym stanem).

STRUKTURA



ELEMENTY

- ▶ **Prototype (Graphic):**
 - obejmuje deklarację interfejsu do klonowania.
- ▶ **ConcretePrototype (Staff, WholeNote, HalfNote):**
 - obejmuje implementację procesu klonowania.
- ▶ **Client (GraphicTool)**
 - tworzy nowy obiekt przez zażądanie od prototypu sklonowania się.

WSPÓŁDZIAŁANIE

- ▶ Klient żąda od prototypu, aby ten się sklonował.

KONSEKWENCJE

Konsekwencje zastosowania Prototypu w dużej części pokrywają się ze skutkami użycia wzorców Fabryka abstrakcyjna (s. 101) i Budowniczy (s. 92). Wzorzec ten ukrywa klasy konkretne produktów przed klientem, zmniejszając w ten sposób liczbę nazw znanych klientom. Ponadto wzorce te umożliwiają klientowi korzystanie z klas specyficznych dla aplikacji bez konieczności modyfikowania go.

Poniżej wymieniamy dodatkowe korzyści płynące z zastosowania wzorca Prototyp:

1. *Możliwość dodawania i usuwania produktów w czasie wykonywania programu.* Prototypy umożliwiają dołączenie do systemu nowej klasy konkretnej produktu przez samo zarejestrowanie prototypowego egzemplarza w kliencie. Zapewnia to nieco większą elastyczność niż inne wzorce konstrukcyjne, ponieważ klient może instalować i usuwać prototypy w czasie wykonywania programu.
2. *Możliwość określania nowych obiektów przez zmianę wartości.* Wysokie dynamiczne systemy umożliwiają definiowanie nowych zachowań przez składanie obiektów — na przykład za pomocą określania wartości zmiennych — a nie przez definiowanie nowych klas. Nowe

rodzaje obiektów można definiować przez tworzenie egzemplarzy istniejących klas i rejestrowanie tych egzemplarzy jako prototypów obiektów klienta. Klient może wykorzystać nowe zachowanie przez oddelegowanie zadania do prototypu.

Projekt tego rodzaju umożliwia użytkownikom definiowanie nowych „klas” bez programowania. Klonowanie prototypu przypomina w swej istocie tworzenie egzemplarza klasy. Wzorzec Prototyp pozwala znacznie zmniejszyć liczbę klas potrzebnych w systemie. W edytorze utworów muzycznych jedna klasa `GraphicTool` może utworzyć nieskończoną liczbę różnorodnych obiektów muzycznych.

3. *Możliwość określania nowych obiektów przez modyfikowanie struktury.* Wiele aplikacji tworzy obiekt z mniej i bardziej złożonych części. Na przykład w edytorsach do projektowania obwodów elektrycznych można budować takie struktury z podkładów¹. Dla wygody aplikacje tego typu często umożliwiają tworzenie egzemplarzy złożonych struktur zdefiniowanych przez użytkownika, na przykład w celu wielokrotnego wykorzystania specyficznego podkładu.

Wzorzec Prototyp obsługuje także to rozwiązanie. Wystarczy dodać określony podkład jako prototyp do palety dostępnych składników układów. Jeśli w składanym obiekcie obwodu operacja `Clone` jest zaimplementowana z wykorzystaniem głębokiego kopiowania, układy o różnych strukturach można tworzyć jako prototypy.

4. *Zmniejszenie liczby podklas.* Wzorzec Metoda wytwarzca (s. 110) często powoduje utworzenie hierarchii klasy `Creator` odpowiadającej hierarchii klasy produktu. Wzorzec Prototyp pozwala skłonować prototyp, zamiast żądać od metody wytwarzcej utworzenia nowego obiektu. Dlatego hierarchia klasy `Creator` w ogóle nie jest potrzebna. Ta zaleta dotyczy głównie języków podobnych do C++, w których klasy nie są traktowane jak standardowe obiekty. W językach, które obsługują obiekty reprezentujące klasy (na przykład w językach Smalltalk i Objective C), te korzyści są mniejsze, ponieważ zawsze można użyć takiego obiektu do tworzenia klas. Obiekty reprezentujące klasy działają w tych językach jak prototypy.

5. *Możliwość dynamicznego konfigurowania aplikacji za pomocą klas.* Niektóre środowiska uruchomieniowe umożliwiają dynamiczne wczytywanie klas do aplikacji. Wzorzec Prototyp to klucz do wykorzystania takich mechanizmów w językach podobnych do C++.

Aplikacja, w której programista chce tworzyć egzemplarze dynamicznie wczytywanej klasy, nie będzie mogła statycznie wskazać jej konstruktora. Zamiast tego środowisko uruchomieniowe automatycznie utworzy egzemplarz każdej klasy w czasie jej wczytywania i zarejestruje ten egzemplarz za pomocą menedżera prototypów (zobacz punkt „Implementacja”). Następnie aplikacja może zażądać od menedżera prototypów egzemplarzy nowo wczytanych klas, które początkowo nie były dołączone do programu. Rozwiążanie to wykorzystano w systemie uruchomieniowym platformy do tworzenia aplikacji ET++ [EGM88].

Główną wadą wzorca Prototyp jest to, że w każdej podklasie klasy `Prototype` trzeba zaimplementować operację `Clone`. Może to sprawiać problemy. Dodanie tej operacji jest trudne, jeśli dane klasy już istnieją. Zaimplementowanie operacji `Clone` może okazać się skomplikowane także wtedy, jeżeli w tych klasach używane są obiekty nieobsługujące kopiowania lub mające referencje cykliczne.

¹ Takie aplikacje ilustrują zastosowanie wzorców Kompozyt (s. 163) i Dekorator (s. 175).

IMPLEMENTACJA

Wzorzec Prototyp jest szczególnie przydatny w językach ze statyczną kontrolą typów, np. w C++, gdzie klasy nie są obiektami i w czasie wykonywania programu dostępnych jest niewiele informacji o typie (lub w ogóle ich brak). Wzorzec ten ma mniejsze znaczenie w takich językach, jak Smalltalk lub Objective C, ponieważ udostępniają one do tworzenia egzemplarzy każdej klasy strukturę o możliwościach prototypu (chodzi tu o obiekt klasy Omawiany wzorzec wbudowano w języki oparte na prototypach, takie jak Self [US87], w których tworzenie obiektów zawsze odbywa się przez klonowanie prototypu).

W czasie implementowania prototypów należy rozważyć następujące kwestie:

1. *Korzystanie z menedżera prototypów.* Jeśli liczba prototypów w systemie nie jest stała (ponieważ można je dynamicznie tworzyć i usuwać), należy przechowywać rejestr dostępnych prototypów. Klienci nie będą wtedy samodzielnie zarządzać prototypami, ale mogą zapisywać je w archiwum i stamtąd pobierać. Klient przed sklonowaniem prototypu musi wtedy zaządać udostępnienia go przez rejestr. To archiwum nazywamy **menedżerem prototypów**.

Menedżer prototypów to struktura asocjacyjna zwracająca prototyp pasujący do podanego klucza. Udostępnia operacje do rejestrowania prototypów za pomocą klucza i wyrejestrowywania ich. Klienci mogą w czasie wykonywania programu modyfikować rejestr, a nawet przeglądać go. Umożliwia to rozszerzanie i sprawdzanie zawartości systemu bez konieczności pisania kodu.

2. *Implementowanie operacji Clone.* Najtrudniejszym aspektem stosowania wzorca Prototyp jest właściwe zaimplementowanie operacji *Clone*. Jest to szczególnie skomplikowane, jeśli struktury obiektu obejmują referencje cykliczne.

Większość języków udostępnia pewne mechanizmy do klonowania obiektów. Na przykład język Smalltalk obejmuje implementację metody *copy* dziedziczoną we wszystkich podklasach klasy *Object*. Język C++ udostępnia konstruktor kopiący. Jednak mechanizmy te nie rozwiązują problemu płytkego i głębokiego kopiowania [GR83] związanego z tym, że klonowanie obiektu spowoduje skopiowanie zmiennych egzemplarza, czy klon i oryginał będą jedynie współużytkować te zmienne.

Płytna kopia jest prosta i zwykle wystarczająca. W języku Smalltalk jest ona tworzona do myślnie. Domyślny konstruktor kopiący w języku C++ przeprowadza kopowanie po szczegółowych składowych, co oznacza, że wskaźniki będą współużytkowane przez kopię i oryginał. Jednak klonowanie prototypów o złożonych strukturach zwykle wymaga utworzenia głębokiej kopii, ponieważ klon i oryginał muszą być niezależne od siebie. Dlatego trzeba zagwarantować, że komponenty klonu to kopie komponentów prototypu. Klonowanie wymaga zadecydowania, co (jeśli cokolwiek) będzie współużytkowane.

Jeżeli obiekty w systemie udostępniają operacje *Save* i *Load*, można je wykorzystać do utworzenia domyślnej implementacji operacji *Clone*, która po prostu zapisuje obiekt i natychmiast ponownie go wczytuje. Operacja *Save* zapisuje obiekt do bufora w pamięci, a operacja *Load* tworzy duplikat przez odtworzenie danego obiektu na podstawie danych z bufora.

3. *Inicjowanie klonów.* Choć niektóre klienty bez problemów korzystają z klonu w jego pierwotnej postaci, w innych pożądane jest zainicjowanie części lub całości wewnętrznego stanu klonu wybranymi wartościami. Zwykle wartości tych nie można przekazać do operacji `Clone`, ponieważ ich liczba jest inna w zależności od klasy prototypu. Niektóre prototypy wymagają wielu parametrów inicjujących, a inne wcale ich nie potrzebują. Przekazywanie parametrów do operacji `Clone` narusza jednolity interfejs klonowania.

Może się zdarzyć, że w klasach prototypów zdefiniowane są operacje do ustawiania lub modyfikowania kluczowych składników stanu. Jeśli tak jest, klienty mogą wywołać te operacje bezpośrednio po klonowaniu. W przeciwnym razie konieczne może być wprowadzenie operacji `Initialize` (zobacz punkt „Przykładowy kod”). Powinna ona przyjmować parametr inicjujący i na jego podstawie ustawiać wewnętrzny stan klonu. Należy zachować szczególną ostrożność przy korzystaniu z operacji `Clone` przeprowadzających głębokie kopowanie. Utworzone przez nie kopie czasem trzeba usunąć (albo bezpośrednio, albo w operacji `Initialize`) przed ich ponownym zainicjowaniem.

PRZYKŁADOWY KOD

Zdefiniujemy tu podklasę `MazePrototypeFactory` klasy `MazeFactory` (s. 106). Do inicjowania klasy `MazePrototypeFactory` posłużą prototypy obiektów, które ma ona utworzyć, dlatego nie trzeba będzie tworzyć jej podklasy tylko po to, aby zmienić generowane w niej ściany lub pomieszczenia.

W klasie `MazePrototypeFactory` wzbogaciliśmy interfejs klasy `MazeFactory` o konstruktor przyjmujący argumenty w postaci prototypów:

```
class MazePrototypeFactory : public MazeFactory {
public:
    MazePrototypeFactory(Maze*, Wall*, Room*, Door*);

    virtual Maze* MakeMaze() const;
    virtual Room* MakeRoom(int) const;
    virtual Wall* MakeWall() const;
    virtual Door* MakeDoor(Room*, Room*) const;

private:
    Maze* _prototypeMaze;
    Room* _prototypeRoom;
    Wall* _prototypeWall;
    Door* _prototypeDoor;
};
```

Nowy konstruktor jedynie inicjuje prototypy:

```
MazePrototypeFactory::MazePrototypeFactory (
    Maze* m, Wall* w, Room* r, Door* d
) {
    _prototypeMaze = m;
    _prototypeWall = w;
    _prototypeRoom = r;
    _prototypeDoor = d;
}
```

Funkcje składowe służące do tworzenia ścian, pomieszczeń i drzwi wyglądają podobnie. Każda z nich klonuje prototyp, a następnie go inicjuje. Oto definicje funkcji MakeWall i MakeDoor:

```
Wall* MazePrototypeFactory::MakeWall () const {
    return _prototypeWall->Clone();
}

Door* MazePrototypeFactory::MakeDoor (Room* r1, Room *r2) const {
    Door* door = _prototypeDoor->Clone();
    door->Initialize(r1, r2);
    return door;
}
```

Klasę MazePrototypeFactory możemy wykorzystać do tworzenia prototypowego (domyślnego) labiryntu przez zainicjowanie jej za pomocą prototypów podstawowych komponentów labiryntu.

```
MazeGame game;
MazePrototypeFactory simpleMazeFactory(
    new Maze, new Wall, new Room, new Door
);

Maze* maze = game.CreateMaze(simpleMazeFactory);
```

Aby zmienić rodzaj labiryntu, należy zainicjować klasę MazePrototypeFactory za pomocą innego zestawu prototypów. Poniższe wywołanie tworzy labirynt z obiektami BombedDoor i RoomWithABomb.

```
MazePrototypeFactory bombedMazeFactory(
    new Maze, new BombedWall,
    new RoomWithABomb, new Door
);
```

Obiekt, który można zastosować jako prototyp (na przykład egzemplarz klasy Wall), musi obsługiwać operację Clone. Musi też posiadać konstruktor kopiący potrzebny do klonowania. Czasem potrzebna jest ponadto odrębna operacja do ponownego inicjowania wewnętrznego stanu. Dodajmy do klasy Door operację Initialize, aby umożliwić klientom inicjowanie po mieszczeń klonu.

Warto porównać poniższą definicję klasy Door do kodu ze strony 83.

```
class Door : public MapSite {
public:
    Door();
    Door(const Door&);

    virtual void Initialize(Room*, Room*);
    virtual Door* Clone() const;
    virtual void Enter();
    Room* OtherSideFrom(Room*);

private:
    Room* _room1;
    Room* _room2;
};
```

```

Door::Door (const Door& other) {
    _room1 = other._room1;
    _room2 = other._room2;
}

void Door::Initialize (Room* r1, Room* r2) {
    _room1 = r1;
    _room2 = r2;
}

Door* Door::Clone () const {
    return new Door(*this);
}

```

W podklasie BombedWall trzeba przesłonić operację Clone i zaimplementować odpowiedni konstruktor kopiący.

```

class BombedWall : public Wall {
public:
    BombedWall();
    BombedWall(const BombedWall&);

    virtual Wall* Clone() const;
    bool HasBomb();

private:
    bool _bomb;
};

BombedWall::BombedWall (const BombedWall& other) : Wall(other) {
    _bomb = other._bomb;
}

Wall* BombedWall::Clone () const {
    return new BombedWall(*this);
}

```

Choć operacja BombedWall::Clone zwraca wskaźnik Wall*, w implementacji tej operacji zwracany jest wskaźnik do nowego egzemplarza podklasty — BombedWall*. Zdefiniowaliśmy operację Clone w klasie bazowej w ten sposób, aby zagwarantować, że klienci klonujące prototyp nie będą potrzebowali informacji o podklasach konkretnych. W klientach nigdy nie powinno być konieczne rzutowanie wartości zwróconej przez operację Clone w dół (do pożądanego typu).

W języku Smalltalk do skłonowania dowolnego obiektu z rodziny MapSite można powtórnie wykorzystać standardową metodę copy odziedziczoną po klasie Object. Klasy MazeFactory można użyć do wytwarzania potrzebnych prototypów. Na przykład aby utworzyć pomieszczenie, należy podać nazwę #room. Klasa MazeFactory obejmuje słownik z odwzorowaniami nazw na prototypy. Metoda make: w tej klasie wygląda tak:

```

make: partName
^ (partCatalog at: partName) copy

```

Po utworzeniu odpowiednich metod do inicjowania klasy `MazeFactory` za pomocą prototypów można zbudować prosty labirynt przy użyciu poniższego kodu:

```
CreateMaze
on: (MazeFactory new
      with: Door new named: #door;
      with: Wall new named: #wall;
      with: Room new named: #room;
      yourself)
```

W tym rozwiążaniu definicja metody statycznej `on:` dla metody `CreateMaze` może wyglądać tak:

```
on: aFactory
| room1 room2 |
room1 := (aFactory make: #room) location: 1@1.
room2 := (aFactory make: #room) location: 2@1.
door := (aFactory make: #door) from: room1 to: room2.

room1
atSide: #north put: (aFactory make: #wall);
atSide: #east put: door;
atSide: #south put: (aFactory make: #wall);
atSide: #west put: (aFactory make: #wall).
room1
atSide: #north put: (aFactory make: #wall);
atSide: #east put: (aFactory make: #wall);
atSide: #south put: (aFactory make: #wall);
atSide: #west put: door.
^ Maze new
addRoom: room1;
addRoom: room2;
yourself
```

ZNANE ZASTOSOWANIA

Prawdopodobnie pierwszym przykładem zastosowania wzorca Prototyp był system Sketchpad Ivana Sutherlanda [Sut63]. Pierwszym powszechnie znanym przypadkiem wykorzystania tego wzorca w języku obiektowym był system ThingLab. Umożliwił on użytkownikom tworzenie kompozytów, a następnie korzystanie z nich jak z prototypów przez zainstalowanie ich w bibliotece obiektów wielokrotnego użytku [Bor81]. Goldberg i Robson wspomnieli o prototypach jako o wzorcu [GR83], jednak dużo pełniejszy opis przedstawił Coplien [Cop92]. Wyjął on idiomy języka C++ związane ze wzorcem Prototyp oraz przedstawił wiele przykładów i wersji tego wzorca.

Etgdb to oparty na platformie ET++ fronton debugerów zapewniający interfejs graficzny dla różnych debugerów działających z poziomu wiersza poleceń. Dla każdego debugera istnieje odpowiednia podkلاśa `DebuggerAdaptor`. Na przykład klasa `GdbAdaptor` przystosowuje narzędzie etgdb do składni debugera GNU gdb, natomiast klasa `SunDbxAdaptor` robi to samo na potrzeby debugera dgx firmy Sun. W etgdb nie ma zapisanego na stałe zestawu klas `Debugger->Adaptor`. Zamiast tego narzędzie wczytuje nazwę adaptera ze zmiennej środowiskowej,

wyszukuje w tabeli globalnej prototyp o określonej nazwie, a następnie klonuje go. Do etgdb można dodawać obsługę nowych debuggerów przez powiązanie ich z klasą DebuggerAdaptor specyficzną dla danego debugera.

„Biblioteka technik interakcji” w aplikacji Mode Composer obejmuje prototypy obiektów obsługujących różne metody interakcji [Sha90]. Każdą technikę utworzoną przez ten program można wykorzystać jako prototyp przez umieszczenie jej w bibliotece. Wzorzec Prototyp umożliwia obsługę w aplikacji Mode Composer nieograniczonego zbioru technik interakcji.

Opisany wcześniej przykład dotyczący edytora utworów muzycznych oparliśmy na platformie do edycji graficznej Unidraw [VL90].

POWIĄZANE WZORCE

Prototyp i Fabryka abstrakcyjna (s. 101) to pod niektórymi względami „konkurencyjne” wzorce (zagadnienie to omawiamy w końcowej części rozdziału). Można ich jednak używać wspólnie. Fabryka abstrakcyjna może przechowywać zestaw prototypów stosowanych do klonowania i zwracania obiektów-produktów.

Zastosowanie wzorca Prototyp może być korzystne także w tych projektach, w których w wielu miejscach wykorzystano wzorce Kompozyt (s. 170) i Dekorator (s. 152).

SINGLETON (SINGLETON)

obiektowy, konstrukcyjny

PRZEZNACZENIE

Gwarantuje, że klasa będzie miała tylko jeden egzemplarz, i zapewnia globalny dostęp do niego

UZASADNIENIE

W przypadku niektórych klas ważne jest, aby miały one tylko jeden egzemplarz. Choć w systemie może działać wiele drukarek, powinien znajdować się w nim tylko jeden program buforujący drukowania. Potrzebny jest tylko jeden system plików i menedżer okien, filtr cyfrowy powinien mieć tylko jeden konwerter analogowy-cyfrowy, a system rozliczeniowy powinien być przeznaczony do obsługi tylko jednej firmy.

Jak można zagwarantować, że klasa będzie miała tylko jeden łatwo dostępny egzemplarz Zmienna globalna zapewnia dostęp do obiektu, jednak pozostawia możliwość utworzenia wielu obiektów.

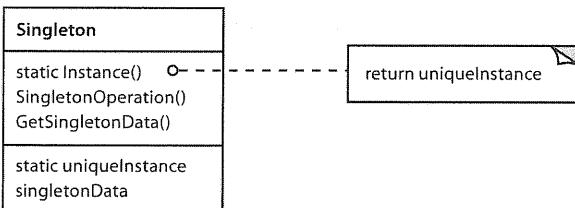
Lepsze rozwiązańe polega na przydzieleniu klasie zadania śledzenia swojego jedynego egzemplarza. Klasa może zagwarantować (przez przechwytywanie żądań utworzenia nowych obiektów), że nie powstanie żaden inny jej egzemplarz, a także może umożliwiać dostęp do jedynego egzemplarza. Tak właśnie działa wzorzec Singleton.

WARUNKI STOSOWANIA

Wzorca Singleton należy używać w następujących warunkach:

- ▶ Jeśli musi istnieć dokładnie jeden egzemplarz klasy dostępny klientom w znany miejscu.
- ▶ Kiedy potrzebna jest możliwość rozszerzania jedynego egzemplarza przez tworzenie pod klas, a klienci powinny móc korzystać ze wzbogaconego egzemplarza bez konieczności wprowadzania zmian w ich kodzie.

STRUKTURA



ELEMENTY

► **Singleton:**

- definiuje operację `Instance` umożliwiającą klientom dostęp do niepowtarzalnego egzemplarza klasy; `Instance` to operacja statyczna (czyli metoda statyczna w języku Smalltalk lub statyczna funkcja składowa w języku C++);
- może odpowiadać za tworzenie własnego niepowtarzalnego egzemplarza.

WSPÓŁDZIAŁANIE

- Klienci mogą uzyskać dostęp do egzemplarza klasy `Singleton` wyłącznie poprzez operację `Instance` z tej klasy.

KONSEKWENCJE

Wzorzec Singleton zapewnia kilka korzyści:

1. *Zapewnia kontrolę dostępu do jedynego egzemplarza.* Ponieważ klasa `Singleton` kapsułkuje swój jedyny egzemplarz, można w niej ścisłe kontrolować, w jaki sposób i kiedy klienci mogą uzyskać do niego dostęp.
2. *Pozwala zmniejszyć przestrzeń nazw.* Wzorzec `Singleton` jest ulepszeniem w porównaniu do zmiennych globalnych. Pozwala uniknąć zaśmiecania przestrzeni nazw zmiennymi globalnymi przechowującymi jedyne egzemplarze.
3. *Umożliwia dopracowywanie operacji i reprezentacji.* Można tworzyć podklasy klasy `Singleton`, a ponadto łatwo jest skonfigurować aplikację za pomocą egzemplarza takiej rozszerzonej klasy. Potrzebną do tego klasę można podać w czasie wykonywania programu.
4. *Umożliwia określenie dowolnego limitu liczby egzemplarzy.* Omawiany wzorzec umożliwia łatwą zmianę podejścia i zezwolenie na tworzenie więcej niż jednego egzemplarza klasy `Singleton`. Ponadto to samo rozwiązanie można zastosować do kontrolowania liczby egzemplarzy używanych w aplikacji. Trzeba wtedy zmodyfikować jedynie operację, która zapewnia dostęp do egzemplarza klasy `Singleton`.
5. *Jest bardziej elastyczny od operacji statycznych.* Inny sposób na opakowanie funkcji singletonu polega na wykorzystaniu operacji statycznych (czyli statycznych funkcji składowych w języku C++ lub metod statycznych w języku Smalltalk). Jednak obie te techniki utrudniają zmianę projektu tak, aby umożliwić tworzenie więcej niż jednego egzemplarza klasy. Ponadto statyczne funkcje składowe w języku C++ nigdy nie są wirtualne, dlatego w podklasach nie można prześledzić ich w sposób polimorficzny.

IMPLEMENTACJA

Oto kwestie związane z implementacją, które należy rozważyć przy stosowaniu wzorca `Singleton`:

1. *Zapewnianie niepowtarzalności egzemplarza.* We wzorcu `Singleton` jedyny egzemplarz jest zwykłym egzemplarzem klasy, jednak jest ona napisana tak, aby można utworzyć tylko ten egzemplarz. Standardowe rozwiązanie polega na ukryciu operacji tworzącej egzemplarz w operacji statycznej (czyli statycznej funkcji składowej lub metodzie statycznej), która

gwarantuje, że może powstać tylko jeden egzemplarz danej klasy. Ta operacja ma dostęp do zmiennej przechowującej ów egzemplarz, a zanim zwróci jej wartość, upewnia się, że zmienna została zainicjowania za pomocą niepowtarzalnego egzemplarza. To podejście gwarantuje, że singleton zostanie utworzony i zainicjowany przed jego pierwszym użyciem.

W języku C++ opisaną operację statyczną można zdefiniować jako statyczną funkcję składową Instance klasy Singleton. W klasie tej należy też zdefiniować statyczną zmienną składową _instance zawierającą wskaźnik do niepowtarzalnego egzemplarza.

Deklaracja klasy Singleton wygląda tak:

```
class Singleton {
public:
    static Singleton* Instance();
protected:
    Singleton();
private:
    static Singleton* _instance;
};
```

A oto jej implementacja:

```
Singleton* Singleton::_instance = 0;

Singleton* Singleton::Instance () {
    if (_instance == 0) {
        _instance = new Singleton;
    }
    return _instance;
}
```

Klienci mogą uzyskać dostęp do singletonu wyłącznie poprzez funkcję składową Instance. Zmienna _instance jest inicjowana wartością 0, a statyczna funkcja składowa Instance zwraca wartość tej zmiennej (przy czym jeśli wartość ta wynosi 0, funkcja inicjuje zmienną za pomocą niepowtarzalnego egzemplarza). W funkcji Instance zastosowano leniwe inicjowanie. Wartość, którą zwraca ta funkcja, nie jest tworzona ani zapisywana do momentu pierwszego wywołania funkcji.

Warto zauważyć, że konstruktor jest chroniony. Jeśli klient spróbuje bezpośrednio utworzyć egzemplarz klasy Singleton, w czasie komplikacji pojawi się komunikat o błędzie. Gwarantuje to, że powstanie tylko jeden egzemplarz tej klasy.

Ponadto z uwagi na to, że zmienna _instance to wskaźnik do obiektu Singleton, funkcja składowa Instance może przypisać do tej zmiennej wskaźnik do obiektu podklasy klasy Singleton. Przykład takiego rozwiązania przedstawiamy w punkcie „Przykładowy kod”.

Warto zwrócić uwagę na jeszcze jeden aspekt przedstawionej implementacji w języku C++. Nie wystarczy zdefiniować singletonu jako obiektu globalnego lub statycznego i zdać się na inicjowanie automatyczne. Wynika to z trzech powodów:

- Nie można zagwarantować, że zadeklarowany zostanie tylko jeden egzemplarz obiektu statycznego.

- b) Możemy nie mieć informacji wystarczających do utworzenia egzemplarza każdego singletonu w czasie statycznego inicjowania. Niezbędne mogą być wartości obliczane w późniejszej fazie działania programu.
- c) W języku C++ kolejność wywoływania konstruktorów obiektów globalnych z różnych jednostek translacji nie jest określona [ES90]. Oznacza to, że między singletonami nie mogą występować zależności. Jeśli warunek ten nie będzie spełniony, z pewnością pojawią się błędy.

Dodatkową (choć niewielką) wadą podejścia opartego na obiektach globalnych lub statycznych jest to, że trzeba utworzyć wszystkie singletony niezależnie od tego, czy są używane czy nie. Zastosowanie statycznej funkcji składowej pozwala uniknąć wszystkich tych problemów.

W języku Smalltalk funkcja zwracająca niepowtarzalny egzemplarz jest implementowana jako metoda statyczna w klasie `Singleton`. Aby zagwarantować, że powstanie tylko jeden jej egzemplarz, należy przesłonić operację `new`. Utworzona w ten sposób klasa `Singleton` może obejmować dwie poniższe metody statyczne (`SoleInstance` to zmienna statyczna używana wyłącznie w tym miejscu):

```
new
    self error: 'Nie można utworzyć nowego obiektu'

default
    SoleInstance isNil ifTrue: [SoleInstance := super new].
    ^ SoleInstance
```

2. *Tworzenie podklas klasy Singleton*. Największy problem jest związany nie tyle z definiowaniem podklas, ile z instalowaniem niepowtarzalnych egzemplarzy, aby klienci mogły z nich korzystać. Rozwiążanie sprowadza się do tego, że zmienną wskazującą na egzemplarz singletonu trzeba zainicjować za pomocą egzemplarza odpowiedniej podklasy. Najprostsza technika, która to umożliwia, polega na określeniu potrzebnego singletonu w operacji `Instance` klasy `Singleton`. W przykładzie w punkcie „Przykładowy kod” pokazaliśmy, jak zastosować tę technikę za pomocą zmiennych środowiskowych.

Inny sposób wybierania podklasy klasy `Singleton` polega na umieszczeniu implementacji operacji `Instance` poza klasą nadzczną (na przykład `MazeFactory`) — w podklasie. Umożliwia to programiście języka C++ ustalenie klasy singletonu w czasie konsolidacji (na przykład przez dołączenie pliku wynikowego zawierającego inną implementację), jednak ukrywa tę klasę przed klientami korzystającymi z singletonu.

W podejściu opartym na dołączaniu klasy singletonu jest ustalana w czasie konsolidacji, co utrudnia wskazanie takiej klasy w czasie wykonywania programu. Wykorzystanie instrukcji warunkowych do określania podklasy to elastyczniejsze rozwiązanie, jednak powoduje zapisanie na stałe zestawu dostępnych klas singletonów. Żadne z tych podejść nie zapewnia wystarczającej elastyczności w każdych warunkach.

Elastyczniejsze podejście polega na zastosowaniu **rejestru singletonów**. Zamiast definiować zestaw dostępnych klas singletonów w operacji `Instance`, można nakazać takim klasom rejestrowanie egzemplarzy singletonów w znany rejestrze za pomocą nazw.

Rejestr łączy nazwy w postaci łańcuchów znaków z singletonami. Kiedy operacja `Instance` potrzebuje singletonu, korzysta z rejestru, żądając singletonu przez podanie jego nazwy. Rejestr wyszukuje wtedy odpowiedni singleton (jeśli taki istnieje) i zwraca go. W tym rozwiązaniu nie trzeba zapisywać w operacji `Instance` wszystkich dostępnych klas lub egzemplarzy singletonów. Wystarczy we wszystkich klasach singletonów umieścić wspólny interfejs z operacjami do obsługi rejestrów:

```
class Singleton {
public:
    static void Register(const char* name, Singleton* );
    static Singleton* Instance();
protected:
    static Singleton* Lookup(const char* name);
private:
    static Singleton* _instance;
    static List<NameSingletonPair>* _registry;
};
```

Operacja `Register` rejestruje egzemplarz singletonu pod podaną nazwą. Aby nie komplikować rejestrów, umieścimy w nim listę obiektów `NameSingletonPair`. Każdy taki obiekt odwzorowuje nazwę na singleton. Operacja `Lookup` wyszukuje singleton na podstawie jego nazwy. Zakładamy, że nazwę potrzebnego singletonu określa zmieniona środowiskowa.

```
Singleton* Singleton::Instance () {
    if (_instance == 0) {
        const char* singletonName = getenv("SINGLETON");
        // Użytkownik lub środowisko podaje tę wartość w czasie uruchamiania programu.

        _instance = Lookup(singletonName);
        // Operacja Lookup zwraca 0, jeśli podany singleton nie istnieje.
    }
    return _instance;
}
```

W którym miejscu klasy singletonów się rejestrują? Jedną z możliwości jest użycie do tego konstruktora. Na przykład w podklasie `MySingleton` można wykonać następującą operację:

```
MySingleton::MySingleton() {
    // ...
    Singleton::Register("MySingleton", this);
}
```

Konstruktor oczywiście zostanie wywołany dopiero w momencie tworzenia egzemplarza danej klasy. W ten sposób wracamy do problemu, który wzorzec `Singleton` miał rozwiązać. W języku C++ można sobie z nim poradzić przez zdefiniowanie statycznego egzemplarza klasy `MySingleton`. Możemy na przykład dodać poniższy kod:

```
static MySingleton theSingleton;
```

Należy go umieścić w pliku z implementacją klasy `MySingleton`.

W tym rozwiązaniu klasa `Singleton` nie odpowiada za tworzenie singletonu. Teraz jej głównym zadaniem jest udostępnianie w systemie wybranego obiektu singletonu. Podejście oparte na statycznym obiekcie nadal ma pewną wadę — trzeba utworzyć egzemplarze wszystkich możliwych podklas klasy `Singleton`, ponieważ w przeciwnym razie singletony nie zostaną zarejestrowane.

PRZYKŁADOWY KOD

Załóżmy, że zdefiniowaliśmy klasę do tworzenia labiryntów, `MazeFactory`, w sposób opisany na stronie 92. Klasa ta określa interfejs do tworzenia różnych części labiryntu. W podklasach można umieścić nowe definicje operacji, aby zwracały egzemplarze wyspecjalizowanych klas produktów (na przykład obiekty `BombedWall` zamiast `Wall`).

Ważne jest to, że w aplikacji Labirynt potrzebny jest tylko jeden egzemplarz klasy `MazeFactory`. Powinien być on dostępny w kodzie tworzącym poszczególne części labiryntu. W uzyskaniu tego efektu pomoże wzorzec Singleton. Przez utworzenie klasy `MazeFactory` jako singletonu można bez uciekania się do korzystania ze zmiennych globalnych sprawić, że obiekt reprezentujący labirynt będzie globalnie dostępny.

Dla uproszczenia przyjmijmy, że nie będziemy tworzyć podklas klasy `MazeFactory` (inną możliwość rozważamy za chwilę). W języku C++ można zdefiniować ją jako klasę typu singleton przez dodanie operacji statycznej `Instance` i zmiennej składowej `_instance` do przechowywania jedynego egzemplarza. Ponadto trzeba zadeklarować konstruktor jako chroniony, aby zapobiec przypadkowemu utworzeniu większej liczby egzemplarzy.

```
class MazeFactory {
public:
    static MazeFactory* Instance();

    // Tu należy umieścić istniejący interfejs.

protected:
    MazeFactory();

private:
    static MazeFactory* _instance;
};
```

A oto implementacja tej klasy:

```
MazeFactory* MazeFactory::_instance = 0;

MazeFactory* MazeFactory::Instance () {
    if (_instance == 0 ) {
        _instance = new MazeFactory();
    }
    return _instance;
}
```

Teraz rozważmy inny przypadek. Załóżmy, że klasa `MazeFactory` ma podklasy, a aplikacja musi wybrać jedną z nich. Rodzaj labiryntu wskażemy za pomocą zmiennej środowiskowej i dodamy kod, który utworzy egzemplarz odpowiedniej podklasy klasy `MazeFactory` na podstawie wartości tej zmiennej. Dobrym miejscem na ten kod jest operacja `Instance`, ponieważ tworzy ona egzemplarz klasy `MazeFactory`:

```
MazeFactory* MazeFactory::Instance () {
    if (_instance == 0 ) {
        const char* mazeStyle = getenv("MAZESTYLE");

        if (strcmp(mazeStyle, "bombed") == 0) {
```

```

        _instance = new BombedMazeFactory;

    } else if (strcmp(mazeStyle, "enchanted") == 0) {
        _instance = new EnchantedMazeFactory;

    // Inne dostępne podklasy.

    } else { // Klasa domyślna.
        _instance = new MazeFactory;
    }
}

return _instance;
}

```

Warto zauważyć, że po zdefiniowaniu każdej nowej podklasy klasy `MazeFactory` operację `Instance` trzeba zmodyfikować. W tej aplikacji nie sprawia to problemu, jednak może okazać się to trudne w przypadku fabryk abstrakcyjnych zdefiniowanych w platformie.

Możliwym rozwiązaniem jest zastosowanie podejścia opartego na rejestrze, które opisaliśmy w punkcie Implementacja. Dynamiczne dodawanie może być przydatne także w tym przypadku, ponieważ sprawia, że aplikacja nie musi wczytywać nieużywanych podklas.

ZNANE ZASTOSOWANIA

Przykładem zastosowania wzorca Singleton w języku Smalltalk-80 [Par90] jest zbiór zmian w kodzie — `ChangeSet current`. Bardziej wyrafinowanym przykładem jest relacja między klasami i ich **metaklasami**. Metaklasa to klasa reprezentująca klasę. Każda metaklasa ma jeden egzemplarz. Metaklasy nie mają nazw (są określane pośrednio poprzez ich jedynie egzemplarze), jednak śledzą swój jedyny egzemplarz i w standardowych warunkach nie tworzą innych.

W pakiecie narzędziowym InterViews [LCI-92] (służy on do tworzenia interfejsów użytkownika) wzorzec Singleton wykorzystano do zapewnienia dostępu do niepowtarzalnych egzemplarzy klas `Session` i `WidgetKit` (a także innych). W klasie `Session` zdefiniowano główną pętlę obsługi zdarzeń aplikacji. Klasa ta służy też do przechowywania bazy danych z preferencjami stylistycznymi użytkownika i zarządzania połączeniami z wyświetlaczami. `WidgetKit` to Fabryka abstrakcyjna (s. 101) do definiowania wyglądu i działania widgetów interfejsu użytkownika. Operacja `WidgetKit::instance()` określa specyficzną podkласę klasy `WidgetKit`, której egzemplarz jest tworzony na podstawie zmiennej środowiskowej zdefiniowanej w klasie `Session`. Podobna operacja w klasie `Session` określa, czy obsługiwane są wyświetlacze czarno-białe czy kolorowe, i zgodnie z tym konfiguruje jedyny egzemplarz klasy `Session`.

POWIĄZANE WZORCE

Za pomocą wzorca Singleton można zaimplementować wiele innych wzorców. Zobacz wzorce Fabryka abstrakcyjna (s. 101), Budowniczy (s. 92) i Prototyp (s. 120).

OMÓWIENIE WZORCÓW KONSTRUKCYJNYCH

Istnieją dwa standardowe sposoby na sparametryzowanie systemu za pomocą klas tworzących w nim obiekty. Jedna możliwość to utworzenie podklasy klasy tworzącej obiekty. W ten sposób działa wzorzec Metoda wytwórcza (s. 110). Podstawową wadą tego podejścia jest konieczność tworzenia nowej podklasy tylko po to, aby zmienić klasę produktu. Czasem z jednej modyfikacji tego rodzaju wynika inna. Na przykład jeśli obiekt tworzący produkt sam jest generowany przez metodę fabryczną, trzeba przesłonić także tę metodę.

Inny sposób parametryzowania systemów jest w większym stopniu oparty na składaniu obiektów. Należy zdefiniować obiekt odpowiedzialny za określanie klasy obiektów-produktów i użyć go jako parametru systemu. Jest to kluczowy aspekt działania wzorców Fabryka abstrakcyjna (s. 101), Budowniczy (s. 92) i Prototyp (s. 120). Wszystkie trzy dotyczą tworzenia nowego „obiektu wytwórczego”, którego zadaniem jest tworzenie obiektów-produktów. We wzorcu Fabryka abstrakcyjna obiekt wytwórczy generuje obiekty kilku klas. We wzorcu Budowniczy obiekt wytwórczy stopniowo buduje złożony produkt za pomocą odpowiednio złożonego protokołu. We wzorcu Prototyp obiekt wytwórczy generuje produkt przez kopowanie obiektu prototypowego. W tym przypadku obiekt wytwórczy i prototyp to ten sam obiekt, ponieważ prototyp odpowiada za zwrócenie produktu.

Rozważmy platformę do tworzenia edytorów graficznych omówioną w opisie wzorca Prototyp. Jest kilka sposobów na sparametryzowanie klasy `GraphicTool` za pomocą klasy produktu:

- ▶ Przy stosowaniu wzorca Metoda wytwórcza dla każdej podklasy klasy `Graphic` zostanie utworzona podklaśa klasy `GraphicTool`. W klasie `GraphicTool` należy umieścić operację `NewGraphic` i przeddefiniować ją w każdej podklasie klasy `GraphicTool`.
- ▶ Przy stosowaniu wzorca Fabryka abstrakcyjna powstanie hierarchia klas z rodzinie `Graphic` → `Factory` (po jednej dla każdej podklasy klasy `Graphic`). Każda fabryka będzie tworzyć tylko jeden produkt. Klasa `CircleFactory` będzie generować obiekty `Circle`, klasa `LineFactory` będzie tworzyć obiekty `Line` itd. Klasę `GraphicTool` należy sparametryzować za pomocą fabryki do tworzenia obiektów `Graphic` odpowiedniego rodzaju.
- ▶ Przy stosowaniu wzorca Prototyp każda podklaśa klasy `Graphic` będzie obejmować implementację operacji `Clone`, a klasa `GraphicTool` zostanie sparametryzowana prototypem klasy `Graphic`, której egzemplarz ma tworzyć.

To, który wzorzec jest najbardziej przydatny, zależy od wielu czynników. W platformie do tworzenia edytorów graficznych początkowo najprościej jest użyć wzorca Metoda wytwórcza. Łatwo jest zdefiniować nową podklaśa klasy `GraphicTool`, a egzemplarze tej klasy są tworzone tylko po zdefiniowaniu palety narzędzi. Główną wadą tego rozwiązania jest duża liczba podklaśs klasy `GraphicTool` i to, że ich zadania są bardzo ograniczone.

Zastosowanie wzorca Fabryka abstrakcyjna nie jest istotnym usprawnieniem, ponieważ wymaga utworzenia równie dużej hierarchii klasy `GraphicsFactory`. Wzorzec ten warto zastosować zamiast Metody wytwórczej tylko wtedy, jeśli hierarchia klasy `GraphicsFactory` już istnieje, ponieważ albo została automatycznie udostępniona przez kompilator (jak ma to miejsce w językach Smalltalk i Objective C), albo jest potrzebna w innej części systemu.

Ogólnie prawdopodobnie najlepszym wzorcem na potrzeby platformy do tworzenia edytörów graficznych jest Prototyp, ponieważ wymaga jedynie zaimplementowania operacji *Clone* w każdej klasie *Graphics*. Pozwala to zmniejszyć liczbę klas, a operację *Clone* można wykorzystać także do innych celów oprócz tworzenia egzemplarzy (na przykład do obsługi operacji *Powiel* w menu).

Metoda wytwórcza zwiększa możliwość dostosowania projektu do własnych potrzeb, a przy tym sprawia, że jest on w niewielkim tylko stopniu bardziej skomplikowany. Inne wzorce projektowe wymagają tworzenia nowych klas, natomiast przy stosowaniu Metody wytwórczej wystarczy dodać nową operację. Programiści często korzystają z tego wzorca jako standardowego sposobu tworzenia obiektów, jednak nie jest to konieczne, jeśli klasa, której egzemplarze powstają, nigdy się nie zmienia, lub jeżeli generowanie obiektów odbywa się w operacji łatwej do przesłonięcia w podklasach (na przykład w operacji odpowiedzialnej za inicjowanie).

Projekty oparte na wzorcach Fabryka abstrakcyjna, Prototyp lub Budowniczy są jeszcze elastyczniejsze od tych, w których zastosowano wzorzec Metoda wytwórcza, jednak dzieje się to kosztem wyższej złożoności. Często projektant początkowo korzysta ze wzorca Metoda wytwórcza, a następnie — kiedy odkryje, że potrzebna jest większa elastyczność — zmienia projekt przez zastosowanie innych wzorców konstrukcyjnych. Znajomość wielu wzorców projektowych zapewnia większy wybór w czasie analizowania różnych kryteriów projektowych.

Metoda wytwórcza jest szczególnie przydatna w sytuacjach, gdy konkretny typ obiektów powstaje w wyniku skomplikowanej logiki. W takich przypadkach konstrukcja klasowej hierarchii może być skomplikowana i nieintuitywna. Wystarczy, że programista chce zmienić sposób tworzenia obiektów, aby zmienić cały kod projektu. W takich sytuacjach Metoda wytwórcza jest doskonałym wyborem, ponieważ pozwala na zmianę sposobu tworzenia obiektów bez zmiany samego kodu projektu. Wystarczy zmienić jedynie operację *Clone*, aby zmienić sposób tworzenia obiektów. Metoda wytwórcza pozwala na łatwe dostosowanie projektu do różnych warunków, takich jak zmiana sposobu tworzenia obiektów, zmiana struktury danych, zmiana sposobu przechowywania obiektów itp.

Metoda wytwórcza jest szczególnie przydatna w sytuacjach, gdy konkretny typ obiektów powstaje w wyniku skomplikowanej logiki. W takich przypadkach konstrukcja klasowej hierarchii może być skomplikowana i nieintuitywna. Wystarczy, że programista chce zmienić sposób tworzenia obiektów, aby zmienić cały kod projektu. W takich sytuacjach Metoda wytwórcza jest doskonałym wyborem, ponieważ pozwala na zmianę sposobu tworzenia obiektów bez zmiany samego kodu projektu. Wystarczy zmienić jedynie operację *Clone*, aby zmienić sposób tworzenia obiektów. Metoda wytwórcza pozwala na łatwe dostosowanie projektu do różnych warunków, takich jak zmiana sposobu tworzenia obiektów, zmiana struktury danych, zmiana sposobu przechowywania obiektów itp.

Rozdział 4.

Wzorce strukturalne

Wzorce strukturalne dotyczą składania klas i obiektów w większe struktury. *Klasowe* wzorce strukturalne są oparte na wykorzystaniu dziedziczenia do składania interfejsów lub implementacji. W ramach prostego przykładu rozważmy, jak w procesie wielodziedziczenia klasy (dwie lub więcej) są łączane w jedną. W efekcie powstaje klasa z właściwościami jej klas nadrzędnych. Ten wzorzec jest szczególnie przydatny do zapewniania współdziałania niezależnie rozwijanych bibliotek klas. Inny przykład to klasowa wersja wzorca Adapter (s. 141). Ogólnie adapter dostosowuje jeden interfejs (adaptowany) do drugiego, zapewniając tym samym jednolitą abstrakcję różnych interfejsów. Adapter klasowy pozwala uzyskać ten efekt przez dziedziczenie prywatne po adaptowanej klasie i zbudowanie interfejsu adaptera w kategoriach elementów tej klasy.

Strukturalne wzorce *obiektowe* nie polegają na składaniu interfejsów lub implementacji, ale opisują sposoby składania obiektów w celu obsługi nowych funkcji. Zwiększoną elastyczność, jaka daje składanie obiektów, wynika z możliwości zmiany układu obiektu w czasie wykonywania programu, co jest niewykonalne przy statycznym składaniu klas.

Kompozyt (s. 170) to przykład strukturalnego wzorca obiektowego. Opisuje on, jak tworzyć hierarchie składające się z klas reprezentujących dwa rodzaje obiektów — proste i złożone. Obiekty złożone (kompozyty) umożliwiają składanie obiektów prostych i innych obiektów złożonych w dowolnie skomplikowanej strukturze. We wzorcu Pełnomocnik (s. 191) pełnomocnik pełni funkcję wygodnego substytutu lub zastępnika innego obiektu. Pełnomocnika można używać na wiele sposobów. Może on pełnić funkcję lokalnego przedstawiciela obiektu w zdalnej przestrzeni adresowej, reprezentować duży obiekt wczytywany na żądanie lub chronić dostęp do wrażliwego obiektu. Pełnomocnik zapewnia poziom pośredniości w procesie dostępu do specyficznych właściwości obiektów, dlatego może ograniczać, wzbogacać lub zmieniać te cechy.

Wzorzec Pyłek (s. 201) określa strukturę służącą do współużytkowania obiektów. Obiekty można współużytkować z przynajmniej dwóch przyczyn — ze względu na wydajność i spójność. Wzorzec Pyłek dotyczy przede wszystkim wydajności w obszarze wykorzystania pamięci. W aplikacjach, w których działa wiele obiektów, trzeba zwracać szczególną uwagę na koszt używania każdego z nich. Współużytkowanie obiektów zamiast ich powielania może przynieść znaczne oszczędności. Jednak obiekty można współużytkować tylko wtedy, jeśli ich stan

nie jest zależny od kontekstu. Obiekty we wzorcu Pyłek spełniają ten warunek. Wszystkie dodatkowe informacje potrzebne od wykonywania zadań są przekazywane do nich wtedy, kiedy są potrzebne. Z uwagi na brak stanu zależnego od kontekstu obiekty-pylki można swobodnie współużytkować.

Wzorzec Pyłek ilustruje generowanie wielu niewielkich obiektów, natomiast wzorzec Fasada (s. 161) pokazuje, jak przedstawić cały podsystem za pomocą jednego obiektu. Fasada to reprezentant zbioru obiektów. Wykonuje ona swoje zadania przez przekazywanie komunikatów do reprezentowanych obiektów. Wzorzec Most (s. 181) oddziela abstrakcję obiektu od jego implementacji, co umożliwia modyfikowanie ich niezależnie od siebie.

Wzorzec Dekorator (s. 152) określa, w jaki sposób dynamicznie dodawać zadania do obiektów. Dekorator to wzorzec strukturalny polegający na rekurencyjnym składaniu obiektów, co umożliwia dołączenie dowolnej liczby dodatkowych zadań. Na przykład za pomocą dekoratora zawierającego komponentu interfejsu użytkownika można dodawać ozdobniki, takie jak ramki lub cienie, lub funkcje, takie jak przewijanie i przybliżanie. W celu dodania dwóch ozdobników wystarczy zagnieździć jeden obiekt dekoratora w drugim (i tak dalej, aby dołączyć następne dodatki). Aby było to możliwe, każdy obiekt dekoratora musi być zgodny z interfejsem komponentu i przekazywać do niego komunikaty. Dekorator może wykonywać swoje zadanie (na przykład wyświetlać ramkę wokół komponentu) albo przed przekazaniem komunikatu, albo po jego wysłaniu.

Wiele wzorców strukturalnych jest ze sobą powiązanych. Relacje między nimi omawiamy w końcowej części rozdziału.

ADAPTER (ADAPTER)

klasowy i obiektowy, strukturalny

PRZEZNACZENIE

Przekształca interfejs klasy na inny, oczekiwany przez klienta. Adapter umożliwia współdziałanie klasom, które z uwagi na niezgodne interfejsy standardowo nie mogą współpracować ze sobą.

INNE NAZWY

Nakładka (ang. *wrapper*).

UZASADNIENIE

Czasem nie można powtórnie wykorzystać zaprojektowanej do wielokrotnego użytku klasy z pakietu narzędziowego, ponieważ jej interfejs nie jest zgodny ze specyficzny dla dziedziny interfejsem wymaganym przez aplikację.

Rozważmy przykład edytora graficznego umożliwiającego użytkownikom rysowanie i porządkowanie elementów graficznych (linii, wielokątów, tekstu itd.) na rysunkach oraz diagramach. Kluczową abstrakcją w edytorze jest obiekt graficzny. Umożliwia on modyfikowanie kształtu i potrafi się wyświetlić. Interfejs obiektów graficznych jest zdefiniowany w postaci klasy abstrakcyjnej o nazwie Shape. W edytorze dla obiektu graficznego każdego rodzaju zdefiniowana jest podkلاśa klasy Shape. W przypadku linii jest to klasa LineShape, w przypadku wielokątów — PolygonShape itd.

Klasy podstawowych figur geometrycznych, takie jak LineShape i PolygonShape, są stosunkowo łatwe w implementacji, ponieważ ich możliwości w zakresie rysowania oraz edycji są ograniczone. Jednak dużo trudniej jest zaimplementować podklaśę TextShape (obsługuje ona wyświetlanie i edycję tekstu), ponieważ nawet podstawowa edycja tekstu wymaga skomplikowanego aktualizowania stanu ekranu i zarządzania buforami. Jednocześnie gotowy pakiet narzędziowy do tworzenia interfejsów użytkownika może udostępniać zaawansowaną klasę TextView przeznaczoną do wyświetlania i edycji tekstu. Chcielibyśmy powtórnie wykorzystać tę klasę do zaimplementowania klasy TextShape, ale pakietu narzędziowego nie zaprojektowano pod kątem klas z rodziny Shape. Dlatego nie możemy zamiennie stosować obiektów TextView i Shape.

Jak sprawić, aby gotowe i niepowiązane klasy, takie jak TextView, działały w aplikacji oczekującej klas o odmiennym i niezgodnym interfejsie? Moglibyśmy zmodyfikować klasę TextView, tak aby była zgodna z interfejsem klasy Shape, jednak jest to niewykonalne, jeśli nie znamy kodu źródłowego pakietu narzędziowego. Nawet gdy uzyskamy dostęp do tego kodu, modyfikowanie klasy TextView nie będzie miało sensu. Nie powinno być konieczne zapewnianie obsługi interfejsów specyficznych dla dziedziny przez pakiet narzędziowy tylko po to, aby umożliwić działanie jednej aplikacji.

Zamiast tego możemy zdefiniować klasę `TextShape` w taki sposób, aby dostosowywała (*adapto-wała*) interfejs `TextView` do klasy `Shape`. Można to zrobić na dwa sposoby: (1) przez odziedziczenie interfejsu klasy `Shape` i implementacji klasy `TextView` oraz (2) przez złożenie egzemplarzy klas `TextView` i `TextShape` oraz zaimplementowanie klasy `TextShape` w kategoriach interfejsu klasy `TextView`. Te dwa podejścia odpowiadają wersjom klasowej i obiektowej wzorca Adapter. Klasę `TextShape` nazywamy w tym kontekście **adapterem**.

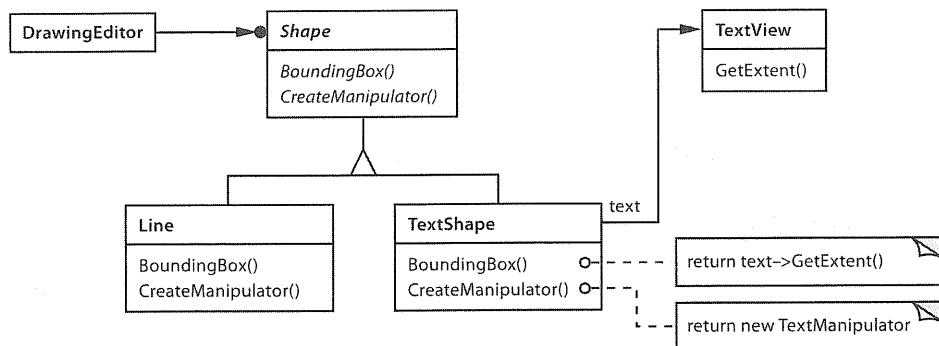


Diagram ten przedstawia adapter obiektowy. Pokazuje, jak żądania `BoundingBox` zadeklarowane w klasie `Shape` są przekształcane na żądania `GetExtent` zdefiniowane w klasie `TextView`. Ponieważ klasa `TextShape` dostosowuje klasę `TextView` do interfejsu `Shape`, w edytorze graficznym można ponownie wykorzystać ogólnie niezgodną klasę `TextView`.

Adapter często odpowiada za wykonywanie zadań nieobsługiwanych przez dostosowywaną klasę. Diagram ilustruje, w jaki sposób adapter może spełniać takie obowiązki. Użytkownik powinien mieć możliwość interaktywnego przeciągnięcia każdego obiektu `Shape` w nowe miejsce, jednak klasa `TextView` nie zaprojektowano z myślą o takim mechanizmie. W klasie `TextShape` trzeba dodać brakującą funkcję przez zaimplementowanie operacji `CreateManipulator` klasie `Shape`. Operacja ta ma zwracać egzemplarz odpowiedniej podklasy klasy `Manipulator`.

`Manipulator` to klasa abstrakcyjna reprezentująca obiekty, które potrafią animować obiekty `Shape` w odpowiedzi na działania użytkownika (na przykład przeciągnięcie figury w nowe miejsce). Istnieją podklasy klasy `Manipulator` odpowiadające różnym figurom. Na przykład klasa `TextManipulator` to podklasa powiązana z klasą `TextShape`. Przez zwrócenie egzemplarza klasy `TextManipulator` można dodać do klasy `TextShape` funkcje wymagane przez klasę `Shape`, ale nieobecne w klasie `TextView`.

WARUNKI STOSOWANIA

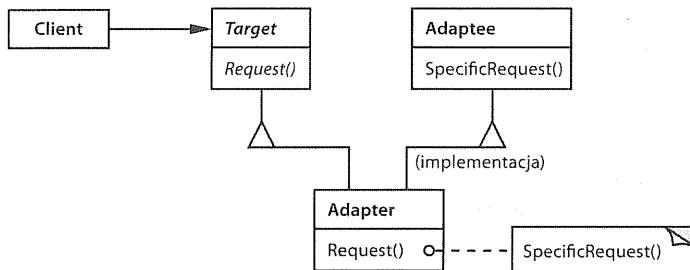
Wzorcu Adapter należy używać w poniższych warunkach:

- Jeśli chcesz wykorzystać istniejącą klasę, ale jej interfejs nie pasuje do tego, który jest potrzebny.
- Kiedy chcesz utworzyć klasę do wielokrotnego użytku współdziałającą z niepowiązanymi lub nieznanymi klasami (czyli takimi, które niekoniecznie będą miały interfejsy zgodne z rozwijaną klasą).

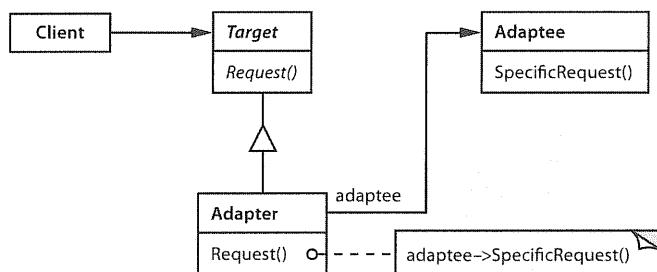
- (Dotyczy tylko adaptera obiektowego) Jeżeli trzeba użyć kilku istniejących podklas, ale dostosowywanie ich interfejsów przez utworzenie dla każdej z nich następnej podklasy jest niepraktyczne. Adapter obiektowy może dostosować interfejs swojej klasy nadzędnej.

STRUKTURA

W adapterze klasowym wykorzystuje się wielodziedziczenie do dostosowywania jednego interfejsu do innego:



Adapter obiektowy jest oparty na składaniu obiektów.



ELEMENTY

- **Target (Shape)**, czyli element docelowy:
 - definiuje specyficzny dla dziedziny interfejs używany przez klienta.
- **Client (DrawingEditor)**:
 - współdziała z obiektami zgodnymi z interfejsem klasy Target.
- **Adaptee (TextView)**, czyli element dostosowywany (adaptowany):
 - definiuje istniejący interfejs, który trzeba dostosować.
- **Adapter (TextShape)**:
 - dostosowuje interfejs klasy Adaptee do interfejsu klasy Target.

WSPÓŁDZIAŁANIE

- Klienci wywołują operacje egzemplarzy klasy Adapter. Z kolei adapter wywołuje operacje klasy Adaptee, które obsługują żądanie.

KONSEKWENCJE

Adaptery klasowe i obiektowe mają odmienne wady oraz zalety. Adapter klasowy:

- ▶ dostosowuje klasę *Adaptee* do klasy *Target* przez dopasowanie się do klasy konkretnej *Adaptee*; powoduje to, że adapter klasowy nie zadziała, jeśli zechcemy dostosować klasę oraz wszystkie jej podklasy;
- ▶ umożliwia przesłonięcie w klasie *Adapter* wybranych działań klasy *Adaptee* (dzieje się tak, ponieważ *Adapter* to podkласa klasy *Adaptee*);
- ▶ powoduje dodanie tylko jednego obiektu, a uzyskanie dostępu do dostosowywanej klasy nie wymaga dodatkowego poziomu pośredniego w postaci wskaźnika.

Adapter obiektowy:

- ▶ umożliwia współdziałanie jednej klasy *Adapter* z wieloma klasami *Adaptee* (czyli z samą klasą *Adaptee* i z wszystkimi jej podklasami, jeśli takie istnieją); w klasie *Adapter* można też dodać funkcje do wszystkich klas *Adaptee* jednocześnie;
- ▶ utrudnia przesłanianie zachowań z klasy *Adaptee*; wymaga to utworzenia podklasy klasy *Adaptee* i wskazywania w klasie *Adapter* tej podklasy zamiast samej klasy *Adaptee*.

Oto kilka kwestii, które należy rozważyć w czasie korzystania ze wzorca *Adapter*:

1. *Jak duży jest zakres dostosowywania w klasie Adapter?* Adaptery różnią się pod względem zakresu prac wykonywanych przy dostosowywaniu klasy *Adaptee* do interfejsu klasy *Target*. Mogą one obejmować bardzo różnorodne zadania — od prostego przekształcania interfejsu (na przykład poprzez zmianę nazw operacji) po obsługę zupełnie odmiennego zestawu operacji. Zakres pracy wykonywanej przez adapter zależy od tego, jak bardzo interfejs klasy *Target* jest podobny do interfejsu klasy *Adaptee*.
2. *Adaptery dołączalne.* Możliwość wielokrotnego wykorzystania klasy wzrasta po zminimalizowaniu założeń, które trzeba poczynić przy korzystaniu z niej w innych klasach. Przez wbudowanie możliwości dostosowania interfejsu w klasę można wyeliminować założenie, że inne klasy korzystają z tego samego interfejsu. Ujmijmy to inaczej — dostosowywanie interfejsu pozwala włączyć klasę w istniejące systemy przygotowane do korzystania z klas o innym interfejsie. W kontekście języka ObjectWorks\Smalltalk [Par90] klasy z wbudowaną możliwością dostosowywania interfejsu nazywane są **adapterami dołączalnymi** (ang. *pluggable adapter*).

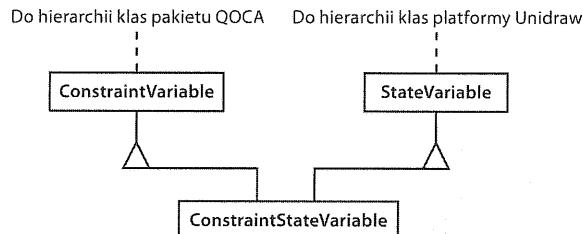
Rozważmy widget *TreeDisplay* potrafiący wyświetlać struktury drzewiaste w postaci graficznej. Jeśli jest to specjalny widget przeznaczony do użytku w jednej tylko aplikacji, możemy wymagać, aby wyświetlane przez niego obiekty miały specyficzny interfejs. Klasy tych obiektów muszą być podklasami klasy abstrakcyjnej *Tree*. Jednak jeżeli chcemy zwiększyć możliwość powtórnego wykorzystania tego widgetu (na przykład w celu umieszczenia go w pakiecie narzędziowym przydatnych widgetów), ten wymóg jest nieuzasadniony. W poszczególnych aplikacjach struktury drzewiaste zdefiniowane są za pomocą różnych klas. Nie należy wymuszać na autorach programów korzystania z naszej klasy abstrakcyjnej *Tree*. Poszczególne struktury drzewiaste będą miały różne interfejsy.

Na przykład w hierarchii katalogów dostęp do elementów podrzędnych może zapewniać operacja `GetSubdirectories`, natomiast w hierarchii dziedziczenia podobna operacja może nosić nazwę `GetSubclasses`. Widget `TreeDisplay` przeznaczony do wielokrotnego użytku musi umożliwiać wyświetlanie hierarchii obu rodzajów, nawet jeśli mają one różne interfejsy. Oznacza to, że w klasę `TreeDisplay` należy wbudować możliwość dostosowywania interfejsu.

W punkcie „Implementacja” przyjrzymy się różnym sposobom wbudowywania w klasy możliwości dostosowania interfejsu.

3. *Stosowanie adapterów dwukierunkowych w celu zapewnienia przezroczystości.* Z adapterami związany jest pewien potencjalny problem — nie dla wszystkich klientów są one przezroczyste. Dostosowany obiekt nie jest zgodny z interfejsem klasy `Adaptee`, dlatego nie zawsze można go użyć tam, gdzie obiektu tej klasy. **Adaptery dwukierunkowe** pozwalają zapewnić przezroczystość i są przydatne wtedy, kiedy dwa różne klienci potrzebują dostępu do innych interfejsów obiektu.

Zastanówmy się nad dwukierunkowym adapterem integrującym platformę do tworzenia edytorów graficznych Unidraw [VL90] i pakiet narzędziowy do rozwiązywania problemów z ograniczeniami QOCA [HHMV92]. W obu systemach znajdują się klasy bezpośrednio reprezentujące zmienne. W platformie Unidraw jest to klasa `StateVariable`, a w pakiecie QOCA — klasa `ConstraintVariable`. Aby platforma Unidraw współdziałała z pakietem QOCA, klasę `ConstraintVariable` trzeba dostosować do klasy `StateVariable`. Aby pakiet QOCA mógł przesyłać rozwiązania do platformy Unidraw, klasę `StateVariable` trzeba dostosować do klasy `ConstraintVariable`.



Rozwiązanie polega na zastosowaniu dwukierunkowego adaptera klasowego `Constraint` \rightarrow `StateVariable`. Jest to podklasa obu klas (`StateVariable` i `ConstraintVariable`) dostosowująca ich interfejsy do siebie. Wielodziedziczenie jest tu możliwym rozwiązaniem, ponieważ interfejsy dostosowywanych klas znacznie różnią się od siebie. Adapter dwukierunkowy jest dopasowany do obu dostosowywanych klas i może działać w każdym systemie.

IMPLEMENTACJA

Choć implementacja wzorca Adapter jest zwykle prosta, poniżej opisujemy kilka zagadnień, o których warto pamiętać:

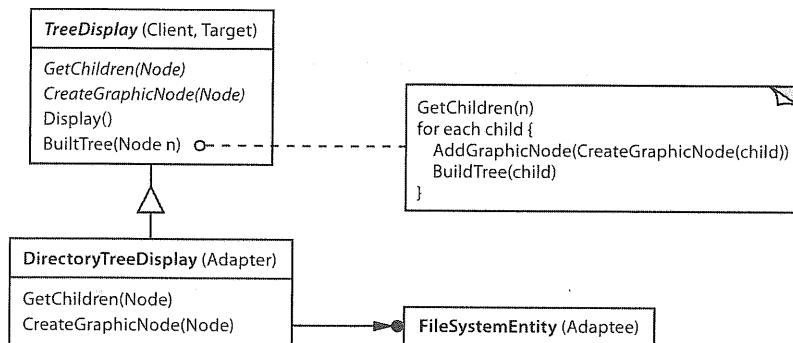
1. *Implementowanie adapterów klasowych w języku C++.* W implementacji adaptera klasowego w języku C++ klasa `Adapter` dziedziczy publicznie po klasie `Target` i prywatnie po klasie `Adaptee`. Dlatego `Adapter` to podtyp typu `Target`, ale już nie typu `Adaptee`.

2. *AdAPTERY DOŁĄCZALNE.* Przyjrzyjmy się trzem sposobom implementowania adapterów dołączalnych dla opisanego wcześniej widgetu TreeDisplay (służy on do automatycznego określania układu i wyświetlania struktur hierarchicznych).

Pierwszy krok, wspólny we wszystkich trzech omawianych tu implementacjach, polega na znalezieniu zawiązzonego interfejsu klasy Adaptee, czyli najmniejszego zbioru operacji, które pozwolą dostosować daną klasę. Zawiązany interfejs składający się tylko z kilku operacji łatwiej jest dostosować niż interfejs obejmujący ich kilkadziesiąt. W przypadku widgetu TreeDisplay dostosowywana może być dowolna struktura hierarchiczna. Najmniejszy interfejs może obejmować dwie operacje — pierwszą do określania, jak graficznie przedstawić węzeł w strukturze hierarchicznej, i drugą do pobierania elementów podległych węzła.

Zawiązany interfejs można zaimplementować na trzy sposoby:

- Za pomocą operacji abstrakcyjnych.* Należy zdefiniować w klasie TreeDisplay odpowiednie operacje abstrakcyjne zawiązzonego interfejsu dostosowywanej klasy. W podklasach trzeba zaimplementować te operacje i dostosować obiekt o strukturze hierarchicznej. Na przykład w podklasie DirectoryTreeDisplay należy zaimplementować te operacje za pomocą operacji dostępu do struktury katalogów.



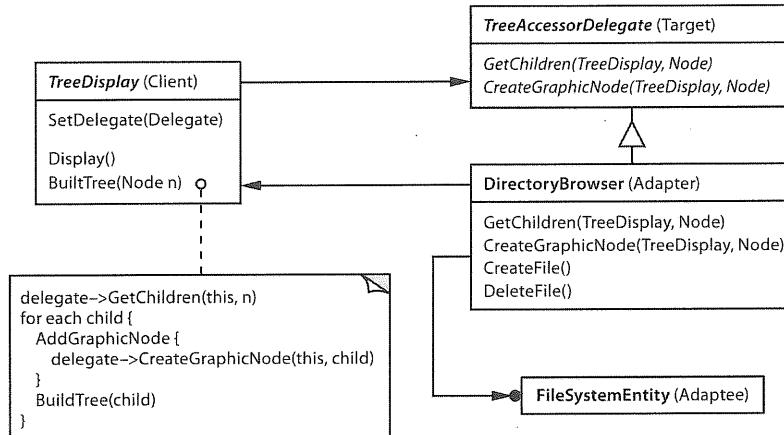
W klasie DirectoryTreeDisplay zawiązany interfejs jest wyspecjalizowany w taki sposób, że pozwala wyświetlić strukturę katalogu składającą się z obiektów FileSystemEntity.

- Za pomocą delegatów.* W tym podejściu obiekt TreeDisplay przekazuje żądania związane z dostępem do struktury hierarchicznej do obiektu **delegata**. Aby zastosować w klasie TreeDisplay inną strategię dostosowywania, należy użyć innego delegata.

Załóżmy na przykład, że z widgetu TreeDisplay korzysta klasa DirectoryBrowser. Klasa ta może okazać się dobrym delegatem dostosowującym widget do hierarchicznej struktury katalogów. W językach z dynamiczną kontrolą typów, takich jak Smalltalk lub Objective C, to podejście wymaga jedynie utworzenia interfejsu do rejestrowania delegata z adapterem. Następnie widget TreeDisplay może po prostu przekazywać żądania delegatowi. W systemie NEXTSTEP [Add94] podejście to pozwoliło znacznie ograniczyć liczbę podklaśs.

W językach ze statyczną kontrolą typów, na przykład w C++, trzeba jawnie zdefiniować interfejs delegata. Można to zrobić przez umieszczenie zawiązzonego interfejsu potrzebnego klasie TreeDisplay w klasie abstrakcyjnej TreeAccessorDelegate. Następnie można

za pomocą dziedziczenia dołączyć ten interfejs do wybranego delegata (tu jest to klasa `DirectoryBrowser`). Jeśli klasa `DirectoryBrowser` nie ma klasy nadzędnej, należy zastosować zwykłe dziedziczenie. W przeciwnym razie trzeba wykorzystać wielodziedziczenie. Mieszanie klas w ten sposób jest łatwiejsze niż wprowadzanie nowej podklasy klasy `TreeDisplay` i implementowanie jej poszczególnych operacji.



- c) Za pomocą adapterów parametryzowanych. Standardowy sposób obsługi adapterów dołączalnych w języku Smalltalk polega na parametryzowaniu ich za pomocą jednego lub kilku bloków. Bloki umożliwiają dostosowanie klasy bez tworzenia podklasy. Blok może dostosowywać żądanie, a w adapterze można umieścić bloki powiązane z poszczególnymi żądaniami. W omawianym przykładzie oznacza to, że w klasie `TreeDisplay` należy umieścić jeden blok do przekształcania węzłów na obiekty `GraphicNode` i drugi blok do obsługi dostępu do elementów podrzędnych węzłów.

Na przykład aby utworzyć obiekt `TreeDisplay` dla hierarchii katalogów, należy użyć następującego kodu:

```

directoryDisplay :=
  (TreeDisplay on: treeRoot)
    getChildrenBlock:
      [mode | node getSubdirectories]
    createGraphicNodeBlock:
      [:node | node createGraphicNode].
  
```

Jeśli wbudujemy możliwość dostosowania interfejsu w klasę, to podejście będzie wygodną alternatywą dla tworzenia podklas.

PRZYKŁADOWY KOD

Przedstawmy zarys implementacji adapterów klasowych i obiektowych dla przykładu z punktu „Uzasadnienie”. Zaczniemy od klas `Shape` i `TextView`.

```

class Shape {
public:
    Shape();
    virtual void BoundingBox(
  
```

```

        Point& bottomLeft, Point& topRight
    ) const;
    virtual Manipulator* CreateManipulator() const;
};

class TextView {
public:
    TextView();
    void GetOrigin(Coord& x, Coord& y) const;
    void GetExtent(Coord& width, Coord& height) const;
    virtual bool IsEmpty() const;
};

```

W klasie Shape zakładamy, że ramka ograniczająca jest wyznaczana przez jej przeciwnielego wierzchołki. Z kolei obiekt TextView jest zdefiniowany za pomocą punktu początkowego, wysokości i szerokości. Klasa Shape obejmuje też definicję operacji CreateManipulator. Tworzy ona obiekt Manipulator potrafiący animować figury w czasie manipulowania nimi przez użytkownika¹. W klasie TextView nie ma podobnej operacji. Klasa TextShape pełni funkcję adaptora między tymi odmiennymi interfejsami.

W adapterze klasowym do dostosowywania interfejsów służy wielodziedziczenie. Niezwykle istotne jest, aby użyć jednej gałęzi do dziedziczenia interfejsu, a innej — do dziedziczenia implementacji. W języku C++ podział ten odbywa się przez dziedziczenie interfejsu w sposób publiczny, a implementacji — prywatny. Wykorzystamy to podejście do zdefiniowania adaptera TextShape.

```

class TextShape : public Shape, private TextView {
public:
    TextShape();

    virtual void BoundingBox(
        Point& bottomLeft, Point& topRight
    ) const;
    virtual bool IsEmpty() const;
    virtual Manipulator* CreateManipulator() const;
};

```

Operacja BoundingBox przekształca interfejs TextView, tak aby dostosować go do interfejsu klasy Shape.

```

void TextShape::BoundingBox (
    Point& bottomLeft, Point& topRight
) const {
    Coord bottom, left, width, height;

    GetOrigin(bottom, left);
    GetExtent(width, height);

    bottomLeft = Point(bottom, left);
    topRight = Point(bottom + height, left + width);
}

```

¹ Operacja CreateManipulator to przykład zastosowania wzorca Metoda wytwarzca (s. 107).

Operacja `IsEmpty` ilustruje bezpośrednie przekazywanie żądań. Technika ta jest powszechnie stosowana w implementacjach adapterów.

```
bool TextShape::IsEmpty () const {
    return TextView::IsEmpty();
}
```

W ostatnim kroku utworzymy od podstaw operację `CreateManipulator` (klaśa `TextView` jej nie udostępnia). Zakładamy, że zaimplementowaliśmy już klasę `TextManipulator` obsługującą manipulowanie obiektami `TextShape`.

```
Manipulator* TextShape::CreateManipulator () const {
    return new TextManipulator(this);
}
```

W adapterze obiektowym zastosujemy składanie obiektów do połączenia klas o różnych interfejsach. W tym podejściu adapter `TextShape` przechowuje wskaźnik do obiektu `TextView`.

```
class TextShape : public Shape {
public:
    TextShape(TextView*);

    virtual void BoundingBox(
        Point& bottomLeft, Point& topRight
    ) const;
    virtual bool IsEmpty() const;
    virtual Manipulator* CreateManipulator() const;
private:
    TextView* _text;
};
```

W klasie `TextShape` trzeba zainicjować wskaźnik do egzemplarza klasy `TextView`. Odbywa się to w konstruktorze. Ponadto przy każdym wywołaniu operacji tej klasy trzeba wywoływać operacje na obiekcie `TextView`. W tym miejscu zakładamy, że klient tworzy obiekt `TextView` i przekazuje go do konstruktora klasy `TextShape`.

```
TextShape::TextShape (TextView* t) {
    _text = t;
}

void TextShape::BoundingBox (
    Point& bottomLeft, Point& topRight
) const {
    Coord bottom, left, width, height;

    _text->GetOrigin(bottom, left);
    _text->GetExtent (width, height) ;

    bottomLeft = Point(bottom, left);
    topRight = Point(bottom + height, left + width);
}

bool TextShape::IsEmpty () const {
    return _text->IsEmpty();
}
```

Implementacji operacji `CreateManipulator` nie trzeba zmieniać w stosunku do klasowej wersji adaptora, ponieważ utworzyliśmy tę operację od podstaw i nie wykorzystaliśmy w niej powtórnie żadnych istniejących funkcji klasy `TextView`.

```
Manipulator* TextShape::CreateManipulator () const {
    return new TextManipulator(this);
}
```

Porównajmy ten kod do kodu adaptora klasowego. Napisanie adaptora obiektowego wymaga niewiele więcej wysiłku, a rozwiązanie to jest znacznie elastyczniejsze. Wersja obiektowa adaptora `TextShape` będzie działać równie dobrze na przykład z podklasami klasy `TextView`. Wystarczy, że klient przekaże do konstruktora klasy `TextShape` egzemplarz podklasses `TextView`.

ZNANE ZASTOSOWANIA

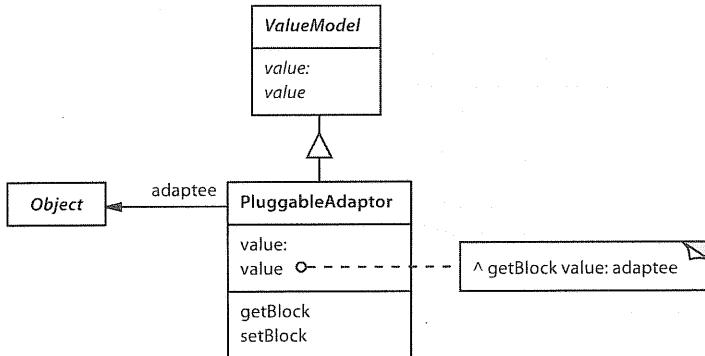
Przykład z punktu „Uzasadnienie” pochodzi z programu `ET++Draw`. Jest to aplikacja graficzna oparta na platformie `ET++` [WGM88]. W programie `ET++Draw` ponownie wykorzystano klasy do edycji tekstu z platformy `ET++` przez zastosowanie adaptora w postaci klasy `TextShape`.

W pakiecie narzędziowym `InterViews 2.6` zdefiniowana jest klasa abstrakcyjna `Interactor`. Na jej podstawie powstają elementy interfejsu użytkownika, takie jak paski przewijania, przyciski i menu [VL88]. Pakiet obejmuje też klasę abstrakcyjną `Graphic` do tworzenia strukturalnych obiektów graficznych, takich jak linie, okręgi, wielokąty i splajny. Obie te klasy mają reprezentację graficzną, ale posiadają odmienne interfejsy i implementacje (nie mają wspólnej klasy nadzędnej), dlatego są niezgodne ze sobą. Nie można bezpośrednio zagnieździć strukturalnego obiektu graficznego na przykład w oknie dialogowym.

Dlatego w pakiecie `InterViews 2.6` zdefiniowano adapter o nazwie `GraphicBlock`. Jest to podklasa klasy `Interactor` obejmująca egzemplarz klasy `Graphic`. Klasa `GraphicBlock` dostosowuje interfejs klasy `Graphic` do interfejsu klasy `Interactor` oraz umożliwia wyświetlanie, przewijanie i przybliżanie egzemplarzy klasy `Graphic` w strukturze klasy `Interactor`.

Adaptry dołączalne są powszechnie spotykane w języku `ObjectWorks\Smalltalk` [Par90]. W standardowej wersji języka `Smalltalk` zdefiniowana jest klasa `ValueModel` przeznaczona dla widoków wyświetlających pojedynczą wartość. Klasa ta udostępnia interfejs w postaci operacji `value` i `value:`, umożliwiający uzyskanie dostępu do wartości. Te operacje to metody abstrakcyjne. Autorzy aplikacji korzystają z wartości za pomocą bardziej specyficznych dla dziedziny nazw, takich jak `width` i `width:`, ale nie powinni być zmuszeni do tworzenia podklas klasy `ValueModel` w celu dostosowania tych nazw do interfejsu owej klasy.

Dlatego w języku `ObjectWorks\Smalltalk` umieszczono podklasses `PluggableAdapter` klasy `ValueModel`. Obiekty `PluggableAdapter` dostosowują inne obiekty do interfejsu klasy `ValueModel` (`value`, `value:`) i można je sparametryzować za pomocą bloków do pobierania i ustawiania odpowiedniej wartości. W klasie `PluggableAdapter` bloki te są używane wewnętrznie do implementowania interfejsu `value`, `value:`. Ponadto klasa ta umożliwia bezpośrednie przekazanie nazw selektorów (na przykład `width` i `width:`), co jest wygodnym rozwiązaniem. Obiekty `PluggableAdapter` automatycznie przekształcają te selektory na odpowiadające im bloki.



Innym przykładem z języka ObjectWorks\Smalltalk jest klasa `TableAdaptor`. Potrafi ona dostosować sekwencję obiektów do postaci tabelarycznej. Tabela wyświetla jeden obiekt na wiersz. Klient parametryzuje obiekt `TableAdaptor` za pomocą zestawu komunikatów, które tabela może wykorzystać do pobrania wartości kolumn z obiektu.

W niektórych klasach pakietu AppKit [Add94] firmy NeXT do dostosowywania interfejsu zastosowano delegaty. Przykładem jest klasa `NXBrower` potrafiąca wyświetlać hierarchiczne listy danych. Klasa ta korzysta z delegata do dostępu do danych i ich dostosowywania.

Opracowana przez Meyera technika „małżeństwo z rozsądku” [Mey88] to odmiana adaptera klasowego. Meyer opisuje, w jaki sposób klasa `FixedStack` dostosowuje implementację klasy `Array` do interfejsu klasy `Stack`. W efekcie powstaje stos o stałej liczbie elementów.

POWIĄZANE WZORCE

Wzorzec Most (s. 181) ma strukturę podobną do adaptera obiektowego, jednak pełni inne funkcje — ma rozdzielać interfejs od implementacji, aby można je modyfikować łatwo i niezależnie od siebie. Adapter służy do zmieniania interfejsu *istniejącego* obiektu.

Wzorzec Dekorator (s. 152) pozwala wzbogacać inne obiekty bez zmiany ich interfejsów. Dlatego jest bardziej przezroczysty dla aplikacji niż adapter. Oznacza to też, że wzorzec Dekorator umożliwia składanie rekurencyjne, niemożliwe przy stosowaniu czystych adapterów.

Wzorzec Pełnomocnik (s. 191) dotyczy definiowania substytutu lub zastępnika innego obiektu, a nie modyfikowania jego interfejsu.

DEKORATOR (DECORATOR)

obiektowy, strukturalny

PRZEZNACZENIE

Dynamicznie dodaje dodatkowe obowiązki do obiektu. Wzorzec ten udostępnia alternatywny elastyczny sposób tworzenia podklas o wzbogaconych funkcjach.

INNE NAZWY

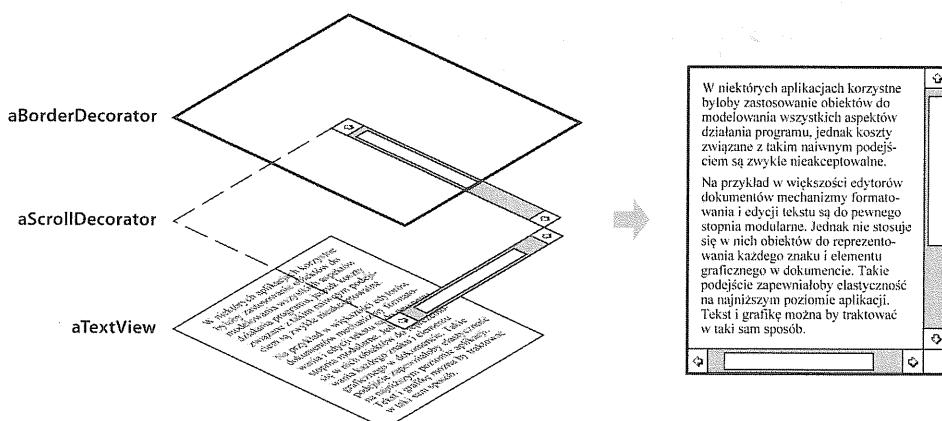
Nakładka (ang. *wrapper*).

UZASADNIENIE

Czasem chcemy dodać zadania do pojedynczych obiektów, a nie do całej klasy. Pakiet narzędziowy do tworzenia graficznych interfejsów użytkownika powinien na przykład umożliwiać dodawanie atrybutów (takich jak ramki) i zachowań (takich jak obsługa przewijania) do dowolnego komponentu z takich interfejsów.

Jednym ze sposobów na dodanie zadań jest dziedziczenie. Odziedziczenie ramki po innej klasie powoduje dodanie obramowania do każdego egzemplarza danej podklasty. Jest to jednak nieelastyczne rozwiązanie, ponieważ wybór ramki odbywa się statycznie. Klient nie może kontrolować sposobu i czasu ozdabiania komponentu za pomocą obramowania.

Elastyczniejsze podejście polega na umieszczeniu komponentu w innym obiekcie, który doda ramkę. Ten zewnętrzny obiekt to tak zwany **dekorator**. Dekorator jest zgodny z interfejsem ozdabianego komponentu, dlatego jego obecność jest niezauważalna dla klientów danego komponentu. Dekorator przekazuje żądania do komponentu, a przed ich wysłaniem lub potem może wykonywać dodatkowe działania (na przykład wyświetlać obramowanie). Ta przeroczeństwo umożliwia rekurencyjne zagnieżdżanie dekoratorów, a tym samym dodanie dowolnej liczby nowych zadań.

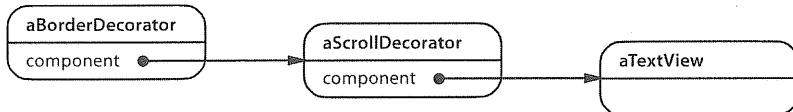


W niektórych aplikacjach korzystane były zastosowanie obiektów do modelowania wszystkich aspektów działania programu, jednak koszty związane z takim naivnym podejściem są zwykle nieakceptowalne.

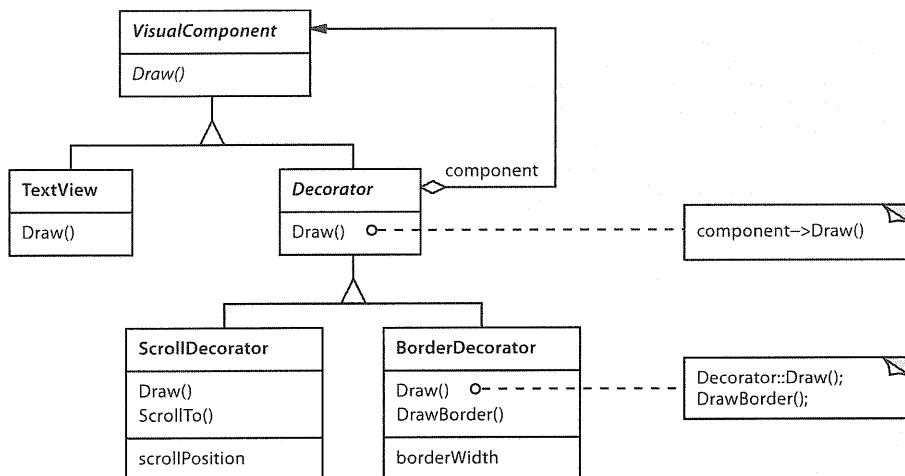
Na przykład w większości edytorów dokumentów mechanizmy formatowania i edycji tekstu są do pewnego stopnia modularne. Jednak nie stosuje się w nich obiektów do reprezentowania każdego znaku i elementu graficznego w dokumencie. Takie podejście zapewniałoby elastyczność na najniższym poziomie aplikacji. Tekst i grafikę można by traktować w taki sam sposób.

Załóżmy na przykład, że korzystamy z obiektu `TextView` do wyświetlania tekstu w oknie. Obiekt ten domyślnie nie ma pasków przewijania, ponieważ nie zawsze są one potrzebne. Jednak kiedy chcemy ich użyć, możemy wykorzystać klasę `ScrollDecorator`, aby je dodać. Przyjmijmy też, że wokół obiektu `TextView` zamierzamy wyświetlić grubą czarną ramkę. Aby dodać ten element, możemy wykorzystać klasę `BorderDecorator`. Wystarczy złożyć oba dekoratory z obiektem `TextView`, żeby uzyskać pożądany efekt.

Poniższy diagram obiektów pokazuje, jak złożyć obiekt `TextView` z obiektami `BorderDecorator` i `ScrollDecorator` w celu dodania obramowania oraz pasków przewijania do okna z tekstem.



Klasy `ScrollDecorator` i `BorderDecorator` to podklasy klasy `Decorator`. Jest to klasa abstrakcyjna reprezentująca komponenty wizualne służące do ozdabiania innych komponentów wizualnych.



`VisualComponent` to klasa abstrakcyjna reprezentująca obiekty wizualne. Definiuje ona interfejs do wyświetlania i obsługi zdarzeń takich obiektów. Warto zauważyć, że klasa `Decorator` po prostu przekazuje żądania wyświetlenia do komponentu, a w jej podkласach można rozszerzyć tę operację.

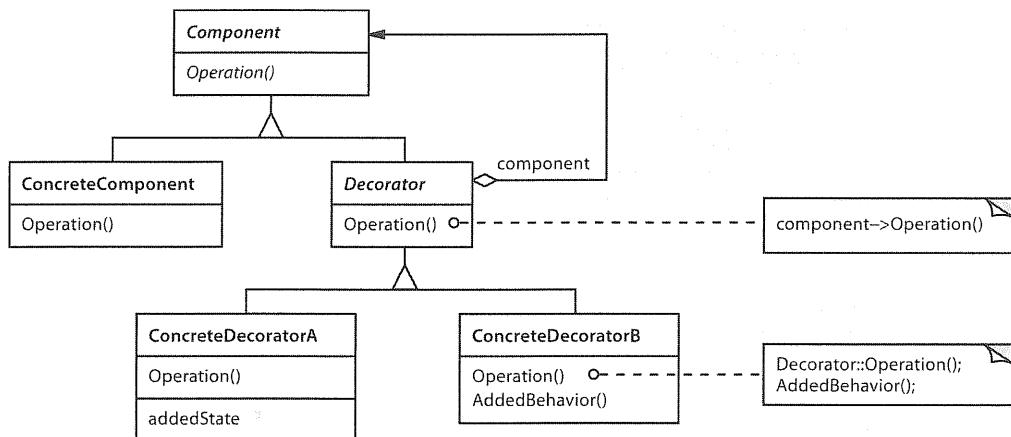
W podkласach klasy `Decorator` można dodać operacje obsługujące określone funkcje. Na przykład operacja `ScrollTo` klasy `ScrollDecorator` umożliwia innym obiektom przewijanie interfejsu, jeśli wykryją one, że w interfejsie dostępny jest obiekt `ScrollDecorator`. Ważną cechą omawianego wzorca jest to, że umożliwia umieszczenie dekoratorów wszędzie tam, gdzie mogą pojawić się obiekty `VisualComponent`. Dzięki temu klienci zwykle nie potrafią odróżnić ozdobionych komponentów od pozostałych, dlatego w ogóle nie są zależne od ozdobników.

WARUNKI STOSOWANIA

Wzorzec Dekorator należy stosować w następujących warunkach:

- ▶ Kiedy trzeba dodawać zadania do poszczególnych obiektów w dynamiczny i przezroczysty sposób (czyli bez wpływu na inne obiekty).
- ▶ Jeśli potrzebny jest mechanizm do obsługi zadań, które można cofnąć.
- ▶ Jeżeli rozszerzanie przez tworzenie podklas jest niepraktyczne. Czasem możliwe jest utworzenie dużej liczby niezależnych rozszerzeń, co prowadzi do znacznego wzrostu liczby podklas potrzebnych do obsługi wszystkich kombinacji. Ponadto definicja klasy może być ukryta lub w inny sposób uniemożliwiać utworzenie podklas.

STRUKTURA



ELEMENTY

- ▶ **Component** (*VisualComponent*):
 - definiuje interfejs obiektów, do których można dynamicznie dodawać obsługę zadań.
- ▶ **ConcreteComponent** (*TextView*):
 - definiuje obiekt, do którego można dołączyć obsługę dodatkowych zadań.
- ▶ **Decorator**:
 - przechowuje referencję do obiektu **Component** i definiuje interfejs zgodny z interfejsem klasy **Component**.
- ▶ **ConcreteDecorator** (*BorderDecorator*, *ScrollDecorator*):
 - dodaje zadania do komponentu.

WSPÓŁDZIAŁANIE

- ▶ Obiekt **Decorator** przekazuje żądania do powiązanego z nim obiektu **Component**. Opcjonalnie może też wykonywać dodatkowe operacje przed przesłaniem żądania lub potem.

KONSEKWENCJE

Wzorzec Dekorator ma przynajmniej dwie podstawowe zalety i dwie wady.

1. *Zapewnia większą elastyczność niż statyczne dziedziczenie.* Wzorzec Dekorator w porównaniu ze statycznym (wielo)dziedziczeniem zapewnia elastyczniejszy sposób dodawania zadań do obiektów. Korzystając z tego wzorca, zadania można dodawać i usuwać w czasie wykonywania programu przez proste dołączanie oraz odłączanie dekoratora. Zupełnie inaczej odbywa się to przy dziedziczeniu, kiedy to dla każdego dodatkowego zadania trzeba utworzyć nową klasę (na przykład `BorderedScrollView` lub `BorderedTextView`). Powoduje to powstanie wielu klas i zwiększa złożoność systemu. Ponadto udostępnienie różnych klas `Decorator` dla określonej klasy `Component` pozwala łączyć i dopasowywać zadania.
2. *Pozwala uniknąć tworzenia przeładowanych funkcjami klas na wysokich poziomach hierarchii.* Dekoratory umożliwiają zastosowanie podejścia „płać za to, z czego korzystasz” przy dodawaniu obsługi zadań. Zamiast próbować zapewnić obsługę wszystkich możliwych do przewidzenia funkcji w złożonej i umożliwiającej dostosowywanie klasie, można zdefiniować prostą klasę i dodawać funkcje stopniowo za pomocą obiektów `Decorator`. Możliwości obiektu można wtedy budować z prostych mechanizmów. Powoduje to, że w aplikacji nie trzeba ponosić kosztów związanych z nieużywanymi funkcjami. Ponadto można łatwo zdefiniować nowe rodzaje dekoratorów niezależnie od klas rozszerzanych obiektów (dotyczy to nawet nieprzewidzianych wcześniej rozszerzeń). W czasie rozszerzania złożonych klas często można odkryć szczegóły niepowiązane z dodawanymi zadaniami.
3. *Dekorator i powiązany z nim komponent nie są identyczne.* Dekorator działa jak niewidoczna otoczka. Jednak pod względem identyczności obiektów udekorowany komponent różni się od jego standardowej wersji, dlatego przy korzystaniu z dekoratorów nie należy zakładać, że obiekty będą identyczne.
4. *Powstawanie wielu małych obiektów.* Projekt, w którym wykorzystano wzorzec Dekorator, często prowadzi do powstania wielu małych i podobnych do siebie obiektów. Różnią się one tylko sposobem połączenia, a nie klasą lub wartościami zmiennych. Choć takie systemy są łatwe w dostosowywaniu przez rozumiejące je osoby, mogą okazać się trudne do oparowania i diagnozowania.

IMPLEMENTACJA

W czasie stosowania wzorca Dekorator trzeba uwzględnić kilka kwestii.

1. *Zgodność z interfejsem.* Interfejs obiektu dekoratora musi być zgodny z interfejsem ozdabianego komponentu. Dlatego klasy `ConcreteDecorator` powinny dziedziczyć po wspólnej klasie (przynajmniej w języku C++).
2. *Pomijanie klasy abstrakcyjnej Decorator.* Jeśli chcesz dodać tylko jedno zadanie, nie musisz definiować klasy abstrakcyjnej `Decorator`. Sytuacja taka często ma miejsce, kiedy trzeba przekształcić istniejącą hierarchię klas, a nie zaprojektować nową. Można wtedy włączyć zadanie klasy `Decorator` (polegające na przekazywaniu żądań do komponentu) do klasy `ConcreteDecorator`.

3. *Tworzenie prostych klas Component.* Aby interfejsy były zgodne, komponenty i dekoratory powinny dziedziczyć po wspólnej klasie Component. Ważne jest, aby ta wspólna klasa była prosta. Oznacza to, że należy skoncentrować się w niej na definiowaniu interfejsu, a nie na przechowywaniu danych. Definiowanie reprezentacji danych należy odłożyć do czasu tworzenia podklas. W przeciwnym razie złożoność klasy Component może sprawić, że dekoratory staną się zanadto rozbudowane, aby można korzystać z dużej liczby obiektów tego rodzaju. Umieszczenie wielu funkcji w klasie Component zwiększa ponadto prawdopodobieństwo ponoszenia niepotrzebnych kosztów w określonych podklasach.

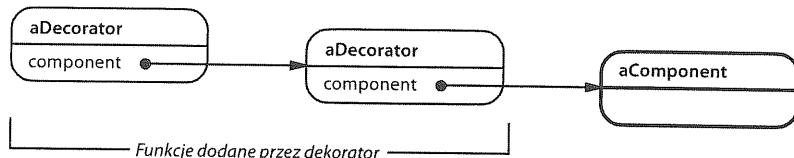
4. *Zmiananie „skórki” a modyfikowanie mechanizmów obiektów.* Możemy potraktować dekorator jak „skórkę” obiektu zmieniającą jego działanie. Inna możliwość to zmodyfikowanie wewnętrznych mechanizmów obiektu. Strategia (s. 321) to dobry przykład wzorca przeznaczonego do zmianiania mechanizmów.

Stosowanie strategii to lepsze rozwiązanie, jeśli klasa Component jest z natury rozbudowana, przez co wykorzystanie wzorca Dekorator będzie zbyt kosztowne. We wzorcu Strategia część zadań komponentów jest przekazywana do odrębnego obiektu strategii. Wzorzec ten umożliwia modyfikowanie lub rozszerzanie funkcji komponentu przez zastępowanie obiektów strategii.

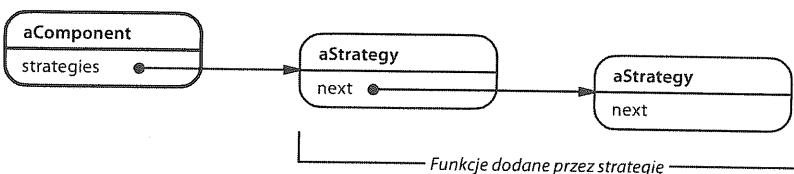
Można na przykład dodać obsługę różnych stylów obramowania przez przeniesienie zadania wyświetlania ramki z komponentu do odrębnego obiektu Border. Obiekt Border to obiekt strategii kapsułkujący strategię wyświetlania obramowania. Przez zwiększenie liczby strategii z jednej do dowolnie długiej listy uzyskamy ten sam efekt, co przy rekurencyjnym zagnieżdżaniu dekoratorów.

W platformach MacApp 3.0 [App89] i Bedrock [Sym93a] komponenty graficzne (tak zwane widoki) przechowują listę „obiektów upiększających”, które pozwalają dołączyć do komponentu widoku dodatkowe upiększenia, na przykład ramki. Jeśli widok jest połączony z upiększeniami, umożliwia im wyświetlenie dodatkowych ozdobników. W platformach MacApp i Bedrock to rozwiązanie jest wymuszone, ponieważ klasa View jest rozbudowana. Korzystanie z kompletnego obiektu View tylko w celu dodania ramki byłoby zbyt kosztowne.

Ponieważ wzorzec Dekorator zmienia komponent jedynie od zewnętrz, w komponentach nie są potrzebne informacje o dekoratorach. Oznacza to, że dekoratory są niewidoczne dla komponentu.



Przy korzystaniu ze strategii komponent sam musi znać możliwe rozszerzenia. Dlatego trzeba w nim wskazywać i przechowywać odpowiednie strategie.



Podejście oparte na wzorcu Strategia może wymagać zmodyfikowania komponentu pod kątem nowych rozszerzeń. Z drugiej strony strategia może mieć własny wyspecjalizowany interfejs, natomiast interfejs dekoratora musi być zgodny z interfejsem komponentu. W strategii wyświetlającej ramkę wystarczy zdefiniować potrzebny do tego interfejs (operacje DrawBorder, GetWidth itd.), co oznacza, że strategia może być prosta nawet wtedy, gdy klasa Component jest rozbudowana.

W platformach MacApp i Bedrock podejście to zastosowano nie tylko do upiększania widoków. Wykorzystano je także do wzbogacenia zachowania obiektów w zakresie obsługi zdarzeń. W obu platformach widok przechowuje listę obiektów reprezentujących zachowania. Obiekty te mogą modyfikować i przechwytywać zdarzenia. Widok umożliwia każdemu z zarejestrowanych obiektów reprezentujących zachowanie obsłużenie zdarzenia i ewentualnie dopiero potem uruchamia zachowania niezarejestrowane, co w efekcie powoduje przesłonięcie tych ostatnich. Można na przykład wzbogacić widok o specjalną obsługę zdarzeń związanych z klawiaturą. W tym celu należy zarejestrować obiekt reprezentujący zachowanie przeznaczony do przechwytywania i obsługi zdarzeń dotyczących klawiszy.

PRZYKŁADOWY KOD

Poniższy kod ilustruje, jak zaimplementować dekoratory interfejsu użytkownika w języku C++. Zakładamy, że odpowiednik klasy Component nosi nazwę VisualComponent.

```
class VisualComponent {
public:
    VisualComponent();

    virtual void Draw();
    virtual void Resize();
    // ...
};
```

Zdefiniujmy podkласę klasy VisualComponent, Decorator. W celu uzyskania różnych dekoracji utworzymy następnie jej podklasy.

```
class Decorator : public VisualComponent {
public:
    Decorator(VisualComponent*);

    virtual void Draw();
    virtual void Resize();
    // ...
private:
    VisualComponent* _component;
};
```

Obiekt Decorator dekoruje obiekt VisualComponent wskazywany przez zmienną egzemplarza _component. Zmienna ta jest inicjowana w konstruktorze. Dla każdej operacji z interfejsu klasy VisualComponent należy w klasie Decorator zdefiniować domyślną implementację przekazującą żądanie do zmiennej _component.

```

void Decorator::Draw () {
    _component->Draw();
}

void Decorator::Resize () {
    _component->Resize();
}

```

Podklasy klasy `Decorator` definiują określone dekoracje. Na przykład klasa `BorderDecorator` dodaje ramkę do komponentu zawierającego obiekt tej klasy. `BorderDecorator` to podklasa klasy `Decorator`. W podklasie tej przesłonięto operację `Draw`, tak aby wyświetlała obramowanie. W `BorderDecorator` zdefiniowano też prywatną operację pomocniczą `DrawBorder`, która rysuje ramkę. Omawiana podklasa dziedziczy implementacje wszystkich pozostałych operacji po klasie `Decorator`.

```

class BorderDecorator : public Decorator {
public:
    BorderDecorator(VisualComponent*, int borderWidth);

    virtual void Draw();
private:
    void DrawBorder(int);
private:
    int _width;
};

void BorderDecorator::Draw () {
    Decorator::Draw();
    DrawBorder(_width);
}

```

Podobnie będzie wyglądała implementacja klas `ScrollDecorator` i `DropShadowDecorator`. Dodają one do wizualnych komponentów obsługę przewijania i cienia.

Teraz możemy złożyć egzemplarze opisanych klas, aby dodać różne dekoracje. Poniższy kod pokazuje, jak wykorzystać dekoratory do utworzenia obiektu `TextView` z ramką i możliwością przewijania.

Najpierw należy umożliwić umieszczenie komponentu wizualnego w obiekcie `Window`. Założymy, że klasa `Window` udostępnia przeznaczoną do tego operację `SetContents`.

```

void Window::SetContents (VisualComponent* contents) {
    // ...
}

```

Teraz możemy utworzyć obiekt `TextView` i okno, w którym go umieścimy.

```

Window* window = new Window;
TextView* textView = new TextView;

```

Obiekt `TextView` jest jednocześnie obiektem typu `VisualComponent`, co umożliwia umieszczenie go w oknie.

```
window->SetContents(textView);
```

Chcemy jednak, aby obiekt `TextView` miał ramkę i umożliwiał przewijanie. Dlatego przed dodaniem go do okna odpowiednio go udekorujemy.

```
window->SetContents(
    new BorderDecorator(
        new ScrollDecorator(textView), 1
    )
}
```

Ponieważ obiekty `Window` mają dostęp do swojej zawartości za pośrednictwem interfejsu klasy `VisualComponent`, nie wykrywają zastosowania dekoratorów. Jednak w kodzie klienta można śledzić obiekt `TextView`, jeśli trzeba bezpośrednio wchodzić z nim w interakcje (na przykład w razie konieczności wywoływania operacji spoza interfejsu klasy `VisualComponent`). Klienty polegające na identyczności komponentów także powinny odwoływać się bezpośrednio do obiektu `TextView`.

ZNANE ZASTOSOWANIA

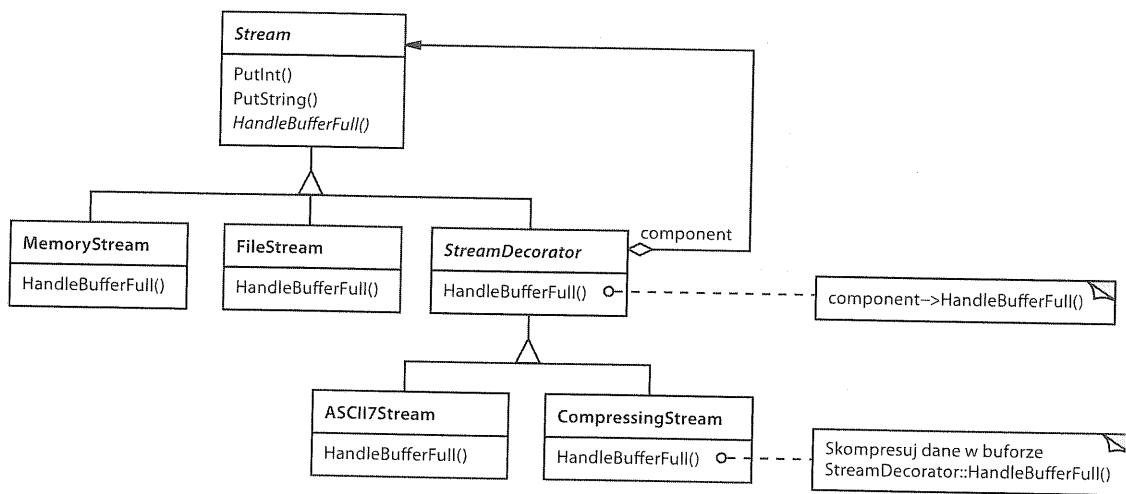
Dekoratory w wielu pakietach narzędziowych przeznaczonych do tworzenia obiektowych interfejsów użytkownika służą do dodawania ozdobników graficznych do widgetów. Do przykładów należą między innymi pakiet InterViews [LVC89, LCI-92], platforma ET++ [WGM88] i biblioteka klas języka ObjectWorks\Smalltalk [Par90]. Bardziej wymyślne zastosowania wzorca Dekorator to klasa `DebuggingGlyph` z pakietu InterViews i klasa `PassivityWrapper` z języka Smalltalk firmy ParcPlace. Obiekty `DebuggingGlyph` wyświetlają informacje diagnostyczne przed i po przekazaniu żądań określenia układu do komponentów zawierających te obiekty. Wyświetlone dane można wykorzystać do analizowania i diagnozowania procesu określania układu przez obiekty w skomplikowanych złożeniach. Obiekty `PassivityWrapper` pozwalają włączyć lub wyłączyć możliwość interakcji użytkownika z danym komponentem.

Jednak zastosowania wzorca Dekorator w żadnym razie nie ograniczają się do graficznych interfejsów użytkownika, co ilustruje następny przykład, oparty na klasach reprezentujących strumienie z platformy ET++ [WGM88].

Strumienie to podstawowa abstrakcja w większości mechanizmów wejścia-wyjścia. Strumień może udostępniać interfejs do przekształcania obiektów na sekwencje bajtów lub znaków. Umożliwia to zapis obiektu do pliku lub do łańcucha znaków w pamięci w celu jego późniejszego odtworzenia. Prostym sposobem na uzyskanie takiego efektu jest zdefiniowanie abstrakcyjnej klasy `Stream` oraz podklas `MemoryStream` i `FileStream`. Założymy jednak, że chcemy, aby można przeprowadzić także poniższe operacje:

- ▶ kompresję danych ze strumienia za pomocą różnych algorytmów kompresji (RLE, Lempel-Ziv itd.);
- ▶ zredukowanie danych ze strumienia do postaci 7-bitowych znaków ASCII, co umożliwi ich transmisję przez kanał komunikacyjny ASCII.

Wzorzec Dekorator zapewnia elegancki sposób dodawania obsługi takich zadań do strumieni. Poniższy diagram przedstawia jedno z rozwiązań opisanego problemu.



Klasa abstrakcyjna `Stream` ma wewnętrzny bufor i udostępnia operacje potrzebne do zapisywania danych do strumienia (`PutInt`, `PutString`). Kiedy bufor się zapłni, obiekt `Stream` wywoła operację abstrakcyjną `HandleBufferFull` odpowiadającą za rzeczywisty transfer danych. W klasie `FileStream` operacja ta jest przesłonięta, a jej nowa wersja zapisuje zawartość bufora do pliku.

Kluczową klasą jest tu `StreamDecorator`. Przechowuje ona referencję do komponentu reprezentującego strumień i przekazuje do niego żądania. Podklasy klasy `StreamDecorator` przesłaniają operację `HandleBufferFull` i wykonują dodatkowe działania przed wywołaniem operacji `HandleBufferFull` z klasy `StreamDecorator`.

Na przykład podklaśa `CompressingStream` kompresuje dane, a klasa `ASCII7Stream` przekształca je na 7-bitowe znaki ASCII. Teraz w celu utworzenia obiektu `FileStream`, który kompresuje dane oraz przekształca skompresowane dane binarne na 7-bitowe znaki ASCII, udekorujemy taki obiekt za pomocą obiektów `CompressingStream` i `ASCII7Stream`.

```

Stream* aStream = new CompressingStream(
    new ASCII7Stream(
        new FileStream("aFileName")
    )
);
aStream->PutInt(12);
aStream->PutString("aString");
    
```

POWIĄZANE WZORCE

Wzorzec Adapter (s. 141). Dekorator różni się od adaptera, ponieważ modyfikuje jedynie zadania obiektu, a nie jego interfejs. Adapter zapewnia obiektowi zupełnie nowy interfejs.

Kompozyt (s. 170). Dekorator można traktować jako uproszczony obiekt złożony obejmujący tylko jeden komponent. Jednak dekorator dodaje nowe zadania i nie jest przeznaczony dołączenia obiektów.

Strategia (s. 321). Dekorator umożliwia zmianę „skórki” obiektu, natomiast strategia służy do modyfikowania mechanizmów. Są to dwa różne sposoby zmieniania obiektów.

FASADA (FAÇADE)

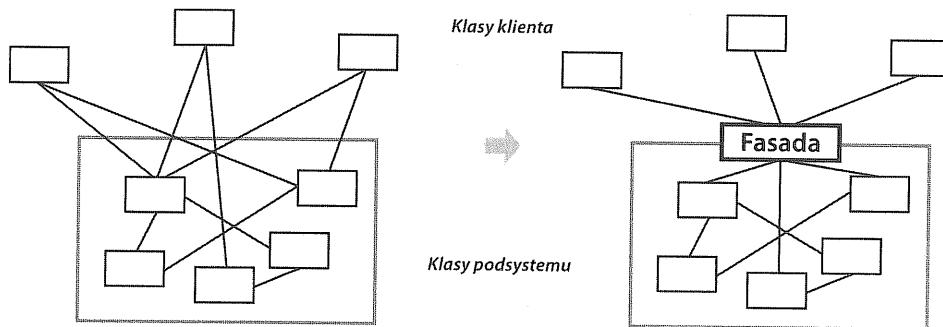
obiektowy, strukturalny

PRZEZNACZENIE

Udostępnia jednolity interfejs dla zbioru interfejsów z podsystemu. Fasada określa interfejs wyższego poziomu ułatwiający korzystanie z podsystemów.

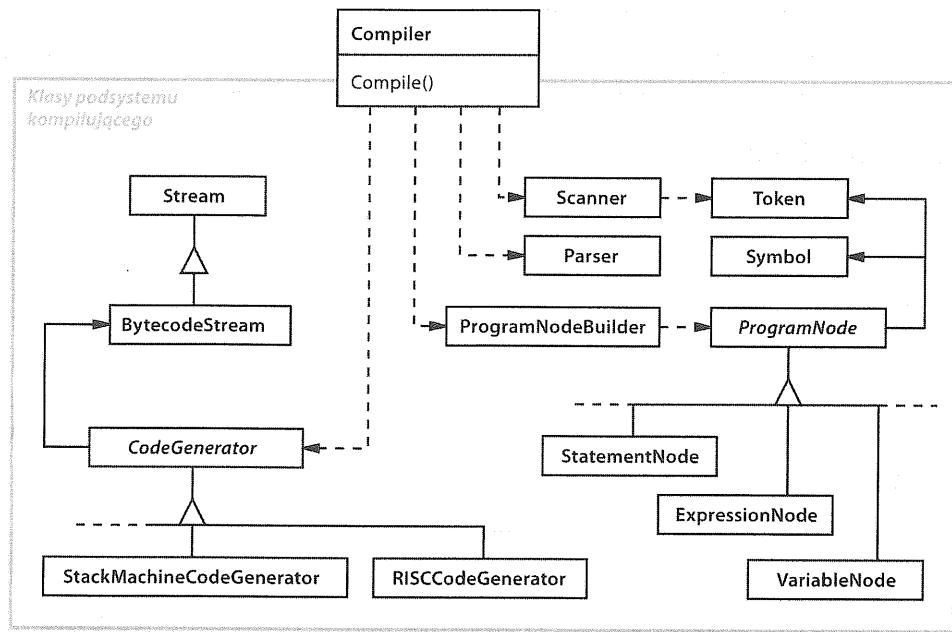
UZASADNIENIE

Podział systemu na podsystemy pomaga zmniejszyć jego złożoność. Standardowym celem projektowym jest zminimalizowanie komunikacji i zależności między podsystemami. Jednym ze sposobów na uzyskanie tego efektu jest wprowadzenie obiektu **fasadowego** udostępniającego jeden uproszczony interfejs dla ogólniejszych mechanizmów podsystemu.



Rozważmy na przykład środowisko programistyczne, które zapewnia aplikacjom dostęp do podsystemu komplilującego. Ten podsystem obejmuje takie klasy, jak Scanner, Parser, ProgramNode, BytecodeStream i ProgramNodeBuilder. Składają się one na implementację komplilatora. Niektóre wyspecjalizowane aplikacje mogą wymagać bezpośredniego dostępu do tych klas. Jednak w większości klientów szczegóły, takie jak parsowanie i generowanie kodu, nie mają znaczenia. Autorzy tych klientów chcą tylko kompilować kod. Dla nich dające duże możliwości, ale niskopoziomowe interfejsy podsystemu komplilatora jedynie komplikują zadanie.

Aby udostępnić wysokopoziomowy interfejs, który ukryje wyspecjalizowane klasy przed klientami, w podsystemie komplilującym umieszczono także klasę Compiler. Definiuje ona jednolity interfejs do funkcji komplilatora. Klasa Compiler odgrywa rolę fasady — udostępnia klientom jeden prosty interfejs do podsystemu komplilującego. W ten sposób łączy klasy obejmujące implementację funkcji komplilatora bez całkowitego ich ukrywania. Fasada komplilatora ułatwia pracę większości programistów, a przy tym nie ukrywa niskopoziomowych funkcji przed nieliczną grupą osób, które ich potrzebują.

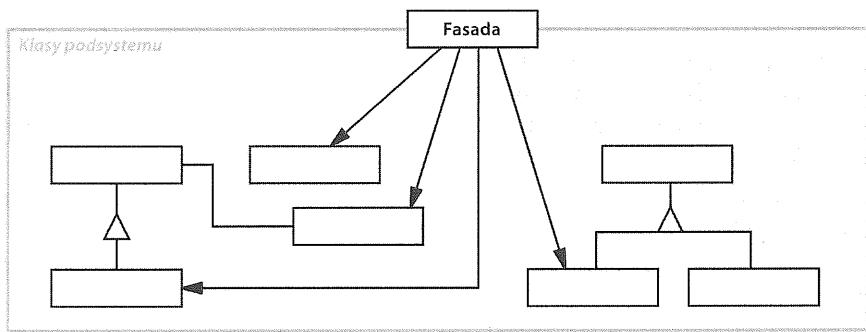


WARUNKI STOSOWANIA

Oto warunki stosowania ze wzorcą Fasada:

- ▶ Programista chce udostępnić prosty interfejs do złożonego podsystemu. Złożoność podsystemów często rośnie wraz z ich rozwijaniem. Zastosowanie wzorców prowadzi zwykle do powstania większej liczby mniejszych klas. Zwiększa to możliwość dostosowywania i powtórnego wykorzystania podsystemu, ale utrudnia stosowanie go w klientach, w których modyfikacje nie są potrzebne. Fasada pozwala udostępnić prosty domyślny interfejs do podsystemu, wystarczająco dobry dla większości klientów. Jedynie programiści potrzebujący większych możliwości dostosowywania będą musieli zajrzeć za fasadę.
- ▶ Występowanie wielu zależności między klientami i klasami z implementacją abstrakcji. Można wtedy wprowadzić fasadę, aby oddzielić podsystem od klientów i innych podsystemów, co korzystnie wpływa na niezależność podsystemu oraz możliwość jego przenoszenia między platformami.
- ▶ Programista chce podzielić podsystemy na warstwy. Fasada służy wtedy do definiowania punktu wejścia do każdego poziomu podsystemu. Jeśli podsystemy są zależne od siebie, można uprościć zależności między nimi przez wykorzystanie do komunikacji tylko ich fasad.

STRUKTURA



ELEMENTY

- ▶ **Facade (Compiler):**
 - zna klasy podsystemu odpowiadające za obsługę żądań;
 - deleguje żądania od klienta do odpowiednich obiektów podsystemu.
- ▶ **Klasy podsystemu (Scanner, Parser, ProgramNode itd.):**
 - obejmują implementację funkcji podsystemu;
 - obsługują zadania przydzielane przez obiekt Facade;
 - nie wiedzą o istnieniu fasady (nie przechowują referencji do niej).

WSPÓŁDZIAŁANIE

- ▶ Klienci komunikują się z podsystemem przez przesyłanie żądań do fasady, która przekazuje je do odpowiednich obiektów podsystemu. Choć to obiekty podsystemu wykonują zadania, fasada musi samodzielnie przekształcać żądania kierowane do jej interfejsu na polecenia z interfejsów podsystemu.
- ▶ Klienci korzystające z fasady nie muszą mieć bezpośredniego dostępu do obiektów podsystemu.

KONSEKWENCJE

Stosowanie wzorca Fasada daje następujące korzyści:

1. Oddziela klienci od komponentów podsystemu, przez co zmniejsza liczbę obiektów, których używają klienci, i ułatwia korzystanie z podsystemu.
2. Pomaga zachować luźne powiązanie między podsystemem i klientami. Komponenty w podsystemie często są mocno powiązane. Luźne powiązanie pozwala na modyfikowanie komponentów podsystemu bez wpływu na klienci. Fasady pomagają podzielić system i zależności między obiektami na warstwy. Pozwala to wyeliminować skomplikowane lub cykliczne zależności. Może to mieć duże znaczenie, jeśli implementowanie klienta i podsystemu przebiega niezależnie.

Zmniejszenie zależności kompilacyjnych jest niezwykle istotne w dużych systemach oprogramowania. Programistom zależy na przyspieszeniu pracy przez zminimalizowanie zakresu rekompilacji kodu po zmodyfikowaniu klas podsystemu. Zmniejszanie zależności kompilacyjnych za pomocą fasad pozwala ograniczyć rekompilację po niewielkich modyfikacjach w ważnym podsystemie. Fasada może też uproszczyć przenoszenie systemów na inne platformy, ponieważ zmniejsza prawdopodobieństwo, że zbudowanie jednego podsystemu będzie wymagało tego samego w przypadku wszystkich pozostałych.

3. Nie uniemożliwia aplikacjom korzystania z klas podsystemów, jeśli jest to konieczne. Dlatego można wybierać między łatwością użytkowania i ogólnością.

IMPLEMENTACJA

W czasie implementowania fasad warto rozważyć następujące zagadnienia:

1. *Zmniejszenie powiązania między klientem i podsystemem.* Powiązanie między klientami i podsystemem można jeszcze bardziej zmniejszyć przez utworzenie fasady jako klasy abstrakcyjnej i dodanie podklas konkretnych dla różnych implementacji podsystemu. Wtedy klienci mogą komunikować się z podsystemem poprzez interfejs klasy abstrakcyjnej Facade. To abstrakcyjne powiązanie sprawia, że klienci nie wiedzą, z której implementacji podsystemu korzystają.

Oprócz tworzenia podklas można też skonfigurować obiekt Facade przy użyciu różnych obiektów podsystemu. Wtedy aby dostosować fasadę, wystarczy zastąpić wybrane obiekty podsystemu innymi.

2. *Publiczne i prywatne klasy podsystemu.* Podsystem to odpowiednik klasy, ponieważ także ma interfejsy i kapsuluje pewne elementy (klasa kapsuluje stan i operacje, natomiast podsystem — klasy). Dlatego interfejs podsystemu można — tak samo jak w klasach — podzielić na publiczny i prywatny.

Publiczny interfejs podsystemu składa się z klas dostępnych dla wszystkich klientów. Interfejs prywatny jest przeznaczony tylko dla osób rozszerzających podsystem. Klasa Facade to oczywiście część interfejsu publicznego, jednak obejmuje on także inne elementy. Publiczne są też zwykle inne klasy podsystemu, takie jak Parser i Scanner z podsystemu kompilującego.

Tworzenie prywatnych klas podsystemu byłoby przydatne, jednak nieliczne języki obiektowe to umożliwiają. W przeszłości zarówno C++, jak i Smalltalk miały globalną przestrzeń nazw klas. Jednak niedawno komitet standaryzacyjny języka C++ dodał do niego przestrzenie nazw [Str94], co umożliwia udostępnianie tylko publicznych klas podsystemu.

PRZYKŁADOWY KOD

Przyjrzyjmy się dokładniej temu, jak dodać fasadę do podsystemu kompilującego.

Podsystem kompilujący obejmuje klasę `BytecodeStream` z implementacją strumienia obiektów `Bytecode`. Obiekty `Bytecode` kapsułkują kod bajtowy, który może określać instrukcje maszynowe. Podsystem obejmuje też klasę `Token`. Reprezentuje ona obiekty kapsułkujące znaczniki w językach programowania.

Klasa Scanner przyjmuje strumień znaków i tworzy strumień znaczników (po jednym naraz).

```
class Scanner {
public:
    Scanner(istream&);
    virtual ~Scanner();

    virtual Token& Scan();
private:
    istream& _inputStream;
};
```

W klasie Parser wykorzystaliśmy obiekt ProgramNodeBuilder do utworzenia drzewa składni na podstawie znaczników z obiektu Scanner.

```
class Parser {
public:
    Parser();
    virtual ~Parser();

    virtual void Parse(Scanner&, ProgramNodeBuilder&);
};
```

Obiekt Parser zwrotnie wywołuje obiekt ProgramNodeBuilder w celu stopniowego utworzenia drzewa składni. Interakcja między tymi klasami odbywa się zgodnie ze wzorcem Budowniczy (s. 92).

```
class ProgramNodeBuilder {
public:
    ProgramNodeBuilder();

    virtual ProgramNode* NewVariable(
        const char* variableName
    ) const;

    virtual ProgramNode* NewAssignment(
        ProgramNode* variable, ProgramNode* expression
    ) const;

    virtual ProgramNode* NewReturnStatement(
        ProgramNode* value
    ) const;

    virtual ProgramNode* NewCondition(
        ProgramNode* condition,
        ProgramNode* truePart, ProgramNode* falsePart
    ) const;
    // ...

    ProgramNode* Get rootNode();
private:
    ProgramNode* _node;
};
```

Drzewo składni zbudowane jest z egzemplarzy podklas klasy `ProgramNode`. Te podklasy to na przykład `StatementNode`, `ExpressionNode` itd. Hierarchia klasy `ProgramNode` to przykład zastosowania wzorca Kompozyt (s. 170). Klasa ta definiuje interfejs do manipulowania obiektem `ProgramNode` i jego elementami podrzędnymi (jeśli takie istnieją).

```
class ProgramNode {
public:
    // Manipulowanie obiektem ProgramNode.
    virtual void GetSourcePosition(int& line, int& index);
    // ...

    // Manipulowanie elementami podrzędnymi.
    virtual void Add(ProgramNode* );
    virtual void Remove(ProgramNode* );
    // ...

    virtual void Traverse(CodeGen& );
protected:
    ProgramNode();
};

Operacja Traverse przyjmuje obiekt CodeGenerator. Podklasy klasy ProgramNode wykorzystują ten obiekt do generowania kodu maszynowego w postaci obiektów Bytecode w obiekcie BytecodeStream. Klasa CodeGenerator pełni tu rolę odwiedzającego (zobacz wzorzec Odwiedzający, s. 280).
```

```
class CodeGenerator {
public:
    virtual void Visit(StatementNode* );
    virtual void Visit(ExpressionNode* );
    // ...

protected:
    CodeGenerator(BytecodeStream& );
protected:
    BytecodeStream& _output;
};

Klasa CodeGenerator ma podklasy (na przykład StackMachineCodeGenerator i RISCCodeGenerator) generujące kod maszynowy dla różnych architektur sprzętowych.
```

W każdej podklasie klasy `ProgramNode` operacja `Traverse` jest zaimplementowana tak, aby wywoływała operację `Traverse` na podrzędnych obiektach `ProgramNode` danej podklasy. Z kolei elementy podrzędne robią to samo w stosunku do swoich elementów podrzędnych i tak dalej w rekurencyjny sposób. Na przykład w podklasie `ExpressionNode` definicja operacji `Traverse` wygląda tak:

```
void ExpressionNode::Traverse (CodeGen& cg) {
    cg.Visit(this);

    ListIterator<ProgramNode*> i(_children);

    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Traverse(cg);
    }
}
```

Klasy opisane do tej pory składają się na podsystem komplilujący. Teraz wprowadzimy klasę `Compiler`. Jest to fasada, która łączy wszystkie pozostałe elementy. Klasa ta udostępnia prosty interfejs do komplilowania kodu źródłowego i generowania kodu dla określonej maszyny.

```
class Compiler {
public:
    Compiler();

    virtual void Compile(istream&, BytecodeStream&);
};

void Compiler::Compile (
    istream& input, BytecodeStream& output
) {
    Scanner scanner(input);
    ProgramNodeBuilder builder;
    Parser parser;

    parser.Parse(scanner, builder);

    RISCCodeGenerator generator(output);
    ProgramNode* parseTree = builder.GetRootNode();
    parseTree->Traverse(generator);
}
```

W tej implementacji typ generatora kodu jest zapisany na stałe, dlatego programiści nie muszą określać docelowej architektury. Może to być uzasadnione, jeśli docelowa architektura jest tylko jedna. Jeżeli jest inaczej, możemy zdecydować się na zmodyfikowanie konstruktora klasy `Compiler`, tak aby przyjmował parametr typu `CodeGenerator`. Następnie programiści będą mogli wskazać potrzebny generator w czasie tworzenia egzemplarza klasy `Compiler`. W fasadzie kompilatora można sparametryzować też inne elementy, na przykład klasy `Scanner` i `ProgramNodeBuilder`, co zwiększa elastyczność, ale jest niezgodne z celem stosowania wzorca Fasada, czyli upraszczaniem interfejsu pod kątem wykonywania standardowych zadań.

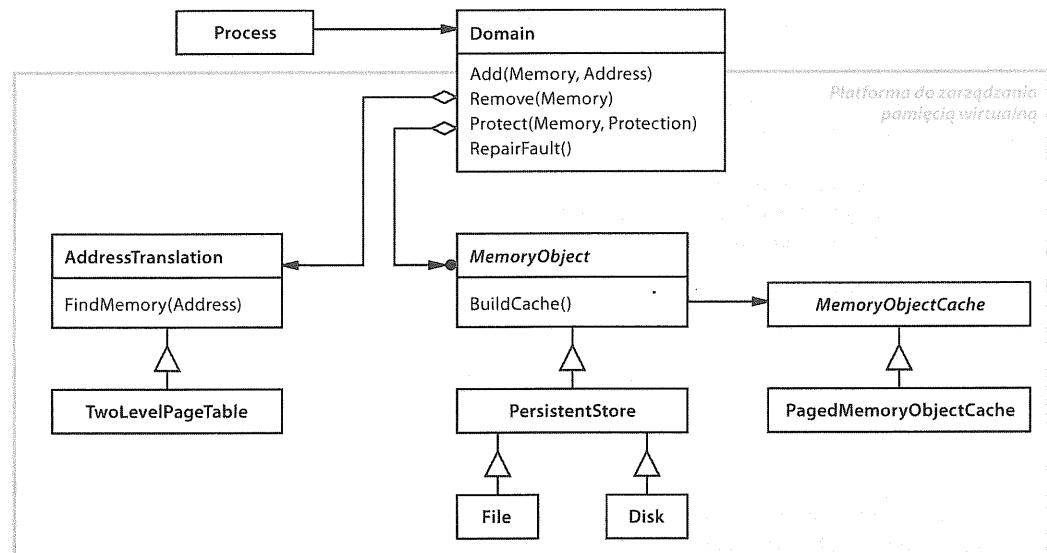
ZNANE ZASTOSOWANIA

Przykład dotyczący kompilatora opisany w punkcie „Przykładowy kod” oparliśmy na systemie komplilującym z języka ObjectWorks\Smalltalk [Par90].

Z pomocą platformy ET++ [WGM88] można utworzyć aplikację z wbudowaną przeglądarką do analizy obiektów w czasie wykonywania programu. Ta przeglądarka jest zaimplementowana w odrębnym podsystemie obejmującym odpowiednik klasy `Facade` — klasę `ProgrammingEnvironment`. Zdefiniowane są w niej między innymi operacje `InspectObject` i `InspectClass` umożliwiające dostęp do przeglądarki.

W aplikacji opartej na platformie ET++ można też zrezygnować z wbudowanej obsługi przeglądania. Wtedy w klasie `ProgrammingEnvironment` obsługa żądań jest zaimplementowana jako puste operacje (nie wykonują one żadnych działań). Tylko podklasa `ETProgrammingEnvironment` obejmuje implementacje obsługi żądań w postaci operacji wyświetlających odpowiednie przeglądarki. Aplikacja nie wie, czy środowisko przeglądarki jest dostępne czy nie. Między aplikacją i podsystemem przeglądarki występuje tu powiązanie abstrakcyjne.

W systemie operacyjnym Choices [CIRM93] fasady służą do składania wielu platform w jedną. Kluczowe abstrakcje w tym systemie opisują procesy, pamięć i przestrzeń adresowe. Dla każdej z tych abstrakcji istnieje odpowiedni podsystem zaimplementowany w postaci platformy. Podsystemy te umożliwiają przenoszenie systemu Choices na różne platformy sprzętowe. Dwa spośród wspomnianych podsystemów mają „reprezentanta” (czyli fasadę). Te fasady to klasy `FileSystemInterface` (dla pamięci) i `Domain` (dla przestrzeni adresowych).



Przykładowo fasadą platformy do zarządzania pamięcią wirtualną jest klasa `Domain`. Reprezentuje ona przestrzeń adresową i udostępnia odwzorowanie między adresami wirtualnymi oraz pozycjami w obiektach pamięci, plikach i pamięci rezerwowej. Główne operacje klasy `Domain` umożliwiają dodawanie obiektu pamięci pod określonym adresem, usuwanie takiego obiektu i obsługę błędów związanych ze stronami.

Jak ilustruje to wcześniejszy diagram, w podsystemie pamięci wirtualnej wykorzystano następujące komponenty:

- ▶ klasę `MemoryObject` reprezentującą pamięć na dane;
- ▶ klasę `MemoryObjectCache` przechowującą dane obiektu `MemoryObjects` w pamięci fizycznej (klasa `MemoryObjectCache` jest strategią (s. 321) określającą politykę obsługi pamięci podrzcznej);
- ▶ klasę `AddressTranslation` kapsułkującą sprzęt odpowiedzialny za translację adresów.

Operacja `RepairFault` jest wywoływaną przy każdym przerwaniu spowodowanym błędem związanym ze stroną. Obiekt `Domain` wyszukuje obiekt pamięci zapisany pod adresem, który wywołał błąd, i deleguje operację `RepairFault` do pamięci podrzcznej powiązanej z określonym obiektem pamięci. Obiekty `Domain` można dostosowywać do potrzeb przez modyfikowanie ich komponentów.

POWIĄZANE WZORCE

Wraz ze wzorcem Fasada można używać wzorca Fabryka abstrakcyjna (s. 101), aby udostępnić interfejs do tworzenia obiektów podsystemu niezależnie od podsystemów. Fabrykę abstrakcyjną można też zastosować zamiast fasady do ukrycia klas specyficznych dla platformy.

Wzorzec Mediator (s. 254) przypomina wzorzec Fasada, ponieważ też umożliwia abstrakcyjne ujęcie funkcji istniejących klas. Jednak celem stosowania wzorca Mediator jest wyabstrahowanie dowolnej komunikacji między współpracującymi obiektami, często związane ze scentralizowaniem funkcji, które nie należą do żadnego z takich obiektów. Elementy współpracujące z mediatorem wykrywają jego istnienie i komunikują się z nim zamiast bezpośrednio ze sobą. Zupełnie inaczej działa fasada, która służy do abstrakcyjnego ujęcia interfejsu do obiektów podsystemu w celu ułatwienia korzystania z nich. Fasada nie definiuje nowych funkcji, a klasy podsystemu nie wiedzą o jej istnieniu.

Zwykle potrzebny jest tylko jeden obiekt fasady. Dlatego obiekty tego rodzaju często są singletonami (s. 130).

KOMPOZYT (COMPOSITE)

obiektowy, strukturalny

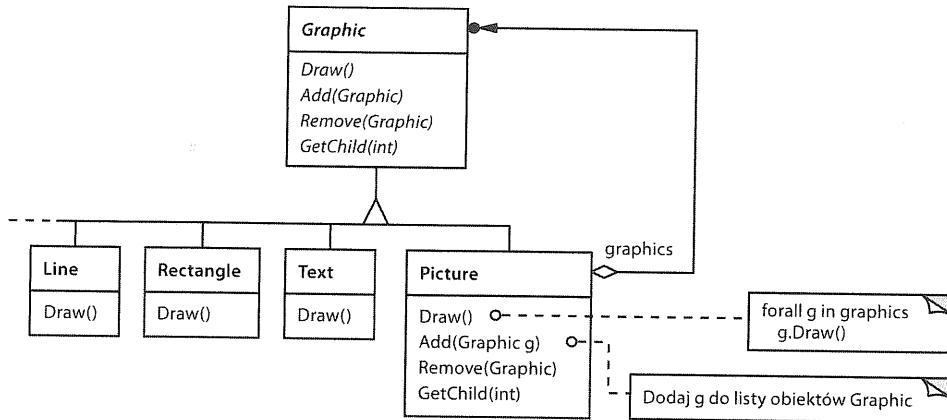
PRZEZNACZENIE

Składa obiekty w strukturę drzewiastą odzwierciedlającą hierarchię typu część-całość. Wzorzec ten umożliwia klientom traktowanie poszczególnych obiektów i ich złożień w taki sam sposób.

UZASADNIENIE

Aplikacje graficzne, na przykład edytory graficzne i edytory schematów, umożliwiają użytkownikom budowanie złożonych diagramów z prostych komponentów. Użytkownik może połączyć komponenty w większe jednostki, a te — w jeszcze bardziej rozbudowane komponenty. W podstawowej implementacji można zdefiniować klasy prostych elementów graficznych, takie jak `Text` i `Line`, oraz inne klasy pełniące funkcję kontenerów dla owych prostych jednostek.

Jednak podejście to ma pewną wadę. W kodzie obiekty proste i kontenerowe trzeba traktować w odmienny sposób, choć użytkownicy przeważnie korzystają z nich tak samo. Konieczność rozróżniania rodzajów obiektów sprawia, że aplikacja staje się bardziej skomplikowana. Wzorzec Kompozyt opisuje, jak zastosować składanie rekurencyjne, aby w kodzie klientów nie trzeba było uwzględniać wspomnianego podziału.

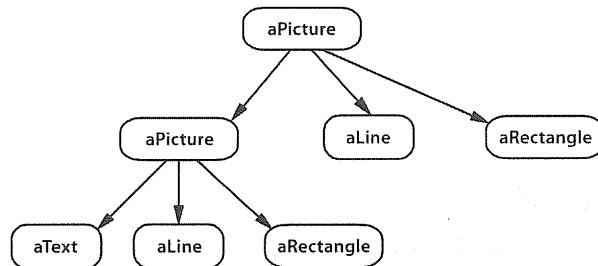


Kluczowym elementem wzorca Kompozyt jest klasa abstrakcyjna reprezentująca zarówno typy proste, jak i zawierające je kontenery. W systemie graficznym może to być klasa `Graphic`. Należy zadeklarować w niej operacje specyficzne dla obiektów graficznych, takie jak `Draw`, a także operacje wspólne wszystkim obiektom złożonym, służące na przykład do udostępniania elementów podrzędnych i zarządzania nimi.

Podklasy `Line`, `Rectangle` i `Text` (zobacz wcześniejszy diagram) definiują proste obiekty graficzne. Klasy te obejmują implementacje operacji `Draw` wyświetlające linie, prostokąty i tekst. Ponieważ proste obiekty graficzne nie mają graficznych elementów podrzędnych, żadna z wymienionych podklas nie zawiera implementacji operacji związanych z takimi elementami.

Klasa Picture definiuje agregat obiektów klasy Graphic. Operacja Draw w klasie Picture jest zaimplementowana tak, aby wywoływała operację Draw elementów podległych. Klasa ta obejmuje też implementację operacji związanych z takimi elementami. Ponieważ interfejs klasy Picture jest zgodny z interfejsem klasy Graphic, obiekty Picture można rekurencyjnie składać z innymi takimi obiektami.

Poniższy diagram przedstawia typową strukturę obiektu złożonego na przykładzie rekurencyjnie złożonych obiektów Graphic.

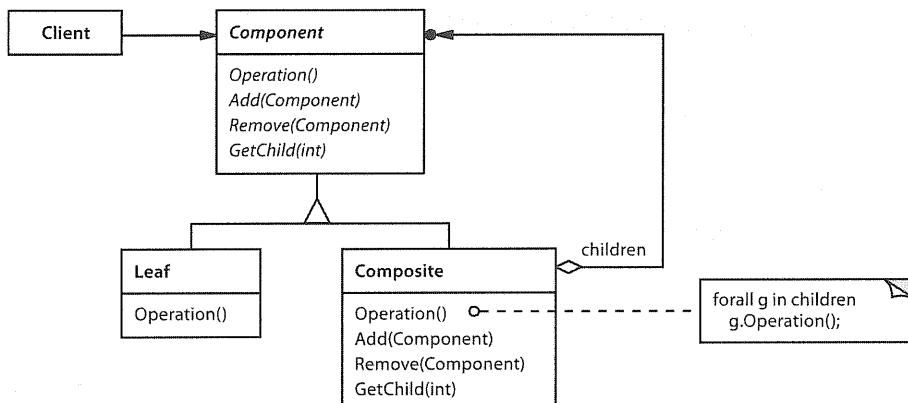


WARUNKI STOSOWANIA

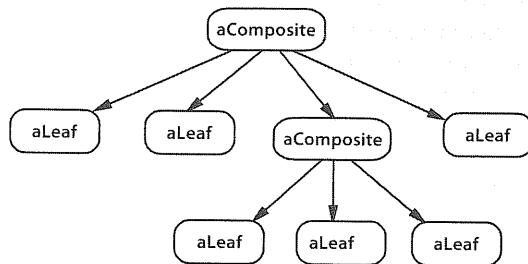
Wzorzec Kompozyt należy stosować w poniższych warunkach:

- ▶ Jeżeli chcesz przedstawić hierarchie obiektów typu część-całość.
- ▶ Kiedy chcesz, aby w klientach można było zignorować różnice między złożeniami obiektów i pojedynczymi obiektami. Klienci będą wtedy traktować wszystkie obiekty ze struktury w taki sam sposób.

STRUKTURA



Struktura typowego obiektu Composite może wyglądać tak:



ELEMENTY

► **Component** (Graphic):

- obejmuje deklarację interfejsu składanych obiektów;
- obejmuje implementację domyślnego zachowania na potrzeby interfejsu wspólnego dla wszystkich klas (dotyczy to odpowiednich operacji);
- obejmuje deklarację interfejsu umożliwiającego dostęp do komponentów podrzędnych i zarządzanie nimi;
- opcjonalnie definiuje interfejs umożliwiający dostęp do elementu nadzawanego danego komponentu w strukturze rekurencyjnej i — jeśli to potrzebne — implementację tego interfejsu.

► **Leaf** (Rectangle, Line, Text itd.), czyli liść:

- reprezentuje liście w złożeniu (liść to obiekt bez elementów podrzędnych);
- definiuje zachowanie obiektów prostych w złożeniu.

► **Composite** (Picture):

- definiuje zachowanie komponentów mających elementy podrzędne;
- przechowuje komponenty podrzędne;
- obejmuje implementację operacji z interfejsu klasy Component związanych z elementami podrzędnymi.

► **Client**:

- manipuluje obiektami w złożeniu poprzez interfejs klasy Component.

WSPÓŁDZIAŁANIE

- Klienci korzystają z interfejsu klasy Component do wchodzenia w interakcje z obiektami ze struktury złożonej. Jeśli żądanie zostanie odebrane przez obiekt Leaf, będzie obsłużone bezpośrednio. Jeżeli odbiorcą jest obiekt Composite, zwykle przekazuje on żądanie do komponentów podrzędnych, przy czym wcześniej i (lub) później może wykonać dodatkowe operacje.

KONSEKWENCJE

Oto cechy wzorca Kompozyt:

- ▶ Umożliwia definiowanie hierarchii składających się z obiektów prostych i złożonych. Obiekty proste można składać w złożone, które także można łączyć ze sobą itd. W kodzie klienta wszędzie tam, gdzie oczekiwany jest obiekt prosty, można podać także obiekt złożony.
- ▶ Upraszcza kod klientów. Klienci mogą traktować struktury złożone i poszczególne obiekty w taki sam sposób. Zwykle klienci nie potrafią określić, czy korzystają z liścia czy z komponentu złożonego (i nie powinno to mieć znaczenia). Upraszcza to kod klientów, ponieważ nie trzeba pisać skomplikowanych funkcji sprawdzających klasy definiujące złożenia.
- ▶ Ułatwia dodawanie komponentów nowego rodzaju. Nowo zdefiniowane podklasy klas Composite lub Leaf automatycznie współdziałają z istniejącymi strukturami i kodem klienta. Nie trzeba modyfikować klientów, aby dostosować je do nowych klas z rodziny Component.
- ▶ Mogą sprawić, że projekt stanie się zanadto ogólny. Wadą możliwości łatwego dodawania nowych komponentów jest to, że trudniej jest wprowadzić ograniczenia co do komponentów, które można dodać do kompozytu. Czasem programista chce, aby kompozyt składał się tylko z komponentów określonego rodzaju. Przy stosowaniu wzorca Kompozyt nie można zakładać, że system typów automatycznie wymusi takie ograniczenia. Trzeba wtedy sprawdzać je w czasie wykonywania programu.

IMPLEMENTACJA

W czasie implementowania wzorca Kompozyt trzeba uwzględnić wiele zagadnień.

1. *Jawne referencje do elementu nadzawanego.* Przechowywanie w komponentach podrzędnych referencji do elementu nadzawanego pozwala uprościć przechodzenie po strukturze złożonej i zarządzanie nią. Taka referencja ułatwia przechodzenie w górę struktury i usuwanie komponentu. Referencje do elementu nadzawanego pomagają też zastosować wzorzec Łącuch zobowiązań (s. 244).

Standardowym miejscem na zdefiniowanie referencji do elementu nadzawanego jest klasa Component. Klasy Leaf i Composite mogą dziedziczyć tę referencję oraz operacje potrzebne do zarządzania nią.

Przy korzystaniu z referencji do elementu nadzawanego konieczne jest stosowanie niezmiennika, zgodnie z którym dla wszystkich elementów podrzędnych w kompozycie elementem nadzawanym jest ten kompozyt i na odwrót. Najprościej jest to zagwarantować przez zmianę elementu nadzawanego komponentu *tylko* po dodaniu lub usunięciu komponentu kompozytu. Jeśli możliwe jest zaimplementowanie tego rozwiązania w jednym miejscu (za pomocą operacji Add i Remove klasy Composite), mogą je odziedziczyć wszystkie podklasy, a niezmiennik będzie przestrzegany automatycznie.

2. *Współużytkowanie komponentów.* Często przydatne jest współużytkowanie komponentów, na przykład w celu zmniejszenia wymogów związanych z pamięcią. Jednak jeśli komponent nie może mieć więcej niż jednego elementu nadzawanego, współużytkowanie staje się trudne.

Możliwe rozwiązywanie polega na przechowywaniu w elementach podrzędnych informacji o wielu elementach nadrzędnych. Prowadzi to jednak do wieloznaczności przy przekazywaniu żądania w strukturze. Wzorzec Pyłek (s. 201) pokazuje, jak zmodyfikować projekt, tak aby całkowicie wyeliminować przechowywanie informacji o elementach nadrzędnych. To podejście można zastosować, jeśli w elementach podrzędnych możliwe jest uniknięcie wysyłania żądań do elementów nadrzędnych (wymaga to zapisania części lub całości stanu elementów podrzędnych poza nimi).

3. *Maksymalizowanie interfejsu klasy Component.* Jednym z celów stosowania wzorca Kompozyt jest ukrycie przed klientami poszczególnych używanych przez nie klas Leaf lub Composite. Aby to osiągnąć, w klasie Component należy zdefiniować jak najwięcej wspólnych operacji klas Composite i Leaf. Klasa Component zwykle obejmuje implementacje domyślne tych operacji, a w podklasach Leaf i Composite są one przesłonięte.

Jednak wspomniany cel bywa sprzeczny z zasadą projektowania hierarchii klas, zgodnie z którą w klasie należy zdefiniować tylko operacje sensowne w jej podklasach. Klasa Component obsługuje wiele operacji, które nie są przydatne w klasach Leaf. Jak utworzyć domyślną implementację tych operacji w klasie Component?

Czasem wystarczy niewielka doza pomysłowości, aby zrozumieć, w jaki sposób operację sensowną na pozór tylko dla klas Composite można zaimplementować dla wszystkich klas z rodziny Component i przenieść ją do klasy Component. Na przykład interfejs umożliwiający dostęp do elementów podrzędnych jest istotnym składnikiem klasy Composite, ale już niekoniecznie klas Leaf. Jednak jeśli potraktujemy obiekt Leaf jak obiekt Component, który nigdy nie ma elementów podrzędnych, możemy zdefiniować w klasie Component operację domyślną, która nigdy nie zwraca takich elementów. W klasach Leaf można wykorzystać tę domyślną implementację, a w klasach Composite ponownie ją zaimplementować, tak aby zwracała elementy podrzędne.

Operacje do zarządzania elementami podrzędnymi sprawiają więcej trudności i omawiamy je w następnym punkcie.

4. *Deklarowanie operacji do zarządzania elementami podrzędnymi.* Choć klasa Composite obejmuje implementację operacji Add i Remove przeznaczonych do zarządzania elementami podrzędnymi, istotnym problemem związanym ze wzorcem Kompozyt jest to, w których klasach hierarchii klasy Composite operacje te są zadeklarowane. Czy należy zadeklarować je w klasie Component i sprawić, że będą sensowne w klasach Leaf, czy może lepiej będzie zadeklarować i zdefiniować je tylko w klasie Composite i jej podklasach?

Ta decyzja wymaga zachowania równowagi między bezpieczeństwem i przezroczystością.

- Zdefiniowanie interfejsu do zarządzania elementami podrzędnymi w elemencie głównym hierarchii klas zapewnia przezroczystość, ponieważ można traktować wszystkie komponenty w taki sam sposób. Dzieje się to jednak kosztem bezpieczeństwa, ponieważ klienci mogą próbować wykonywać bezsensowne operacje, takie jak dodawanie i usuwanie obiektów w liściach.
- Zdefiniowanie interfejsu do zarządzania elementami podrzędnymi w klasie Composite zapewnia bezpieczeństwo, ponieważ w języku ze statyczną kontrolą typów, takim jak C++, próba dodania lub usunięcia obiektów w liściach zostanie wykryta w czasie komplikacji. Powoduje to jednak utratę przezroczystości, ponieważ liście i kompozyty będą miały różne interfejsy.

W tym wzorcu położyliśmy nacisk na przezroczystość kosztem bezpieczeństwa. Jeśli bardziej zależy Ci na bezpieczeństwie, pamiętaj, że czasem możesz utracić informacje o typie i będziesz musiał przekształcić komponent na kompozyt. Jak można to zrobić bez uciekania się do rzutowania niebezpiecznego ze względu na typ?

Jednym z rozwiązań jest zadeklarowanie operacji `Composite* GetComposite()` w klasie `Component`. Klasa ta udostępnia domyślną operację zwracającą wskaźnik pusty. W klasie `Composite` operacja ta jest przedefiniowana i zwraca obiekt tej klasy poprzez wskaźnik `this`.

```
class Composite;

class Component {
public:
    // ...
    virtual Composite* GetComposite() { return 0; }
};

class Composite : public Component {
public:
    void Add(Component*);
    // ...
    virtual Composite* GetComposite() { return this; }
};

class Leaf : public Component {
    // ...
};
```

Operacja `GetComposite` umożliwia skierowanie zapytania do komponentu w celu sprawdzenia, czy jest kompozytem. Następnie można bezpiecznie wykonać operacje `Add` i `Remove` na zwróconym kompozycie.

```
Composite* aComposite = new Composite;
Leaf* aLeaf = new Leaf;

Component* aComponent;
Composite* test;

aComponent = aComposite;
if (test = aComponent->GetComposite()) {
    test->Add(new Leaf);
}

aComponent = aLeaf;

if (test = aComponent->GetComposite()) {
    test->Add(new Leaf); // Jeśli obiekt to liść, operacja Add nie zostanie wywołana.
}
```

Podobne testy pod kątem kompozytów można przeprowadzić za pomocą konstrukcji `dynamic_cast` języka C++.

Wada tego rozwiązania polega oczywiście na tym, że nie traktujemy wszystkich komponentów w ten sam sposób. Musimy uciekać się do sprawdzania różnych typów przyjęciem odpowiednich działań.

Jedyny sposób na zapewnienie przezroczystości polega na zdefiniowaniu domyślny operacji Add i Remove w klasie Component. Stwarza to jednak nowy problem — nie ma możliwości zaimplementowanie operacji Component::Add tak, aby jej wywołanie zawsze zakończyło się powodzeniem. Moglibyśmy nie umieszczać w niej żadnych instrukcji, co oznacza to zignorowanie ważnej kwestii, a mianowicie tego, że próba dodania elementu do liścia prawdopodobnie jest wynikiem błędu. Wtedy operacja Add spowoduje zaśmiecić pamięć. Możemy spowodować, aby usuwała otrzymany argument, jednak klient może oczekwać czegoś innego.

Zwykle lepiej jest sprawić, aby wywołanie operacji Add i Remove domyślnie kończyło się niepowodzeniem (na przykład przez zgłoszenie wyjątku), jeśli komponent nie może mieć elementów podległych lub argumentem operacji Remove nie jest element podległy danego komponentu.

Inna możliwość to niewielkie zmodyfikowanie znaczenia operacji Remove. Jeśli komponent przechowuje referencję do elementu nadzawanego, można przedefiniować operację Component::Remove w taki sposób, aby usuwała dany komponent z elementu nadzawanego. Nie można jednak w podobny sposób zmodyfikować operacji Add.

5. *Czy w klasie Component należy zaimplementować obsługę listy obiektów tego typu?* Możliwe, że kusi Cię myśl o zdefiniowaniu zbioru elementów podległych jako zmiennej egzemplarz klasy Component, gdzie zadeklarowane są operacje umożliwiające dostęp do takich elementów i zarządzania nimi. Jednak umieszczenie wskaźnika do elementu podległego w klasie bazowej powoduje większe zużycie pamięci przez każdy liść, nawet jeśli nie może być on elementów podległych. Takie rozwiązanie jest opłacalne tylko wtedy, jeśli w strukturze znajduje się stosunkowo niewiele elementów podległych.
 6. *Uporządkowanie elementów podległych.* Wiele projektów wyznacza uporządkowanie elementów podległych w klasie Composite. We wcześniejszym przykładzie dotyczącym klasy Graphics uporządkowanie może odzwierciedlać kolejność obiektów od przodu do tyłu. Jeśli obiekty Composite reprezentują drzewa składni, instrukcje złożone mogą być egzemplarzami klasy Composite, w których elementy podległe trzeba uporządkować zgodnie ze strukturą programu.
- Jeśli uporządkowanie elementów podległych jest istotne, trzeba starannie zaprojektować interfejsy umożliwiające dostęp do takich elementów i zarządzanie nimi, aby móc kontrolować ich kolejność. Pomóc w tym może wzorzec Iterator (s. 230).
7. *Korzystanie z pamięci podrzędnej w celu poprawy wydajności.* Jeśli trzeba często przechodzić po złożeniach lub je przeszukiwać, można zapisać w pamięci podrzędnej w klasie Composite informacje o jej elementach podległych związane z tymi operacjami. Klasa Composite może rejestrować wyniki tych działań lub jedynie dane pozwalające skrócić przechodzenie lub wyszukiwanie. Na przykład w klasie Picture z przykładu z punktu „Uzasadnienie” można zapisać ramkę ograniczającą na elementy podległe obiektu tej klasy. Zarejestrowana ramka pozwoli pominąć obiektowi Picture operacje wyświetlania lub wyszukiwania, jeśli jego elementy podległe nie są widoczne w aktywnym oknie.

Zmiany wprowadzane w komponencie będą wymagały unieważnienia pamięci podręcznej jego elementów nadrzędnych. To podejście jest najbardziej skuteczne, jeśli komponent zna swoje elementy nadrzędne. Dlatego jeśli korzystasz z pamięci podręcznej, musisz zdefiniować interfejs do informowania kompozytów o tym, że pamięć podręczna zawiera nieaktualne informacje.

8. *W której klasie należy usuwać komponenty?* W językach bez automatycznego przywracania pamięci zwykle najlepiej jest sprawić, aby to obiekt Composite usuwał swoje elementy podrzędne w momencie, kiedy sam jest usuwany. Wyjątek od tej reguły ma miejsce wtedy, kiedy obiekt Leaf jest niezmienny, co powoduje, że można go współużytkować.
9. *Która struktura danych najlepiej nadaje się do przechowywania komponentów?* W kompozytach można przechowywać elementy podrzędne za pomocą różnych struktur danych, w tym list powiązanych, drzew, tablic i tablic haszujących. Wybór struktury zależy (jak zawsze) od wydajności. Tak naprawdę w ogóle nie trzeba korzystać ze struktur danych przeznaczonych do ogólnego użytku. Czasem w kompozytach można umieścić zmienną dla każdego elementu podrzędnego, choć wymaga to zaimplementowania w każdej podklasie klasy Composite odrębnego interfejsu do zarządzania takimi elementami. Zobacz na przykład wzorzec Interpreter (s. 217).

PRZYKŁADOWY KOD

Sprzęt, taki jak komputery i zestawy stereo, jest często uporządkowany według hierarchii część-całość lub zawierania. Płyta montażowa może na przykład obejmować dyski i płyty główne, magistrala — karty, a obudowa — płyty montażowe, magistrale itd. Za pomocą wzorca Kompozyt można w naturalny sposób tworzyć modele takich struktur.

Klasa Equipment definiuje interfejs dla wszystkich urządzeń z hierarchii część-całość.

```
class Equipment {
public:
    virtual ~Equipment();

    const char* Name() { return _name; }

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();

    virtual void Add(Equipment*);
    virtual void Remove(Equipment*);
    virtual Iterator<Equipment*>* CreateIterator();

protected:
    Equipment(const char*);

private:
    const char* _name;
};
```

W klasie Equipment zadeklarowaliśmy operacje zwracające cechy urządzeń, na przykład połówki mocy i cenę. W podkласach operacje te są zaimplementowane pod kątem poszczególnych rodzajów sprzętu. Klasa Equipment obejmuje także deklarację operacji CreateIterator.

Zwraca ona obiekt `Iterator` (zobacz dodatek C) zapewniający dostęp do części obiektu. Domyślna implementacja tej operacji zwraca obiekt `NullIterator` przechodzący po pustym zbiorze.

Do podklas klasy `Equipment` mogą należeć klasy liści reprezentujące dyski twardy, układy scalone i przełączniki.

```
class FloppyDisk : public Equipment {
public:
    FloppyDisk(const char*);
    virtual ~FloppyDisk();

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();
};
```

`CompositeEquipment` to klasa bazowa urządzeń składających się z innych sprzętów. Jest ona jednocześnie podkąską klasy `Equipment`.

```
class CompositeEquipment : public Equipment {
public:
    virtual ~CompositeEquipment();

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();

    virtual void Add(Equipment*);
    virtual void Remove(Equipment*);
    virtual Iterator<Equipment*>* CreateIterator();
protected:
    CompositeEquipment(const char*);
private:
    List<Equipment*> _equipment;
};
```

Klasa `CompositeEquipment` definiuje operacje umożliwiające dostęp do składników sprzętu i zarządzanie nimi. Operacje `Add` i `Remove` wstawiają oraz usuwają urządzenia na liście urządzeń przechowywanej w składowej `_equipment`. Operacja `CreateIterator` zwraca iterator (a dokładniej —egzemplarz klasy `ListIterator`) służący do przechodzenia po tej liście.

W domyślnej implementacji operacji `NetPrice` można wykorzystać obiekt `CreateIterator` do zsumowania cen składników sprzętu².

```
Currency CompositeEquipment::NetPrice () {
    Iterator<Equipment*>* i = CreateIterator();
    Currency total = 0;
```

² Łatwo jest zapomnieć o usunięciu iteratora po zakończeniu korzystania z niego. Wzorzec `Iterator` pokazuje, jak zabezpieczyć się przed takimi błędami (s. 266).

```

    for (i->First(); !i->IsDone(); i->Next()) {
        total += i->CurrentItem()->NetPrice();
    }
    delete i;
    return total;
}

```

Teraz możemy przedstawić płytę montażową komputera jako podklasę Chassis klasy Composite \hookrightarrow Equipment. Klasa Chassis dziedziczy po klasie CompositeEquipment operacje związane z elementami podrzędnymi.

```

class Chassis : public CompositeEquipment {
public:
    Chassis(const char*);
    virtual ~Chassis();

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();
};

```

W podobny sposób możemy zdefiniować inne kontenery, na przykład klasy Cabinet i Bus. W ten sposób uzyskamy wszystkie elementy potrzebne do złożenia urządzeń w (dość prosty) komputer PC.

```

Cabinet* cabinet = new Cabinet("Obudowa komputera PC");
Chassis* chassis = new Chassis("Płyta montażowa komputera PC");

cabinet->Add(chassis);

Bus* bus = new Bus("Magistrala MCA");
bus->Add(new Card("Karta 16Mbs Token Ring"));

chassis->Add(bus);
chassis->Add(new FloppyDisk("Stacja dyskietek 3,5 cala"));

cout << "Cena netto wynosi" << chassis->NetPrice() << endl;

```

ZNANE ZASTOSOWANIA

Przykłady zastosowania wzorca Kompozyt można znaleźć w niemal każdym systemie obiektowym. Pierwotna wersja klasy View z architektury MVC języka Smalltalk [KP88] była odpowiednikiem klasy Composite, a rozwiążanie to wykorzystano w prawie każdym pakiecie narzędziowym lub platformie do tworzenia interfejsów użytkownika, w tym w ET++ (klasa VObject [WGM88]) i InterViews (klasy Style [LCI-92], Graphic [VL88] i Glyph [CL90]). Warto zauważyć, że pierwotna wersja klasy View z architektury MVC obejmowała zestaw widoków podrzędnego. Oznacza to, że pełniła jednocześnie funkcje klas Component i Composite. W wersji 4.0 języka Smalltalk-80 usprawniono architekturę MVC przez dodanie klasy VisualComponent oraz jej podklas View i CompositeView.

Wzorzec Kompozyt zastosowano też w wielu miejscach platformy kompilatora języka Smalltalk — RTL [JML92]. RTLExpression to odpowiednik klasy Component reprezentujący drzewa składni. Ma ona podklasy, na przykład BinaryExpression, obejmujące obiekty podrzędne typu RTLExpression. Te klasy definiują złożoną strukturę drzew składni. RegisterTransfer to odpowiednik klasy Component reprezentujący program w pośredniej postaci SSA (ang. *Single Static Assignment*, czyli pojedyncze przypisanie statyczne). Podklasy klasy RegisterTransfer reprezentujące liście definiują różne przypisania statyczne, na przykład:

- ▶ proste przypisania wykonujące operacje na dwóch rejestrach i zapisujące wynik w trzecim;
- ▶ przypisania z rejestrem źródłowym, ale bez rejestrów docelowych, co wskazuje na to, że rejestr jest używany po zwróceniu sterowania przez procedurę;
- ▶ przypisania z rejestrem docelowym, ale bez źródła, co wskazuje na to, że dane są przypisywane do rejestrów przed rozpoczęciem działania procedury.

Inna podkласa, RegisterTransferSet, to odpowiednik klasy Composite reprezentujący przypisania zmieniające kilka rejestrów jednocześnie.

Inny przykład zastosowania tego wzorca pochodzi z dziedziny finansów, gdzie portfel obejmuje poszczególne aktywa. Można zapewnić obsługę złożonych zbiorów aktywów przez zaimplementowanie portfela jako klasy Composite zgodnej z interfejsem pojedynczych aktywów [BE93].

Wzorzec Polecenie (s. 302) opisuje, jak składać i porządkować obiekty Command za pomocą klasy MacroCommand (jest to odpowiednik klasy Composite).

POWIĄZANE WZORCE

Powiązanie między komponentami i elementami nadzorowanymi często jest wykorzystywane we wzorcu Łańcuch zobowiązań (s. 244).

Wzorca Dekorator (s. 152) często używa się wraz ze wzorcem Kompozyt. Jeśli dekoratory i kompozyty są stosowane razem, zwykle mają wspólną klasę nadziedzną. Dlatego dekoratory muszą obsługiwać interfejs klasy Component obejmujący takie operacje, jak Add, Remove i GetChild.

Wzorzec Pyłek (s. 201) umożliwia współużytkowanie komponentów, które jednak nie przechowują referencji do elementów nadzorowanych.

Do przechodzenia po zawartości kompozytów można wykorzystać wzorzec Iterotor (s. 230).

Wzorzec Odwiedzający (s. 280) zapewnia jedną lokalizację dla operacji i zachowań, które w innych podejściach są rozproszone po klasach kompozytów i liści.

MOST (BRIDGE)

obiektowy, strukturalny

PRZEZNACZENIE

Oddziela abstrakcję od jej implementacji, dzięki czemu można modyfikować te dwa elementy niezależnie od siebie.

INNE NAZWY

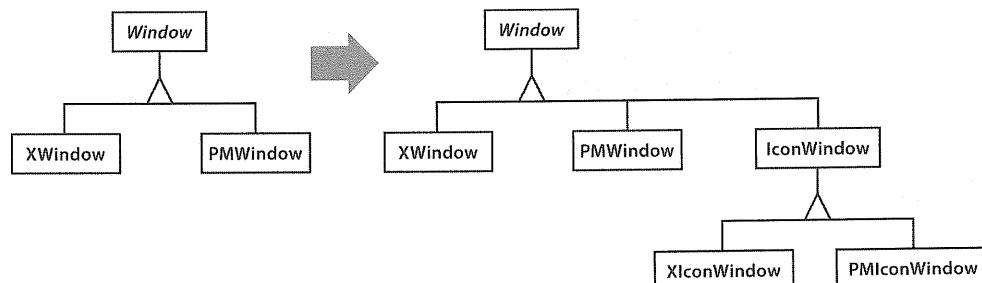
Uchwyt/ciało (ang. *handle/body*).

UZASADNIENIE

Jeśli abstrakcja może mieć jedną z kilku implementacji, zwykle stosowane jest dziedziczenie. W klasie abstrakcyjnej znajduje się definicja interfejsu abstrakcji, a w konkretnych podklasach — różne implementacje. Jednak to podejście nie zawsze jest wystarczająco elastyczne. Dziedziczenie trwale wiąże implementację z abstrakcją, co utrudnia modyfikowanie, rozszerzanie i wielokrotne korzystanie z tych elementów niezależnie od siebie.

Rozważmy implementację przenośnej abstrakcji okna w pakiecie narzędziowym do tworzenia interfejsów użytkownika. Abstrakcja ta powinna umożliwiać rozwijanie aplikacji działających na przykład w systemach X Window i Presentation Manager (PM) firmy IBM. Za pomocą dziedziczenia można zdefiniować klasę abstrakcyjną `Window` oraz podklasy `XWindow` i `PMWindow` z implementacjami interfejsu przeznaczonymi na różne platformy. Jednak podejście to ma dwie wady:

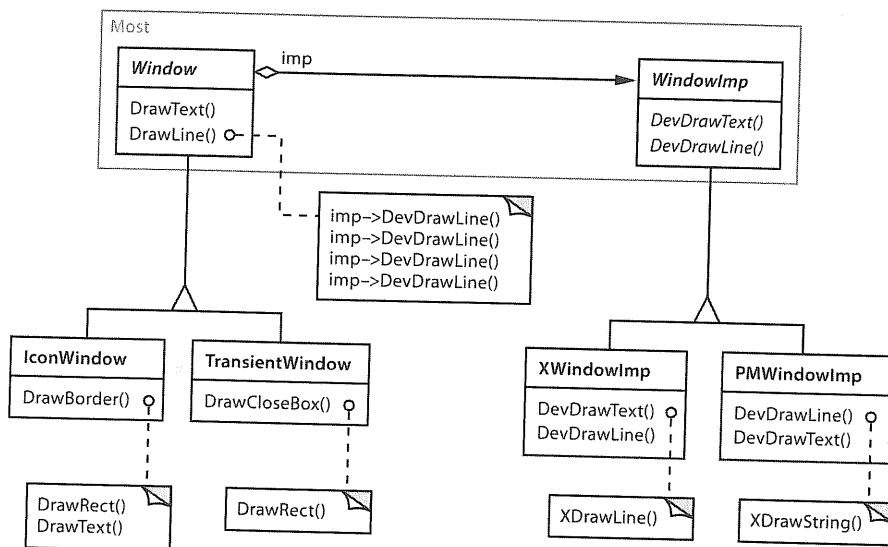
1. Rozszerzanie abstrakcji okna w celu dodania obsługi innych rodzajów okien i nowych platform jest niewygodne. Wyobraźmy sobie podkласę `IconWindow` klasy `Window` (jest to wyspecjalizowana wersja abstrakcji okna reprezentująca ikony). Aby zapewnić obsługę tej podklasy na obu wspomnianych platformach, trzeba zaimplementować *dwie* nowe klasy — `XIconWindow` i `PMIconWindow`. Co gorsze, trzeba zdefiniować dwie klasy dla okna *każdego* rodzaju. Dodanie obsługi trzeciej platformy wymaga utworzenia jeszcze jednej podklasy klasy `Window` dla okien *każdego* typu.



2. Ponadto rozwiązywanie to sprawia, że kod klienta jest zależny od platformy. Kiedy klient generuje okno, tworzy egzemplarz klasy konkretnej o określonej implementacji. Na przykład tworzenie obiektu `XWindow` wiąże abstrakcję okna z implementacją dla systemu X Window, co z kolei powoduje, że kod klienta staje się zależny od tej implementacji. Utrudnia to przeniesienie kodu klienta na inne platformy.

W klientach powinno być możliwe tworzenie okien bez ograniczania się do określonej implementacji. Tylko implementacja okna powinna zależeć od platformy, na której działa aplikacja. Dlatego w kodzie klienta należy tworzyć egzemplarze okien bez wskazywania specyficznych platform.

Wzorzec Most rozwiązuje te problemy, ponieważ umożliwia umieszczenie abstrakcji okna i jej implementacji w odrębnych hierarchiach klas. Należy utworzyć jedną hierarchię klas dla interfejsów okien (`Window`, `IconWindow`, `TransientWindow`) i odrębną dla specyficznych `WindowImp` jako elementem głównym. Podkلاza `XWindowImp` udostępnia na przykład implementację opartą na systemie X Window.



Wszystkie operacje w podklasach klasy `Window` są implementowane w kategoriach operacji abstrakcyjnych z interfejsu klasy `WindowImp`. Rozdziela to abstrakcje okien od różnych specyficznych dla platform implementacji. Relację między klasami `Window` i `WindowImp` nazywamy **mostem**, ponieważ łączy ona abstrakcję z jej implementacją oraz umożliwia ich niezależne modyfikowanie.

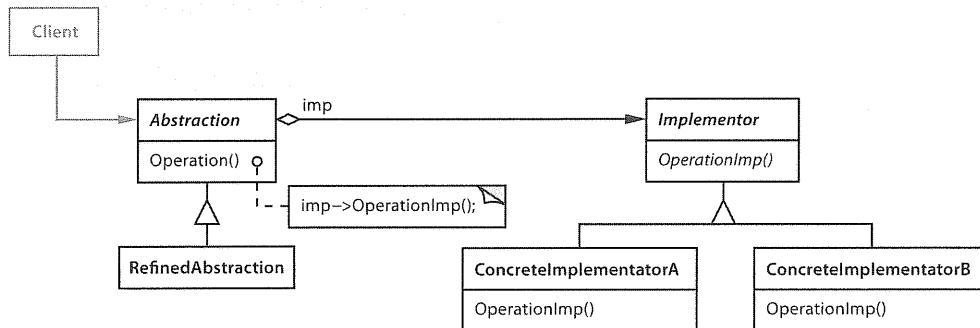
WARUNKI STOSOWANIA

Wzorzec Most należy stosować w następujących warunkach:

- Jeśli chcesz uniknąć trwałego powiązania abstrakcji i jej implementacji. Może to być przydatne na przykład wtedy, kiedy implementację trzeba wybrać lub zmienić w czasie wykonywania programu.

- ▶ Kiedy rozszerzanie przez tworzenie podklas powinno być możliwe zarówno dla abstrakcji, jak i dla implementacji. Wtedy wzorzec Most umożliwia łączenie różnych abstrakcji i implementacji oraz rozszerzanie ich niezależnie od siebie.
- ▶ Jeżeli zmiany w implementacji abstrakcji nie powinny mieć wpływu na klienty (ich kod nie powinien wymagać ponownej komplikacji).
- ▶ W języku C++ — gdy chcesz całkowicie ukryć implementację abstrakcji przed klientami. W języku C++ reprezentacja klasy jest widoczna w jej interfejsie.
- ▶ Jeśli liczba klas szybko rośnie, co przedstawiliśmy na pierwszym diagramie w punkcie „Uzasadnienie”. Taka hierarchia klas wskazuje na potrzebę podziału obiektu na dwie części. Rumbaugh opisuje takie hierarchie za pomocą nazwy „zagospodarowanie uogólnienia” [RBP-91];
- ▶ Kiedy chcesz współużytkować implementację w wielu obiektach (na przykład stosując licznik referencji) i ukryć to przed klientami. Prosty przykład to klasa String opracowana przez Copliena [Cop92]. Umożliwia ona współużytkowanie tej samej reprezentacji łańcucha znaków (StringRep) przez wiele obiektów.

STRUKTURA



ELEMENTY

- ▶ **Abstraction (Window):**
 - definiuje interfejs abstrakcji;
 - przechowuje referencję do obiektu typu **Implementor**.
- ▶ **RefinedAbstraction (IconWindow)**, czyli wzbogacona abstrakcja:
 - rozszerza interfejs zdefiniowany w klasie **Abstraction**.
- ▶ **Implementor (WindowImp):**
 - definiuje interfejs dla klas z implementacją. Interfejs ten nie musi dokładnie odpowiadać interfejsowi klasy **Abstraction** (różnice między nimi mogą być dość znaczne). Zwykle interfejs klasy **Implementor** udostępnia jedynie proste operacje, a w klasie **Abstraction** zdefiniowane są oparte na nich operacje wyższego poziomu.
- ▶ **ConcreteImplementor (XwindowImp, PMwindowImp):**
 - obejmuje implementację interfejsu klasy **Implementor** i definiuje jej implementację konkretną.

WSPÓŁDZIAŁANIE

- Klasa *Abstraction* przekazuje żądania klienta do powiązanego z nią obiektu *Implementor*.

KONSEKWENCJE

Zastosowanie wzorca Most ma następujące konsekwencje:

1. *Oddzielenie interfejsu od implementacji.* Implementacja nie jest trwale powiązana z interfejsem. Implementację abstrakcji można określić w czasie wykonywania programu (można wtedy nawet zmienić implementację obiektu).

Podział na klasy *Abstraction* i *Implementor* eliminuje ponadto zależność od implementacji na etapie komplikowania kodu. Zmiana klasy z implementacją nie wymaga ponownej komplikacji klasy *Abstraction* i korzystających z niej klientów. Ta cecha ma bardzo duże znaczenie, jeśli trzeba zapewnić zgodność binarną między różnymi wersjami biblioteki klas.

Ponadto omawiany podział sprzyja stosowaniu warstw, co może prowadzić do powstania systemu o lepszej strukturze. W wysokopoziomowych częściach systemu potrzebne są tylko informacje o klasach *Abstraction* i *Implementor*.

2. *Łatwiejsze rozszerzanie.* Hierarchie klas *Abstraction* i *Implementor* można rozszerzać niezależnie od siebie.
3. *Ukrycie szczegółów implementacji przed klientami.* Można ukryć przed klientami szczegółowe dotyczące implementacji, takie jak współużytkowanie obiektów *Implementor* i związany z tym licznik referencji (jeśli jest stosowany).

IMPLEMENTACJA

W czasie stosowania wzorca Most należy rozważyć następujące kwestie związane z implementacją:

1. *Korzystanie z tylko jednej klasy Implementor.* Jeśli istnieje tylko jedna implementacja, tworzenie abstrakcyjnej klasy *Implementor* nie jest konieczne. Jest to uproszczona wersja wzorca Most. Między klasami *Abstraction* i *Implementor* występuje tu relacja jeden do jednego. Mimo to ich rozdzielenie jest przydatne, jeśli zmiana w implementacji klasy ma nie wpływać na jej obecne klienty (nie powinno być konieczne ich ponowne komplikowanie, a jedynie powtórne dołączenie).

Carolan [Car89] nazywa ten podział „kotem z Cheshire”. W języku C++ interfejs klasy *Implementor* można zdefiniować w nieudostępnianym klientom prywatnym pliku nagłówkowym. Umożliwia to całkowite ukrycie implementacji przed klientami.

2. *Tworzenie odpowiedniego obiektu Implementor.* Jak, kiedy i gdzie należy określić, którą klasę z rodziny *Implementor* trzeba utworzyć (zakładamy, że istnieje więcej niż jedna taka klasa)?

Jeśli klasa *Abstraction* ma informacje o wszystkich klasach *ConcreteImplementor*, może utworzyć egzemplarz jednej z nich w konstruktorze. Wybór klasy może odbywać się na podstawie parametrów przekazanych do konstruktora. Jeżeli na przykład klasa kolekcji ma wiele implementacji, decyzję można podjąć na podstawie wielkości danej kolekcji. Implementacji w postaci listy powiązanej można użyć dla małych kolekcji, a tablicy haszującej — dla dużych.

Inne podejście polega na wybraniu na początku implementacji domyślnej i późniejszym jej modyfikowaniu w zależności od warunków. Na przykład jeśli wielkość kolekcji wzrośnie ponad określony limit, należy zastąpić jej implementację wersją bardziej odpowiednią dla dużej liczby elementów.

Można też oddelować cały proces podejmowania decyzji do innego obiektu. W przykładzie z klasami `Window` i `WindowImp` można wprowadzić obiekt fabryki (zobacz wzorzec Fabryka abstrakcyjna, s. 101), którego jedynym zadaniem jest kapsułkowanie mechanizmów specyficznych dla platformy. Fabryka potrafi określić, którego rodzaju obiekt `WindowImp` należy utworzyć dla używanej platformy. Obiekt `Window` po prostu żąda obiektu `WindowImp`, a fabryka zwraca obiekt odpowiedniego rodzaju. Zaletą tego rozwiązania jest to, że klasa `Abstraction` nie jest bezpośrednio powiązana z żadną z klas `Implementor`.

3. *Współużytkowanie implementacji.* Coplien w [Cop92] opisuje, jak wykorzystać w języku C++ idiom uchwyty/ciało do współużytkowania implementacji w kilku obiektach. Klasa `Body` przechowuje licznik referencji, a klasa `Handle` zwiększa i zmniejsza jego wartość. Ogólna postać kodu do przypisywania uchwytów ze współużytkowanymi ciałami wygląda tak:

```
Handle& Handle::operator= (const Handle& other) {
    other._body->Ref();
    _body->Unref();

    if (_body->RefCount() == 0) {
        delete _body;
    }
    _body = other._body;

    return *this;
}
```

4. *Stosowanie wielodziedziczenia.* W języku C++ do połączenia interfejsu z jego implementacją można wykorzystać wielodziedziczenie [Mar91]. Klasa może na przykład dziedziczyć publicznie po klasie `Abstraction` i prywatnie po klasie `ConcreteImplementor`. Jednak to podejście oparte jest na dziedziczeniu statycznym, dlatego trwale wiąże implementację z jej interfejsem. Powoduje to, że — przynajmniej w języku C++ — za pomocą wielodziedziczenia nie można zaimplementować prawdziwego wzorca Most.

PRZYKŁADOWY KOD

Poniższy kod w języku C++ to implementacja przykładu dotyczącego klas `Window` i `WindowImp` z punktu „Uzasadnienie”. Klasa `Window` definiuje abstrakcję okna używaną w aplikacjach klienckich.

```
class Window {
public:
    Window(View* contents);

    // Żądania obsługiwane przez okna.
    virtual void DrawContents();

    virtual void Open();
    virtual void Close();
```

```

    virtual void Iconify();
    virtual void Deiconify();

    // Żądania przekazywane do implementacji.
    virtual void SetOrigin(const Point& at);
    virtual void SetExtent(const Point& extent);
    virtual void Raise();
    virtual void Lower();

    virtual void DrawLine(const Point&, const Point&);
    virtual void DrawRect(const Point&, const Point&);
    virtual void DrawPolygon(const Point[], int n);
    virtual void DrawText(const char*, const Point&);

protected:
    WindowImp* GetWindowImp();
    View* GetView();

private:
    WindowImp* _imp;
    View* _contents; // Zawartość okna.
};

```

Klasa `Window` przechowuje referencję do klasy `WindowImp`. Jest to klasa abstrakcyjna z deklaracją interfejsu do używanego systemu okienkowego.

```

class WindowImp {
public:
    virtual void ImpTop() = 0;
    virtual void ImpBottom() = 0;
    virtual void ImpSetExtent(const Point&) = 0;
    virtual void ImpSetOrigin(const Point&) = 0;

    virtual void DeviceRect(Coord, Coord, Coord, Coord) = 0;
    virtual void DeviceText(const char*, Coord, Coord) = 0;
    virtual void DeviceBitmap(const char*, Coord, Coord) = 0;
    // Wiele innych funkcji do wyświetlania okien.

protected:
    WindowImp();
};

```

W podklasach klasy `Window` zdefiniowane są różne rodzaje okien, z których może korzystać aplikacja. Są to między innymi okna aplikacji, ikony, tymczasowe okna dialogowe, pływające palety narzędzi itd.

Na przykład klasa `ApplicationWindow` obejmuje implementację operacji `DrawContents`. Wyświetla ona egzemplarze klasy `View` przechowywane w obiekcie `ApplicationWindow`.

```

class ApplicationWindow : public Window {
public:
    // ...
    virtual void DrawContents();
};

```

```
void ApplicationWindow::DrawContents () {
    GetView()->DrawOn(this);
}
```

Klasa `IconWindow` przechowuje nazwę bitmapy wyświetlanej ikony.

```
class IconWindow : public Window {
public:
    // ...
    virtual void DrawContents();
private:
    const char* _bitmapName;
};
```

Implementacja operacji `DrawContents` w klasie `IconWindow` wyświetla bitmapę w oknie.

```
void IconWindow::DrawContents() {
    WindowImp* imp = GetWindowImp();
    if (imp != 0) {
        imp->DeviceBitmap(_bitmapName, 0.0, 0.0);
    }
}
```

Można utworzyć także wiele innych odmian klasy `Window`. Obiekt `TransientWindow` może wymagać komunikowania się z oknem, które utworzyło go w czasie komunikacji z użytkownikiem (dlatego przechowuje referencję do tego okna). Obiekty `PaletteWindow` zawsze znajdują się nad innymi oknami. Obiekt `IconDockWindow` przechowuje obiekty `IconWindows` i porządkuje je w elegancki sposób.

Operacje klasy `Window` są zdefiniowane w kategoriach interfejsu klasy `WindowImp`. Na przykład operacja `DrawRect` najpierw wyodrębnia cztery współrzędne z dwóch parametrów typu `Point`, a dopiero potem wywołuje operację klasy `WindowImp` przeznaczoną do wyświetlania prostokąta w oknie.

```
void Window::DrawRect (const Point& p1, const Point& p2) {
    WindowImp* imp = GetWindowImp();
    imp->DeviceRect(p1.X(), p1.Y(), p2.X(), p2.Y());
}
```

Podklasy konkretne klasy `WindowImp` obsługują różne systemy okienkowe. Podklasa `XWindowImp` obsługuje system X Window.

```
class XWindowImp : public WindowImp {
public:
    XWindowImp();

    virtual void DeviceRect(Coord, Coord, Coord, Coord);
    // Pozostałe elementy interfejsu publicznego.

private:
    // Dużo specyficznych dla systemu X Window składowych stamu, między innymi:
    Display* _dpy;
    Drawable _winid; // Identyfikator okna.
    GC _gc;          // Kontekst graficzny okna.
};
```

Na potrzeby systemu PM zdefiniujemy klasę PMWindowImp:

```
class PMWindowImp : public WindowImp {
public:
    PMWindowImp();
    virtual void DeviceRect(Coord, Coord, Coord, Coord);

    // Pozostałe elementy interfejsu publicznego.

private:
    // Dużo specyficznych dla systemu PM składowych stanu, między innymi:
    HPS _hps;
};
```

W tych podklasach operacje z klasy WindowImp są zaimplementowane w kategoriach podstawowych operacji systemów okienkowych. Na przykład implementacja operacji DeviceRect dla systemu X wygląda tak:

```
void XWindowImp::DeviceRect (
    Coord x0, Coord y0, Coord x1, Coord y1
) {
    int x = round(min(x0, x1));
    int y = round(min(y0, y1));
    int w = round(abs(x0 - x1));
    int h = round(abs(y0 - y1));
    XDrawRectangle(_dpy, _winid, _gc, x, y, w, h);
}
```

A oto przykładowa implementacja dla systemu PM:

```
void PMWindowImp::DeviceRect (
    Coord x0, Coord y0, Coord x1, Coord y1
) {
    Coord left = min(x0, x1);
    Coord right = max(x0, x1);
    Coord bottom = min(y0, y1);
    Coord top = max(y0, y1);

    PPOINTL point[4];

    point[0].x = left;    point[0].y = top;
    point[1].x = right;   point[1].y = top;
    point[2].x = right;   point[2].y = bottom;
    point[3].x = left;    point[3].y = bottom;

    if (
        (GpiBeginPath(_hps, 1L) == false) ||
        (GpiSetCurrentPosition(_hps, &point[3]) == false) ||
        (GpiPolyLine(_hps, 4L, point) == GPI_ERROR) ||
        (GpiEndPath(_hps) == false)
    ) {
        // Zgłaszanie błędu.
    } else {
        GpiStrokePath(_hps, 1L, 0L);
    }
}
```

W jaki sposób okno może uzyskać egzemplarz odpowiedniej podklasy klasy `WindowImp`? Na potrzeby tego przykładu założymy, że odpowiada za to klasa `Window`. Jej operacja `GetWindowImp` pobiera właściwy egzemplarz z fabryki abstrakcyjnej (zobacz wzorzec Fabryka abstrakcyjna, s. 101), która skutecznie kapsułkuje wszystkie mechanizmy specyficzne dla systemów okienkowych.

```
WindowImp* Window::GetWindowImp () {
    if (_imp == 0) {
        _imp = WindowSystemFactory::Instance()->MakeWindowImp();
    }
    return _imp;
}
```

Operacja `WindowSystemFactory::Instance()` zwraca fabrykę abstrakcyjną wytwarzającą wszystkie obiekty specyficzne dla systemu okienkowego. Dla uproszczenia fabrykę tę utworzyliśmy jako singleton (s. 130) i umożliwiłyśmy klasie `Window` bezpośrednie korzystanie z fabryki.

ZNANE ZASTOSOWANIA

Wcześniej przykład dotyczący klasy `Window` pochodzi z platformy ET++ [WGM88]. W tej platformie odpowiednik klasy `WindowImp` nosi nazwę `WindowPort` i ma podklasy w rodzaju `XWindowPort` oraz `SunWindowPort`. Obiekt okna tworzy odpowiadający mu obiekt z implementacją przez zażądanie go od fabryki abstrakcyjnej o nazwie `WindowSystem`. Klasa `WindowSystem` udostępnia interfejs do tworzenia obiektów specyficznych dla platformy, takich jak czcionki, kursory, bitmapy itd.

W projekcie klas `Window` i `WindowPort` platformy ET++ zastosowano wzbogacony wzorzec Most, ponieważ w obiekcie `WindowPort` znajduje się referencja powrotna do obiektu `Window`. W klasie z implementacją obiektów `WindowPort` referencja ta służy do powiadamiania obiektów `Window` o zdarzeniach specyficznych dla klasy `WindowPort` — wprowadzeniu danych wejściowych, zmianie rozmiaru okna itd.

Zarówno Coplien [Cop92], jak i Stroustrup [Str91] wspominają o klasach `Handle` i podają związane z nimi przykłady. Kładą w nich nacisk na kwestie dotyczące zarządzania pamięcią, takie jak współużytkowanie reprezentacji łańcuchów znaków i obsługa obiektów o zmiennej wielkości. My w większym stopniu koncentrujemy się na umożliwieniu niezależnego rozszerzania abstrakcji i jej implementacji.

W bibliotece `libg++` [Lea88] zdefiniowane są klasy z implementacją standardowych struktur danych — między innymi `Set`, `LinkedSet`, `HashSet` i `HashTable`. `Set` to klasa abstrakcyjna reprezentująca abstrakcję zbioru, natomiast `LinkedList` i `HashTable` to konkretne implementacje listy powiązanej oraz tablicy haszującej. `LinkedSet` i `HashSet` to implementacje abstrakcji `Set` łączące klasę `Set` z jej konkretnymi odpowiednikami `LinkedList` i `HashTable`. Jest to przykład uproszczonego mostu, ponieważ nie występuje tu odpowiednik klasy abstrakcyjnej `Implementor`.

W pakiecie AppKit [Add94] firmy NeXT wzorzec Most zastosowano do implementowania i wyświetlania obrazów. Obraz można przedstawić na kilka różnych sposobów. To, który z nich jest optymalny, zależy od cech wyświetlacza — a konkretnie liczby kolorów i rozdzielczości. Bez pomocy ze strony pakietu AppKit programiści musieliby w każdej aplikacji określić, którą implementację należy zastosować w danych warunkach.

Aby uwolnić programistów od tego zadania, w pakiecie AppKit udostępniono most dla klas `NXImage` i `NXImageRep`. Klasa `NXImage` obejmuje interfejs do obsługi obrazów. Ich implementacja jest zdefiniowana w odrębnej hierarchii klasy `NXImageRep`. Znajdują się w niej takie podklasy, jak `NXEPSImageRep`, `NXCachedImageRep` i `NXBitMapImageRep`. Klasa `NXImage` przechowuje referencję do przynajmniej jednego obiektu `NXImageRep`. Jeśli istnieje więcej niż jedna implementacja obrazu, klasa `NXImage` wybiera najbardziej odpowiednią dla obecnie używanego wyświetlacza. Klasa `NXImage` potrafi nawet w razie potrzeby przekształcić jedną implementację w inną. Ciekawą cechą tej wersji mostu jest to, że w klasie `NXImage` można w danym momencie przechowywać więcej niż jedną implementację klasy `NXImageRep`.

POWIĄZANE WZORCE

Do utworzenia i skonfigurowania określonego mostu można użyć wzorca Fabryka abstrakcyjna (s. 101).

Wzorzec Adapter (s. 141) jest przeznaczony do umożliwiania współdziałania między niepowiązanymi klasami. Zwykle stosuje się go w systemach po ich zaprojektowaniu, natomiast ze wzorca Most korzysta się przed rozpoczęciem projektowania, aby umożliwić niezależne modyfikowanie abstrakcji i implementacji.

PRZEZNACZENIE

Udostępnia zastępnik lub reprezentanta innego obiektu w celu kontrolowania dostępu do niego.

NNE NAZWY

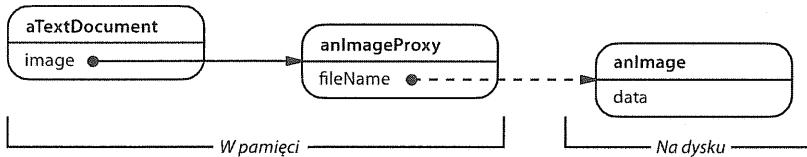
Substytut (ang. *surrogate*).

JZASADNIENIE

Jedną z przyczyn kontrolowania dostępu do obiektu jest odraczanie ponoszenia pełnych kosztów jego tworzenia i inicjowania do momentu, kiedy będzie potrzebny. Rozważmy edytor dokumentów, który umożliwia zagnieżdżanie w plikach obiektów graficznych. Tworzenie niektórych obiektów tego rodzaju, na przykład dużych rysunków rastrowych, może być kosztowne. Jednak otwieranie dokumentu powinno przebiegać szybko, dlatego w momencie uruchamiania pliku należy uniknąć jednoczesnego tworzenia wszystkich kosztownych obiektów. Zresztą tworzenie ich w ten sposób nie jest konieczne, ponieważ nie każdy z tych obiektów będzie widoczny w dokumencie w tym samym momencie.

Te ograniczenia wskazują na to, że program powinien tworzyć każdy kosztowny obiekt *na żądanie*, czyli w tym przykładzie wtedy, kiedy rysunek stanie się widoczny. Co jednak należy umieścić w dokumencie zamiast obrazka? I jak ukryć fakt, że jest on tworzony na żądanie, tak aby nie komplikować implementacji edytora? Wspomniana optymalizacja nie powinna mieć wpływu na przykład na kod wyświetlający i formatujący dokument.

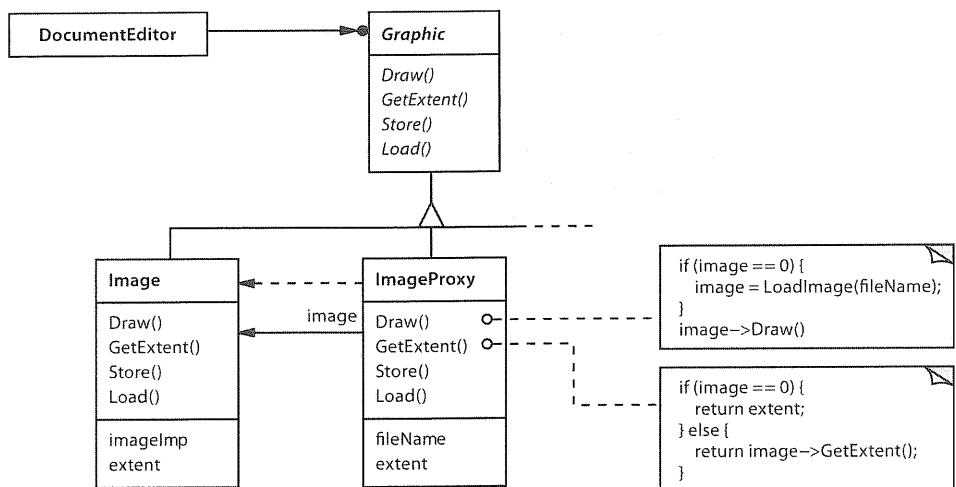
Rozwiązań polega na zastosowaniu innego obiektu, **pełnomocnika** rysunku, który będzie działał jako zastępca rzeczywistego obrazka. Pełnomocnik funkcjonuje tak samo jak rysunek i odpowiada za utworzenie egzemplarza obrazka, kiedy ten jest potrzebny.



Pełnomocnik tworzy rzeczywisty rysunek tylko wtedy, kiedy edytor dokumentów wywoła operację Draw obrazka, żądając w ten sposób jego wyświetlenia. Od tej pory pełnomocnik przekazuje żądania bezpośrednio do rysunku. Dlatego musi przechowywać referencję do niego po jego utworzeniu.

Założymy, że rysunki są przechowywane w odrębnych plikach. W takich warunkach jako referencję do rzeczywistego obiektu możemy wykorzystać nazwę pliku. Pełnomocnik przechowuje też rozmiar obiektu, czyli jego wysokość i szerokość. Informacje o rozmiarze umożliwiają pełnomocnikowi reagowanie na wysypane przez mechanizm formatujący żądania podania wielkości rysunku bez konieczności tworzenia jego egzemplarza.

Poniższy diagram klas ilustruje ten przykład bardziej szczegółowo.



Edytor dokumentów uzyskuje dostęp do zagnieźdzonych rysunków poprzez interfejs zdefiniowany w klasie abstrakcyjnej `Graphic`. `ImageProxy` to klasa obrazków tworzonych na żądanie. Przechowuje ona nazwę pliku jako referencję do rysunku zapisanego na dysku. Nazwa ta jest przekazywana jako argument do konstruktora klasy `ImageProxy`.

Obiekt `ImageProxy` przechowuje też ramkę ograniczającą rysunku i referencję do prawdziwego egzemplarza rysunku. Referencja ta jest nieprawidłowa do momentu utworzenia egzemplarza obrazka przez pełnomocnika. Operacja `Draw` przed przekazaniem żądania do egzemplarza rysunku sprawdza, czy jest on gotowy. Operacja `GetExtent` przekazuje żądanie do rysunku tylko wtedy, jeśli utworzony jest jego egzemplarz. Jeżeli taki egzemplarz nie istnieje, obiekt `ImageProxy` zwraca zapisany rozmiar.

WARUNKI STOSOWANIA

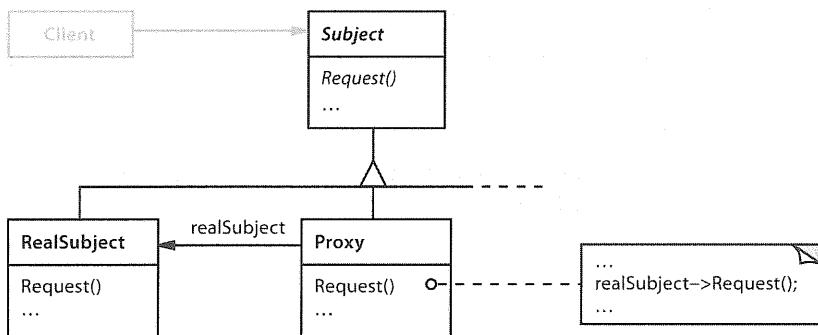
Z pełnomocnikiem można korzystać zawsze wtedy, kiedy potrzebna jest bardziej wszechstronna i zaawansowana referencja do obiektu niż zwykły wskaźnik. Oto kilka typowych sytuacji, w których wzorzec Pełnomocnik jest przydatny:

1. **Zdalny pełnomocnik** to lokalny reprezentant obiektu w innej przestrzeni adresowej. W systemie NEXTSTEP [Add94] zastosowano w tym celu klasę `NXProxy`. Coplien [Cop92] nazywa pełnomocnika tego rodzaju „ambasadorem”.
2. **Pełnomocnik wirtualny** na żądanie tworzy kosztowne obiekty. Przykładem takiego pełnomocnika jest klasa `ImageProxy` opisana w punkcie „Uzasadnienie”.
3. **Pośrednik zabezpieczający** kontroluje dostęp do pierwotnego obiektu. Pełnomocnicy tego rodzaju są przydatne, jeśli obiekty powinny mieć różne prawa dostępu. Na przykład pełnomocnik `KernelProxy` w systemie operacyjnym Choices [CIRM93] zabezpiecza dostęp do obiektów systemu operacyjnego.

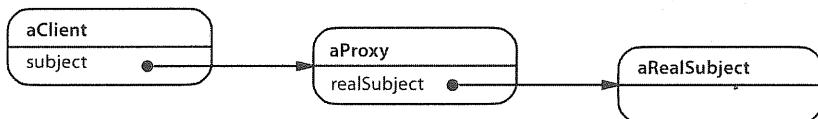
4. Inteligentne referencje zastępują zwykłe wskaźniki i wykonują dodatkowe operacje przy dostępie do obiektu. Oto typowe zastosowania tej techniki:

- zliczanie referencji prowadzących do pierwotnego obiektu, co umożliwia jego automatyczne zwolnienie, kiedy liczba referencji spadnie do zera (inna nazwa to **inteligentne wskaźniki** [Ede92]);
- wczytywanie trwałych obiektów do pamięci w momencie utworzenia pierwszej referencji do nich;
- sprawdzanie przed dostępem do pierwotnego obiektu, czy jest on zablokowany, co gwarantuje, że żaden inny obiekt go nie zmodyfikuje.

STRUKTURA



Oto przykładowy diagram obiektów przedstawiający strukturę z pełnomocnikiem w czasie wykonywania programu:



ELEMENTY

► Proxy (ImageProxy), czyli pełnomocnik:

- przechowuje referencję umożliwiającą pełnomocnikowi dostęp do rzeczywistego obiektu; referencja ta może prowadzić do obiektu Subject, jeśli klasy RealSubject i Subject mają takie same interfejsy;
- udostępnia interfejs identyczny z interfejsem klasy Subject, dzięki czemu pełnomocnik można podstawić pod rzeczywisty obiekt;
- kontroluje dostęp do rzeczywistego obiektu i może odpowiadać za jego tworzenie oraz usuwanie;

- pełni inne, zależne od rodzaju pełnomocnika zadania:
 - *Pełnomocnicy zdalne* odpowiadają za kodowanie żądań i ich argumentów oraz przekazywanie zakodowanych żądań do rzeczywistego obiektu w innej przestrzeni adresowej.
 - *Pełnomocniki wirtualne* mogą przechowywać dodatkowe informacje o rzeczywistym obiekcie, co pozwala opóźnić dostęp do niego. Na przykład klasa *ImageProxy* z punktu „Uzasadnienie” przechowuje rozmiar rzeczywistego rysunku.
 - *Pełnomocniki zabezpieczające* sprawdzają, czy nadawca żądania ma uprawnienia dostępu potrzebne do wysłania danego żądania.
- **Subject (Graphic)**, czyli obiekt:
 - definiuje wspólny interfejs klas *RealSubject* i *Proxy*, dzięki czemu obiektów *Proxy* można używać wszędzie tam, gdzie oczekiwane są obiekty *RealSubject*.
- **RealSubject (Image)**, czyli rzeczywisty obiekt:
 - definiuje rzeczywisty obiekt reprezentowany przez pełnomocnik.

WSPÓŁDZIAŁANIE

- Obiekt *Proxy* przekazuje w odpowiednich momentach (zależą one od rodzaju pełnomocnika) żądania do obiektu *RealSubject*.

KONSEKWENCJE

Wzorzec Pełnomocnik wprowadza poziom pośredniości przy dostępie do obiektu. Ma to wiele zastosowań zależnych od rodzaju pośrednika:

1. Pośrednik zdalny może ukrywać fakt, że obiekt znajduje się w innej przestrzeni adresowej.
2. Pośrednik wirtualny może umożliwiać optymalizację, na przykład przez tworzenie obiektów na żądanie.
3. Pełnomocniki zabezpieczające i inteligentne referencje umożliwiają wykonywanie dodatkowych operacji porządkowych przy dostępie do obiektu.

Jest jeszcze jedna optymalizacja, którą można ukryć przed klientami za pomocą wzorca Pełnomocnik. Jej nazwa to **kopiowanie przy zapisie**. Technika ta powiązana jest z tworzeniem obiektów na żądanie. Kopiowanie dużych i skomplikowanych obiektów bywa kosztowną operacją. Jeśli kopia nigdy nie jest modyfikowana, nie ma potrzeby ponoszenia takich kosztów. Przez zastosowanie pełnomocnika do opóźnienia procesu kopowania gwarantujemy, że koszty te pojawią się tylko wtedy, jeżeli obiekt zostanie zmieniony.

Aby technika kopowania przy zapisie mogła działać, trzeba zliczać referencje do podmiotu. Przy kopowaniu pełnomocnika wystarczy wtedy jedynie zwiększyć wartość licznika referencji. Tylko wtedy, kiedy klient zażąda uruchomienia operacji modyfikującej obiekt, pełnomocnik rzeczywiście go skopiuje. Wtedy pełnomocnik musi też zmniejszyć wartość licznika referencji do obiektu. Kiedy spadnie ona do zera, obiekt zostanie usunięty.

Kopiowanie przy zapisie może znacznie zmniejszyć koszt kopowania dużych obiektów.

IMPLEMENTACJA

Przy stosowaniu wzorca Pełnomocnik można wykorzystać następujące funkcje języka:

1. *Przeciążanie operatora dostępu do składowych w języku C++.* Język ten obsługuje przeciążanie operatora dostępu do składowych — `operator->`. Przeciążenie tego operatora umożliwia wykonywanie dodatkowych zadań przy operacji dereferencji obiektu. Może to być pomocne przy implementowaniu pełnomocników niektórych rodzajów (pełnomocnik działa wtedy jak wskaźnik).

Poniższy przykład pokazuje, jak zastosować tę technikę do zaimplementowania pełnomocnika wirtualnego (tu nosi on nazwę `ImagePtr`).

```
class Image;
extern Image* LoadAnImageFile(const char* );
// Zewnętrzna funkcja.

class ImagePtr {
public:
    ImagePtr(const char* imageFile);
    virtual ~ImagePtr();

    virtual Image* operator->();
    virtual Image& operator*();

private:
    Image* LoadImage();
private:
    Image* _image;
    const char* _imageFile;
};

ImagePtr::ImagePtr (const char* theImageFile) {
    _imageFile = theImageFile;
    _image = 0;
}

Image* ImagePtr::LoadImage () {
    if (_image == 0) {
        _image = LoadAnImageFile(_imageFile);
    }
    return _image;
}
```

W przeciążonych operatorach `->` i `*` wykorzystano operację `LoadImage` do zwrócenia zmiennej `_image` do nadawcy żądania (w razie konieczności rysunek jest wczytywany).

```
Image* ImagePtr::operator-> () {
    return LoadImage();
}

Image& ImagePtr::operator* () {
    return *LoadImage();
}
```

To podejście umożliwia wywoływanie operacji z klasy `Image` za pośrednictwem obiektów `ImagePtr` bez kłopotania się o dodawanie tych operacji do interfejsu klasy `ImagePtr`.

```
ImagePtr image = ImagePtr("anImageFileName");
image->Draw(Point(50, 100));
// (image.operator->())->Draw(Point(50, 100))
```

Warto zauważyć, że pełnomocnik `image` działa jak wskaźnik, ale nie jest zadeklarowany jako wskaźnik do obiektów `Image`. Oznacza to, że nie można korzystać z niego jak z prawdziwego wskaźnika do obiektów `Image`. Dlatego w tym rozwiążaniu klienci muszą traktować obiekty `Image` i `ImagePtr` w odmienny sposób.

Nie we wszystkich pełnomocnikach przeciążanie operatora dostępu do składowych jest dobrym rozwiążaniem. Niektóre pełnomocniki muszą dokładnie ustalić, które operacje są wywoływane. Wtedy nie można zastosować przeciążania operatora dostępu do składowych.

Zastanówmy się nad przykładowym pełnomocnikiem wirtualnym z punktu „Uzasadnienie”. Rysunek należy wczytać w określonym czasie (czyli przy wywołaniu operacji `Draw`), a nie przy każdym użyciu referencji do niego. Przeciążenie operatora dostępu nie umożliwia wprowadzenia tego rozróżnienia. W tym przypadku trzeba ręcznie zaimplementować każdą operację pełnomocnika przekazującą żądanie do podmiotu.

Wspomniane operacje są zwykle bardzo podobne do siebie, jak ilustruje to punkt „Przykładowy kod”. Standardowo wszystkie operacje przed przekazaniem żądania do obiektu sprawdzają, czy jest ono dozwolone, czy pierwotny obiekt istnieje itd. Dlatego powszechnie korzysta się z preprocesora do automatycznego generowania takich operacji.

2. *Stosowanie metody `doesNotUnderstand` w języku Smalltalk.* Język Smalltalk udostępnia mechanizm, który można wykorzystać do obsługi automatycznego przekazywania żądań. Kiedy klient prześle komunikat do odbiorcy, który nie udostępnia odpowiedniej metody, język Smalltalk wywołuje operację `doesNotUnderstand`: `aMessage`. W klasie `Proxy` można przedefiniować metodę `doesNotUnderstand`, tak aby przekazać komunikat do obiektu.

Aby zagwarantować, że żądanie trafi do obiektu i nie zostanie „po cichu” przejęte przez pełnomocnik, można zdefiniować klasę `Proxy` niezdolną do rozpoznania *jakichkolwiek* komunikatów. Język Smalltalk umożliwia osiągnięcie tego celu przez zdefiniowanie `Proxy` jako klasy pozbawionej nadklas³.

Główna wadą stosowania metody `doesNotUnderstand`: jest to, że w większości systemów opartych na języku Smalltalk działa zestaw komunikatów specjalnych obsługiwanych bezpośrednio przez maszynę wirtualną. Nie wymagają one standardowego wyszukiwania metod. Jedynym komunikatem, który zwykle jest implementowany w klasie `Object` (a tym samym wpływa na pełnomocniki), jest operator tożsamości (`==`).

Jeśli zamierzasz zastosować metodę `doesNotUnderstand`: do zaimplementowania wzorca Pełnomocnik, musisz rozwiązać w projekcie ten problem. Nie możesz oczekiwać, że identyczność pełnomocników oznacza identyczność rzeczywistych obiektów. Dodatkową wadą omawianego podejścia jest to, że metodę `doesNotUnderstand`: opracowano do obsługi błędów, a nie do tworzenia pełnomocników, dlatego nie działa ona zbyt szybko.

³ Technikę tę zastosowano do implementacji obiektów rozproszonych (a dokładniej — klasy `NXProxy`) w systemie NEXTSTEP [Add94]. W systemie tym przedefiniowano instrukcję `forward` — odpowiednik omawianego mechanizmu w systemie NEXTSTEP.

3. Pełnomocnik nie zawsze musi znać typ rzeczywistego obiektu. Jeśli klasa Proxy potrafi obsługiwać obiekt wyłącznie za pośrednictwem abstrakcyjnego interfejsu, nie trzeba tworzyć od自救nej klasy tego rodzaju dla każdej klasy RealSubject. Wtedy pełnomocnik może w jednolity sposób obsługiwać wszystkie klasy RealSubject. Jednak jeśli klasy Proxy mają tworzyć egzemplarze klas RealSubject (tak działają pełnomocniki wirtualne), muszą znać ich klasy konkretne.

Inne zagadnienie implementacyjne związane jest z tym, jak odwoływać się do obiektu przed utworzeniem jego egzemplarza. Niektóre pełnomocniki muszą odwoływać się do obiektu niezależnie od tego, czy znajduje się on na dysku czy w pamięci. Oznacza to, że muszą korzystać z pewnego rodzaju identyfikatorów obiektów niezależnych od przestrzeni adresowej. W punkcie „Uzasadnienie” użyliśmy do tego nazw plików.

PRZYKŁADOWY KOD

W poniższym kodzie zaimplementowano pełnomocniki dwóch rodzajów — wirtualny pełnomocnik opisany w punkcie „Uzasadnienie” i pełnomocnik oparty na metodzie `doesNotUnderstand:`⁴.

1. *Pośrednik wirtualny.* Klasa `Graphic` definiuje interfejs obiektów graficznych:

```
class Graphic {
public:
    virtual ~Graphic();

    virtual void Draw(const Point& at) = 0;
    virtual void HandleMouse(Event& event) = 0;

    virtual const Point& GetExtent() = 0;

    virtual void Load(istream& from) = 0;
    virtual void Save(ostream& to) = 0;
protected:
    Graphic();
};
```

W klasie `Image` zaimplementowany jest interfejs klasy `Graphic`, co umożliwia wyświetlanie plików graficznych. Ponadto w klasie `Image` przesłonięto operację `HandleMouse`, aby umożliwić użytkownikom interaktywne zmienianie rozmiarów obrazków.

```
class Image : public Graphic {
public:
    Image(const char* file); // Wczytuje rysunek z pliku.
    virtual ~Image();

    virtual void Draw(const Point& at);
    virtual void HandleMouse(Event& event);

    virtual const Points GetExtent();
    virtual void Load(istream& from);
```

⁴ Inny rodzaj pełnomocnika opisaliśmy w podrozdziale poświęconym wzorcowi Iterator (s. 257) na stronie 266.

```

        virtual void Save(ostream& to);
private:
    // ...
};

```

Klasa ImageProxy ma taki sam interfejs jak klasa Image:

```

class ImageProxy : public Graphic {
public:
    ImageProxy(const char* fileName);
    virtual ~ImageProxy();

    virtual void Draw(const Point& at);
    virtual void HandleMouse(Event& event);

    virtual const Point& GetExtent();

    virtual void Load(istream& from);
    virtual void Save(ostream& to);

protected:
    Image* GetImage();
private:
    Image* _image;
    Point _extent;
    char* _fileName;
};

```

Konstruktor zapisuje lokalną kopię nazwy pliku z rysunkiem oraz inicjuje zmienne `_extent` i `_image`:

```

ImageProxy::ImageProxy (const char* fileName) {
    _fileName = strdup(fileName);
    _extent = Point::Zero; // Na razie rozmiar nie jest znany.
    _image = 0;
}

Image* ImageProxy::GetImage() {
    if (_image == 0) {
        _image = new Image(_fileName);
    }
    return _image;
}

```

Operacja `GetExtent` zwraca zapisany rozmiar, jeśli jest to możliwe; jeżeli nie można tego zrobić, wczytuje rysunek z pliku. Operacja `Draw` wczytuje obrazek, a operacja `HandleMouse` przekazuje zdarzenie do rzeczywistego obiektu rysunku.

```

const Point& ImageProxy::GetExtent () {
    if (_extent == Point::Zero) {
        _extent = GetImage()->GetExtent();
    }
    return _extent;
}

void ImageProxy::Draw (const Point& at) {
    GetImage()->Draw(at);
}

```

```

}

void ImageProxy::HandleMouse (Event& event) {
    GetImage()->HandleMouse(event);
}

```

Operacja Save zapisuje do strumienia dane z pamięci podręcznej (rozmiar rysunku i nazwę pliku). Operacja Load wczytuje te informacje i inicjuje odpowiednie składowe.

```

void ImageProxy::Save (ostream& to) {
    to << _extent << _fileName;
}

void ImageProxy::Load (istream& from) {
    from >> _extent >> _fileName;
}

```

Na zakończenie założymy, że program obejmuje klasę TextDocument zawierającą obiekty Graphic:

```

class TextDocument {
public:
    TextDocument();

    void Insert(Graphic*);
    // ...
};

```

Obiekt ImageProxy możemy wstawić do obiektu TextDocument w następujący sposób:

```

TextDocument* text = new TextDocument;
// ...
text->Insert(new ImageProxy("anImageFileName"));

```

2. *Pełnomocniki korzystające z metody doesNotUnderstand*. W języku Smalltalk można utworzyć uniwersalne pełnomocniki przez zdefiniowanie klas o nadklasie nil⁵ i użycie metody doesNotUnderstand: do obsługi komunikatów.

Poniższa metoda oparta jest na założeniu, że pełnomocnik udostępnia metodę realSubject zwracającą rzeczywisty obiekt. W klasie ImageProxy metoda ta najpierw sprawdza, czy obiekt Image został utworzony, następnie — jeśli jest to konieczne — generuje go, a na końcu zwraca. W celu wywołania przechwyconego komunikatu na rzeczywistym obiekcie w metodzie wykorzystano instrukcję perform:withArguments:.

```

doesNotUnderstand: aMessage
    ^ self realSubject
        perform: aMessage selector
            withArguments: aMessage arguments

```

Argumentem metody doesNotUnderstand: jest egzemplarz klasy Message reprezentujący komunikat niezrozumiany przez pełnomocnika. Dlatego pełnomocnik reaguje na wszystkie komunikaty przez upewnienie się, że rzeczywisty obiekt istnieje, a dopiero potem przekazuje do niego wiadomość.

⁵ Ostateczną nadklassą prawie wszystkich klas jest Object. Dlatego stwierdzenie z tekstu oznacza: „zdefiniowanie klasy, której nadklassą nie jest Object”.

Jedną z zalet stosowania metody `doesNotUnderstand:` jest to, że pozwala ona wykonywać dowolne zadania. Można na przykład utworzyć pełnomocnik zabezpieczający przez określone zestawu akceptowanych komunikatów `legalMessages` i umieszczenie w pełnomocniku następującej metody:

```
doesNotUnderstand: aMessage
    ^ (legalMessages includes: aMessage selector)
        ifTrue: [self realSubject
            perform: aMessage selector
            withArguments: aMessage arguments]
        ifFalse: [self error: 'Niedozwolony operator']
```

Metoda ta przed przekazaniem komunikatu do rzeczywistego obiektu sprawdza, czy dany komunikat jest dozwolony. Jeśli nie jest, metoda wysyła wiadomość `error:` do pełnomocnika, co doprowadzi do powstania nieskończonej pętli zgłaszanych błędów, jeżeli w pełnomocniku nie ma zdefiniowanej metody `error:`. Dlatego należy skopiować z klasy `Object` definicję tej metody wraz z wykorzystywanyymi w niej metodami.

ZNANE ZASTOSOWANIA

Przykład dotyczący pełnomocnika wirtualnego z punktu „Uzasadnienie” pochodzi z klas platformy ET++ reprezentujących bloki to tworzenia tekstu.

W systemie NEXTSTEP [Add94] pełnomocniki (egzemplarze klasy `NXProxy`) pełnią funkcję lokalnych reprezentantów obiektów, z których można korzystać w środowisku rozproszonym. Serwer tworzy pełnomocniki zdalnych obiektów na żądanie klientów. Po otrzymaniu komunikatu pełnomocnik koduje go wraz z argumentami, a następnie przekazuje zakodowaną wiadomość do zdalnego obiektu. W podobny sposób obiekt koduje zwiercane wyniki i przesyła je z powrotem do obiektu `NXProxy`.

McCullough [McC87] analizuje wykorzystanie pełnomocników w języku Smalltalk do dostępu do zdalnych obiektów. Pascoe [Pas86] opisuje, jak za pomocą obiektów `Encapsulator` wykonywać dodatkowe zadania przy wywoływaniu metod i kontroli dostępu.

POWIĄZANE WZORCE

Adapter (s. 141). Adapter służy do tworzenia nowego interfejsu dostosowywanego obiektu, natomiast interfejs udostępniany przez pełnomocnik jest taki sam jak w powiązanym z nim obiekcie. Jednak pełnomocnik stosowany do zabezpieczania dostępu może odrzucić żądanie wykonania operacji, którą obiekt obsługuje, dlatego interfejs pełnomocnika może być tylko podzioborem interfejsu obiektu.

Dekorator (s. 152). Choć dekoratory można implementować podobnie jak pełnomocniki, ich przeznaczenie jest inne. Dekorator dodaje zadania do obiektu, natomiast pełnomocnik kontroluje dostęp do niego.

Poszczególne pełnomocniki różnią się między sobą ze względu na podobieństwo ich implementacji do dekoratorów. Pełnomocnik zabezpieczający można zaimplementować dokładnie tak jak dekorator. Jednak pełnomocnik zdalny nie przechowuje bezpośredniej referencji do rzeczywistego obiektu, a jedynie odwołanie pośrednie, na przykład identyfikator hosta i lokalny adres w danym hoście. Pośrednik wirtualny początkowo obejmuje referencję pośrednią, taką jak nazwa pliku, jednak ostatecznie pozyskuje i stosuje referencję bezpośrednią.

PYŁEK (FLYWEIGHT)

obiektowy, strukturalny

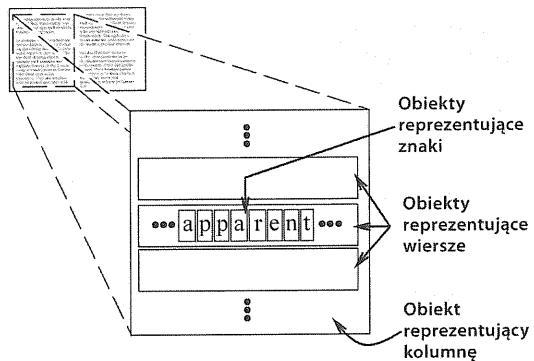
PRZEZNACZENIE

Wykorzystuje współdzielenie do wydajnej obsługi dużej liczby małych obiektów.

UZASADNIENIE

W niektórych aplikacjach korzystne jest całkowite oparcie projektu na obiektach, ale koszty naiwnej realizacji tego rozwiązania mogą okazać się nieakceptowalne.

Na przykład w implementacjach większości edytorów tekstu znajdują się narzędzia do formatowania i edycji tekstu podzielone do pewnego stopnia na moduły. W obiektowych edytorach tekstu obiekty zwykle służą do reprezentowania zagnieżdżonych elementów, takich jak tabele i rysunki. Jednak zazwyczaj obiekty nie są wykorzystywane do przedstawiania każdego znaku w dokumencie, choć takie podejście zwiększa elastyczność aplikacji na jej najniższym poziomie. Znaki i zagnieżdżone elementy można wtedy traktować jednakowo w czasie wyświetlania i formatowania. Aplikację można rozszerzyć o obsługę nowych zestawów znaków bez naruszania innych funkcji. Struktura obiektów aplikacji może odzwierciedlać fizyczną strukturę dokumentu. Poniższy diagram pokazuje, jak w edytorze dokumentów wykorzystać obiekty do reprezentowania znaków.



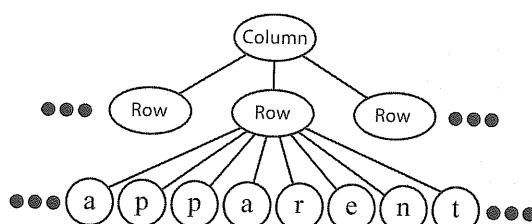
Wadą takiego projektu jest koszt jego stosowania. Nawet średniej wielkości dokumenty mogą wymagać utworzenia setek tysięcy obiektów reprezentujących znaki, co spowoduje zużycie dużej ilości pamięci i może prowadzić do nieakceptowanego spadku wydajności w czasie wykonywania programu. Wzorzec Pyłek pokazuje, jak współużytkować obiekty, tak aby można korzystać z nich na szczegółowym poziomie bez ponoszenia nadmiernych kosztów.

Pyłek to współużytkowany obiekt, z którego można jednocześnie korzystać w wielu kontekstach. Pyłek w każdym kontekście działa jak niezależny obiekt. Nie da się go odróżnić od niewspółużytkowanych obiektów. Działanie pyłków nie może być zależne od kontekstu. Kluczowym zagadnieniem jest tu rozróżnienie na stan **wewnętrzny** i **zewnętrzny**. Stan wewnętrzny jest

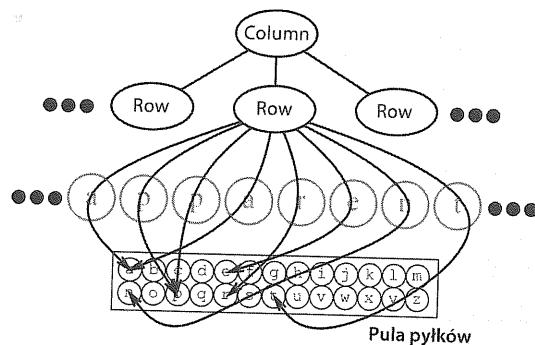
zapisywany w pyłku. Składa się z informacji niezależnych od kontekstu pyłku, co umożliwia jego współużytkowanie. Stan zewnętrzny jest zależny od kontekstu i zmienia się wraz z nim, dlatego nie można go współużytkować. Obiekty klienckie odpowiadają za przekazanie do pyłku zewnętrznego stanu, kiedy jest on potrzebny.

Pyłki są modelem elementów lub jednostek, których zwykle jest za dużo, aby można przedstawić je za pomocą obiektów. Na przykład w edytorze dokumentów można utworzyć pyłki przedstawiające wszystkie litery alfabetu. Każdy pyłek przechowuje kod znaku, jednak współrzędne jego pozycji w dokumencie i styl typograficzny można określić za pomocą algorytmów odpowiedzialnych za układ tekstu i poleceń formatujących, zastosowanych w miejscu, gdzie dany znak się znajduje. Kod znaku jest stanem wewnętrznym, natomiast pozostałe informacje są zewnętrzne.

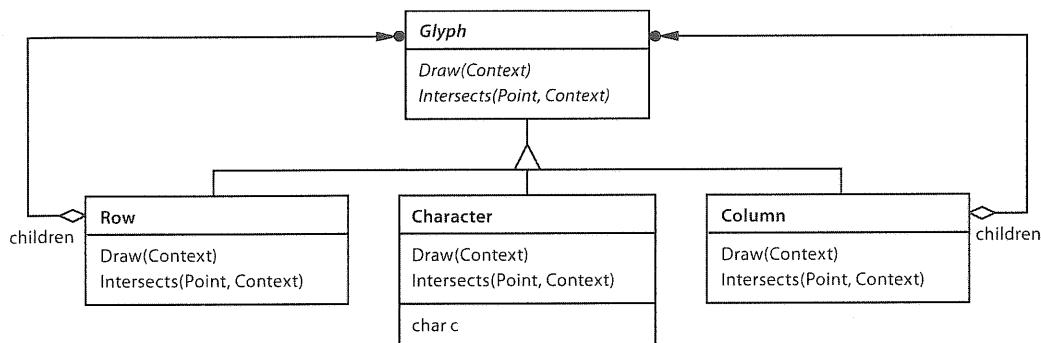
Logicznie każdemu wystąpieniu znaku w dokumencie odpowiada obiekt:



Jednak fizycznie dla znaków każdego rodzaju tworzony jest jeden wspólnie użytkowany obiekt-pyłek pojawiający się w różnych kontekstach w strukturze dokumentu. Każde wystąpienie obiektu reprezentującego określony znak prowadzi do tego samego egzemplarza we wspólnie użytkowanej puli obiektów-pyłków:



Poniżej przedstawiamy strukturę klas tych obiektów. **Glyph** to klasa abstrakcyjna obiektów graficznych (niektóre z nich to pyłki). Operacje, które mogą zależeć od zewnętrznego stanu, otrzymują go jako parametr. Na przykład operacje **Draw** i **Intersects** muszą znać kontekst, w jakim znajduje się glif, zanim będą mogły wykonać swoje zadania.



Pyłek reprezentujący literę „a” przechowuje tylko jej kod. Nie musi obejmować lokalizacji znaku ani nazwy czcionki. To klienci podają powiązane z kontekstem informacje potrzebne pyłkowi do wyświetlenia swojej reprezentacji. Na przykład glif klasy Row potrafi ustalić, w których miejscach powinny wyświetlić się jego elementy podrzędne, aby były uporządkowane poziomo. Dlatego może przekazać do każdego z tych elementów jego lokalizację w żądaniu wyświetlenia się.

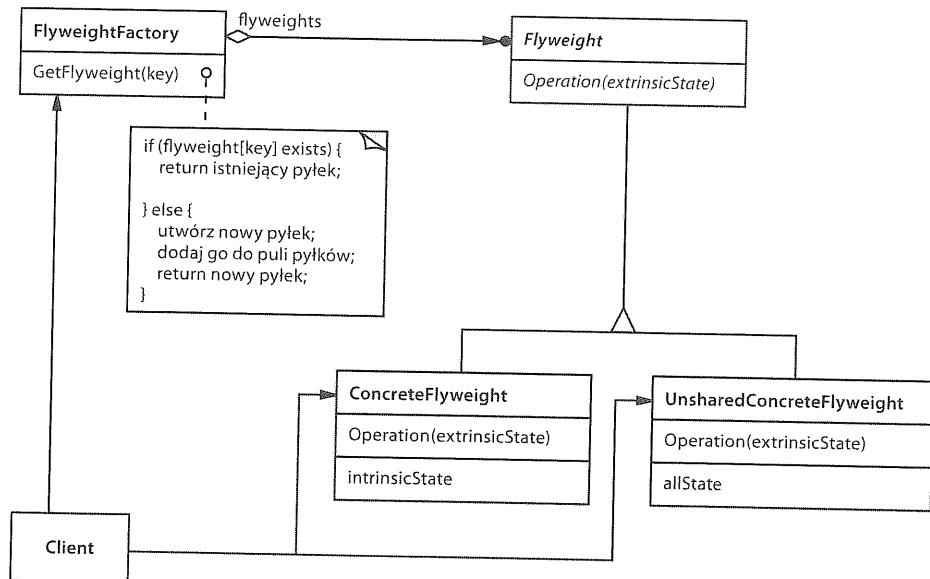
Ponieważ liczba obiektów reprezentujących różne znaki jest znacznie mniejsza od liczby znaków w dokumencie, łączna liczba obiektów jest zdecydowanie niższa niż przy naiwnej implementacji. W dokumencie, w którym wszystkie znaki mają tę samą czcionkę i identyczny kolor, niezależnie od jego długości trzeba utworzyć około 100 obiektów reprezentujących znaki (jest to w przybliżeniu wielkość zbioru znaków ASCII). Ponieważ w większości plików stosowanych jest nie więcej niż 10 różnych kombinacji czcionki i koloru, liczba ta w rzeczywistości nie będzie znacząco większa. Dlatego stosowanie abstrakcji obiektowej dla poszczególnych rodzajów znaków okazuje się praktycznym rozwiązaniem.

WARUNKI STOSOWANIA

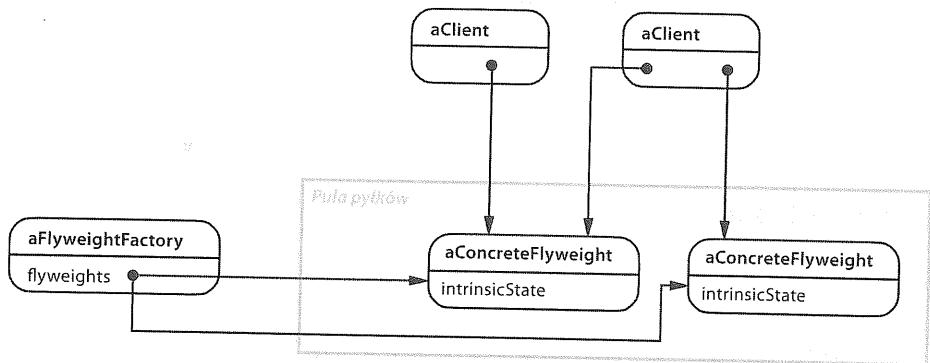
Skuteczność wzorca Pyłek zależy w dużym stopniu od tego, jak i gdzie jest używany. Należy go stosować, jeśli spełnione są *wszystkie* poniższe warunki.

- ▶ Aplikacja korzysta z dużej liczby obiektów.
- ▶ Koszty przechowywania obiektów są wysokie z uwagi na samą ich liczbę.
- ▶ Większość stanu obiektów można zapisać poza nimi.
- ▶ Po przeniesieniu stanu na zewnątrz wiele grup obiektów można zastąpić stosunkowo niewielkimi obiektami współużytkowanymi.
- ▶ Aplikacja nie jest zależna od tożsamości obiektów. Ponieważ obiekty-pyłki można wspólnie użytkować, testy identyczności zwrócą wartość true dla w istocie odrebnego obiektów.

STRUKTURA



Poniższy diagram ilustruje współużytkowanie pyłków.



ELEMENTY

- ▶ **Flyweight (Glyph)**, czyli pyłek:
 - obejmuje deklarację interfejsu, przez który pyłki mogą otrzymać zewnętrzny stan i wykorzystać go do działania.
- ▶ **ConcreteFlyweight (Character)**, czyli pyłek konkretny:
 - obejmuje implementację interfejsu i przechowuje stan wewnętrzny (jeśli jest potrzebny). Obiekty `ConcreteFlyweight` muszą umożliwiać współużytkowanie. Zapisany w nich stan musi być wewnętrzny (nie może zależeć od kontekstu działania takich obiektów).

- ▶ **UnsharedConcreteFlyweight** (*Row*, *Column*), czyli niewspółużytkowany pyłek konkretny:
 - nie wszystkie podklasy klasy Flyweight muszą być współużytkowane. Interfejs klasy Flyweight *umożliwia* współużytkowanie, ale go nie wymusza. Obiekty *UnsharedConcreteFlyweight* na pewnym poziomie struktury obiektów często mają obiekty podrzędne z rodziny *ConcreteFlyweight* (dotyczy to na przykład klas *Row* i *Column*).
- ▶ **FlyweightFactory**, czyli fabryka pyłków:
 - tworzy obiekty-pyłki i zarządza nimi;
 - gwarantuje, że pyłki są prawidłowo współużytkowane. Jeśli klient zażąda pyłku, obiekt *FlyweightFactory* udostępní istniejący egzemplarz lub — jeżeli takiego egzemplarza nie ma — utworzy nowy.
- ▶ **Client**:
 - przechowuje referencje do pyłków;
 - oblicza lub przechowuje zewnętrzny stan pyłków.

WSPÓŁDZIAŁANIE

- ▶ Stan potrzebny pyłkom do działania trzeba podzielić na wewnętrzny i zewnętrzny. Stan wewnętrzny przechowują obiekty *ConcreteFlyweight*. Stan wewnętrzny jest zapisywany przez obiekty *Client* lub przez nie obliczany. Klienci przekazują ten stan do pyłku w momencie wywoływania jego operacji.
- ▶ Klienci nie powinny bezpośrednio tworzyć egzemplarzy klas *ConcreteFlyweight*. Muszą otrzymywać je wyłącznie od obiektu *FlyweightFactory*, co gwarantuje prawidłowe współużytkowanie obiektów *ConcreteFlyweight*.

KONSEKWENCJE

Korzystanie z pyłków może w czasie wykonywania programu spowodować koszty związane z przenoszeniem, wyszukiwaniem i (lub) obliczaniem stanu zewnętrznego, zwłaszcza jeśli był on wcześniej zapisany jako stan wewnętrzny. Jednak wyższa od tych kosztów jest oszczędność pamięci rosnąca wraz ze współużytkowaniem większej liczby pyłków.

Oszczędność pamięci zależy od kilku czynników:

- ▶ zmniejszenia łącznej liczby egzemplarzy, co jest wynikiem współużytkowania;
- ▶ wielkości stanu wewnętrznego na jeden obiekt;
- ▶ tego, czy stan wewnętrzny jest obliczany czy zapisywany.

Im więcej pyłków jest współużytkowanych, tym większa oszczędność pamięci. Rośnie ona także wraz z wielkością współużytkowanego stanu. Największe oszczędności można uzyskać, kiedy obiekty mają rozbudowany stan wewnętrzny i zewnętrzny, a ten ostatni można obliczać, zamiast przechowywać. Wtedy zyski związane z pamięcią są dwójakie — współużytkowanie zmniejsza koszt przechowywania stanu wewnętrznego, a w przypadku stanu zewnętrznego koszty związane z pamięcią są zamieniane na czas procesora.

Wzorzec Pyłek często stosowany jest razem ze wzorcem Kompozyt (s. 170) do reprezentowania struktury hierarchicznej jako grafu ze współużytkowanymi węzłami-liściami. Z uwagi na współużytkowane węzły-liście (czyli pyłki) nie mogą przechowywać wskaźnika do obiektu nadrzednego. Zamiast tego wskaźnik ten jest przekazywany do pyłku jako część stanu zewnętrznego. Ma to istotny wpływ na komunikowanie się ze sobą obiektów w takiej hierarchii.

IMPLEMENTACJA

W czasie implementowania wzorca Pyłek należy rozważyć poniższe kwestie:

1. *Usuwanie stanu zewnętrznego.* Możliwość zastosowania wzorca zależy w dużym stopniu od tego, jak łatwo można określić stan zewnętrznego i usunąć go ze współużytkowanych obiektów. Pozbycie się stanu zewnętrznego nie pomoże w obniżeniu kosztów przechowywania, jeśli istnieje tyle różnych stanów, co obiektów przed zastosowaniem współużytkowania. W idealnych warunkach stan zewnętrzny można obliczyć na podstawie zewnętrznej struktury obiektowej, wymagającej znacznie mniejszej ilości pamięci.

W omawianym edytorze dokumentów można na przykład przechowywać mapę informacji typograficznych w odrębnej strukturze, zamiast zapisywać czcionkę i styl pisma w każdym obiekcie reprezentującym znak. Taka mapa pozwala śledzić grupy znaków o identycznych atrybutach typograficznych. W ramach procesu przechodzenia programu po wyświetlanych elementach znak otrzymuje w momencie wyświetlania się atrybuty typograficzne. Ponieważ w dokumentach zwykle stosuje się niedużą liczbę czcionek i stylów pisma, zapisanie tych informacji poza poszczególnymi obiektami reprezentującymi znaki jest znacznie wydajniejsze od przechowywania tych danych wewnętrznie.

2. *Zarządzanie współużytkowanymi obiektami.* Ponieważ obiekty są współużytkowane, klienci nie powinny tworzyć ich bezpośrednio. Klasa `FlyweightFactory` umożliwia klientom znalezienie określonego pyłku. Obiekty `FlyweightFactory` często korzystają ze struktur asocjacyjnych, aby umożliwić klientom wyszukiwanie potrzebnych pyłków. Fabryka pyłków w przykładowym edytorze dokumentów może przechowywać tablicę pyłków indeksowaną kodami znaków. Menedżer powinien na podstawie otrzymanego kodu zwracać odpowiedni pylek (tworząc go, jeśli jeszcze nie istnieje).

Możliwość współużytkowania oznacza, że trzeba zastosować określone metody zliczania referencji i przywracania pamięci, związane z odzyskiwaniem pamięci zajmowanej przez niepotrzebne już pyłki. Jednak żaden z tych mechanizmów nie jest niezbędnym, jeśli liczba pyłków jest stała i niewielka (na przykład reprezentują one zbiór znaków ASCII). Wtedy warto trwale przechowywać pyłki.

PRZYKŁADOWY KOD

Wróćmy do przykładu dotyczącego formatowania dokumentów. Możemy zdefiniować klasę bazową `Glyph` dla graficznych obiektów-pyłków. Logicznie glify to obiekty złożone (zobacz wzorzec Kompozyt, s. 170) mające atrybuty graficzne i potrafiące się wyświetlić. W tym miejscu koncentrujemy się wyłącznie na atrybutie reprezentującym czcionkę, jednak to samo podejście można zastosować dla dowolnych innych właściwości graficznych glifów.

```

class Glyph {
public:
    virtual ~Glyph();

    virtual void Draw(Window*, GlyphContext&);

    virtual void SetFont(Font*, GlyphContext&);
    virtual Font* GetFont(GlyphContext&);
    virtual void First(GlyphContext&);
    virtual void Next(GlyphContext&);
    virtual bool IsDone(GlyphContext&);
    virtual Glyph* Current(GlyphContext&);

    virtual void Insert(Glyph*, GlyphContext&);
    virtual void Remove(GlyphContext&);

protected:
    Glyph();
};


```

Podklasa `Character` służy wyłącznie do przechowywania kodów znaków.

```

class Character : public Glyph {
public:
    Character(char);

    virtual void Draw(Window*, GlyphContext&);

private:
    char _charcode;
};


```

Aby uniknąć przydzielania w każdym glifie pamięci na atrybut reprezentujący czcionkę, zapiszemy go zewnętrznie — w obiekcie `GlyphContext`. Obiekt ten działa jak pamięć z zewnętrznym stanem. Przechowuje zwięzłe odwzorowania między glifami i czcionkami (oraz dowolnymi innymi atrybutami graficznymi glifów) obowiązujące w różnych kontekstach. Egzemplarz klasy `GlyphContext` jest przekazywany jako parametr do każdej operacji, która potrzebuje informacji o czcionce glifu w danym miejscu. Następnie taka operacja może zażądać od obiektu `GlyphContext` podania czcionki stosowanej w tym kontekście (zależy on od miejsca glifu w strukturze glifów). Dlatego operacje przechodzące po elementach podrzędnych obiektów `Glyph` i manipulujące nimi muszą przy każdym wywołaniu aktualizować obiekty `GlyphContext`.

```

class GlyphContext {
public:
    GlyphContext();
    virtual ~GlyphContext();

    virtual void Next(int step = 1);
    virtual void Insert(int quantity = 1);

    virtual Font* GetFont();
    virtual void SetFont(Font*, int span = 1);

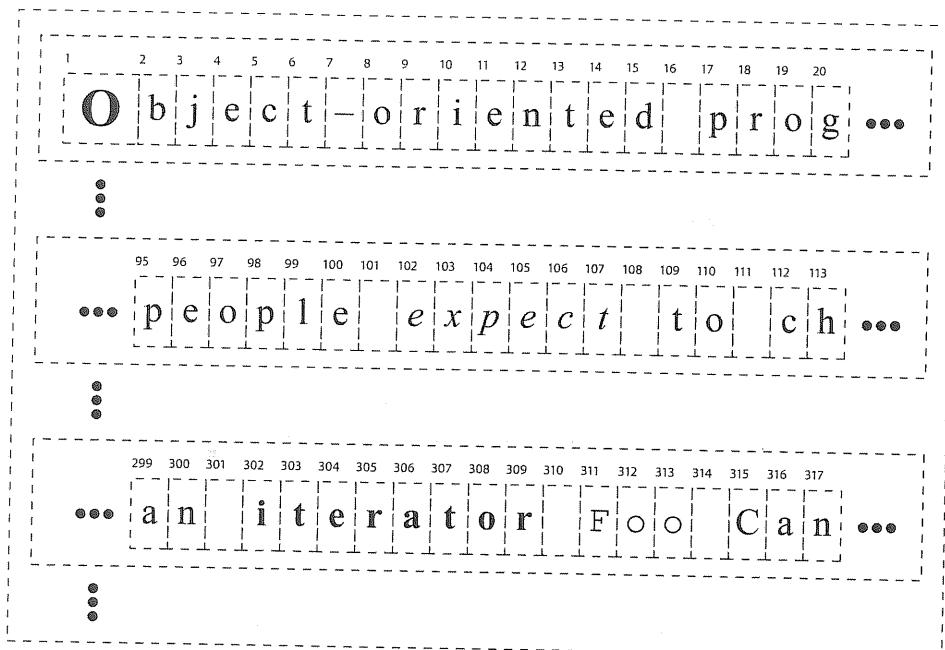
private:
    int _index;
    BTTree* _fonts;
};


```

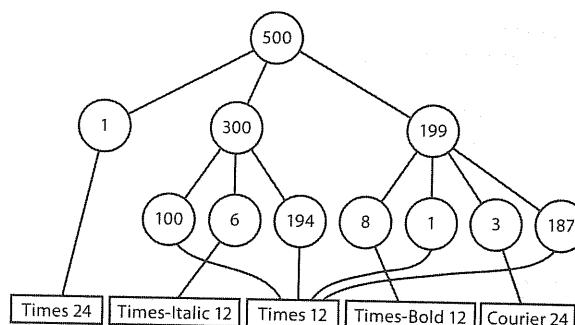
Obiekt `GlyphContext` trzeba informować o bieżącej pozycji w strukturze glifów podczas przechodzenia po niej. Operacja `GlyphContext::Next` zwiększa w tym procesie wartość zmiennej `_index`. W podklasach `Glyph` mających elementy podrzędne (na przykład w podklasach `Row` i `Column`) trzeba zaimplementować operację `Next` w taki sposób, aby na każdym etapie przechodzenia po takich elementach wywoływała operację `GlyphContext::Next`.

W operacji `GlyphContext::GetFont` indeks wykorzystano jako klucz struktury `BTree` przechowującej odwzorowanie z glifów na czcionki. Każdy węzeł w drzewie `BTree` ma zapisaną długość łańcucha znaków, dla którego zwraca informacje o czcionce. Liście w tym drzewie określają czcionkę, natomiast węzły wewnętrzne dzielą łańcuch znaków na podłańcuchy (po jednym na każdy element podrzędny).

Rozważmy następujący fragment ze złożenia glifów.



Struktura `BTree` z informacjami o czcionce może wyglądać tak:

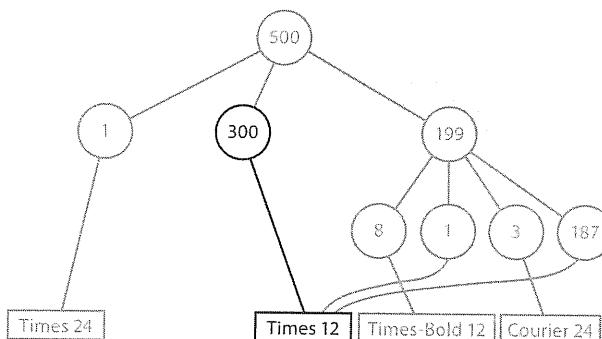


Węzły wewnętrzne definiują przedziały indeksów glifów. Drzewo BTTree jest aktualizowane w odpowiedzi na zmianę czcionki oraz po dodaniu lub usunięciu glifów w ich strukturze. Założymy na przykład, że program doszedł w czasie przechodzenia do indeksu 102. Poniższy kod przypisuje do każdego znaku w słowie „expect” czcionkę otaczającego je tekstu (czyli obiekt times12 — egzemplarz klasy Font reprezentujący czcionkę Times Roman o rozmiarze 12).

```
GlyphContext gc;
Font* times12 = new Font("Times-Roman-12");
Font* timesItalic12 = new Font("Times-Italic-12");
// ...

gcSetFont(times12, 6);
```

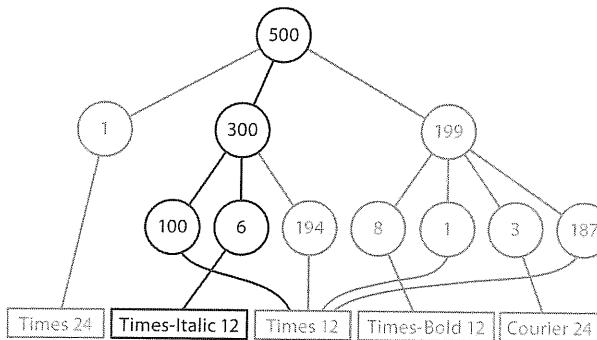
Teraz struktura drzewa BTTree wygląda tak (zmiany wyróżniono kolorem czarnym):



Załóżmy, że przed słowem „expect” dodajemy wyraz „don’t” (wraz z końcowym odstępem). Poniższy kod informuje obiekt gc o tym zdarzeniu (przyjmujemy, że program nadal znajduje się pod indeksem 102):

```
gc.Insert(6);
gcSetFont(timesItalic12, 6);
```

Struktura drzewa BTTree będzie teraz wyglądać w następujący sposób:



Jeśli obiekt `GlyphContext` otrzyma żądanie określenia czcionki aktualnie przetwarzanego glifu, zacznie przechodzić w dół drzewa `BTTree`, dodając napotkane po drodze indeksy, aż znajdzie czcionkę dla bieżącego indeksu. Ponieważ zmiany czcionki są stosunkowo rzadkie, drzewo jest niewielkie w porównaniu do struktury glifów. Pozwala to utrzymać koszty przechowywania na niskim poziomie bez nadmiernego wydłużania czasu wyszukiwania⁶.

Ostatni potrzebny obiekt to `FlyweightFactory`. Jego zadanie to tworzenie glifów i zapewnianie ich prawidłowego współużytkowania. Klasa `GlyphFactory` tworzy egzemplarze klasy `Character` i glify innych rodzajów. W tym programie współużytkujemy tylko obiekty `Character`. Liczba glifów złożonych jest tu znacznie mniejsza, a ważne informacje o ich stanie (czyli o elementach podrzędnych) i tak są wewnętrzne.

```
const int NCHARCODES = 128;

class GlyphFactory {
public:
    GlyphFactory();
    virtual ~GlyphFactory();
    virtual Character* CreateCharacter(char);
    virtual Row* CreateRow();
    virtual Column* CreateColumn();
// ...
private:
    Character* _character[NCHARCODES];
};
```

Tablica `_character` obejmuje wskaźniki do glifów `Character`, a jej indeksy to kody znaków. W konstruktorze tablica ta jest inicjowana wartością zero.

```
GlyphFactory::GlyphFactory () {
    for (int i = 0; i < NCHARCODES; ++i) {
        _character[i] = 0;
    }
}
```

Operacja `CreateCharacter` wyszukuje znak wśród glifów znaków w tablicy i zwraca odpowiedni glif (jeśli taki istnieje). Jeżeli takiego glifu nie ma, operacja tworzy go, umieszcza w tablicy i zwraca.

```
Character* GlyphFactory::CreateCharacter (char c) {
    if (!_character[c]) {
        _character[c] = new Character(c);
    }
    return _character[c];
}
```

⁶ Czas wyszukiwania w tym rozwiążaniu jest proporcjonalny do częstotliwości modyfikowania czcionki. Wydajność będzie najwyższa, jeśli czcionka będzie się zmieniać co znak, jednak w praktyce zdarza się to bardzo rzadko.

Inne operacje po prostu tworzą nowy obiekt przy każdym wywołaniu, ponieważ glify nieznanowe nie są współużytkowane.

```
Row* GlyphFactory::CreateRow () {
    return new Row;
}

Column* GlyphFactory::CreateColumn () {
    return new Column;
}
```

Można pominąć te operacje i umożliwić klientom bezpośrednie tworzenie niewspółużytkowanych glifów. Jednak jeśli później zdecydujemy się na umożliwienie współużytkowania tych glifów, będziemy musieli zmodyfikować kod generujących je klientów.

ZNANE ZASTOSOWANIA

Wykorzystanie obiektów-pylków jako techniki projektowej po raz pierwszy opisano i zbadano przy tworzeniu pakietu InterViews 3.0 [CL90]. Jej autorzy w celu udowodnienia przydatności tego rozwiązania stworzyli rozbudowany edytor dokumentów — Doc [CL92]. W edytorze tym obiekty glifów reprezentują każdy znak w dokumencie. Edytor tworzy tylko jeden egzemplarz klasy `Glyph` dla każdego znaku o określonym stylu (wyznacza on atrybuty graficzne znaków). Dlatego wewnętrzny stan znaku składa się z kodu i informacji o stylu (indeksu tablicy stylów)⁷. Oznacza to, że zewnętrznie przechowywana jest tylko pozycja, dzięki czemu edytor Doc działa szybko. Do reprezentowania dokumentów służy klasa `Document`, która jednocześnie pełni funkcję fabryki pylków. Pomiary dotyczące edytora Doc wykazały, że współużytkowanie znaków w postaci pylków to stosunkowo wydajna technika. W standardowych warunkach dokument zawierający 180 000 znaków wymaga przydzielenia pamięci tylko dla 480 obiektów reprezentujących znaki.

W platformie ET++ [WGM88] pylki zapewniają niezależność od wyglądu i działania⁸. Standard wyglądu i działania wpływa na układ elementów interfejsu użytkownika (na przykład pasków przewijania, przycisków i menu, nazywanych ogólnie widgetami) i ich ozdobników (takie jak cienie i ukośne ramki). Widgety delegują wszystkie zadania związane z układem i wyświetleniem do odrębnego obiektu `Layout`. Podmiana obiektu `Layout` umożliwia zmianę wyglądu i działania widgetów nawet w czasie wykonywania programu.

Dla każdej klasy reprezentującej widget istnieje odpowiednia klasa `Layout` (na przykład `ScrollbarLayout`, `MenubarLayout` itd.). Oczywisty problem związany z tym podejściem polega na tym, że wykorzystanie odrębnych obiektów do określania układu podwaja liczbę obiektów interfejsu użytkownika, ponieważ dla każdego standardowego obiektu potrzebny jest obiekt z rodziny `Layout`. Aby uniknąć tych kosztów, obiekty `Layout` zaimplementowano jako pylki. Obiekty te dobrze nadają się na pylki, ponieważ służą głównie do definiowania zachowania i łatwo jest przekazać do nich niewielką ilość stanu zewnętrznego potrzebną do określenia układu obiektu lub jego wyświetlenia.

⁷ We wcześniejszym punkcie „Przykładowy kod” informacje o stylu są przechowywane zewnętrznie, przez co stan wewnętrzny obejmuje tylko kod znaku.

⁸ Inny sposób na zapewnianie niezależności od stylu i działania to zastosowanie wzorca Fabryka abstrakcyjna (s. 87).

Za tworzenie obiektów Layout i zarządzanie nimi odpowiadają obiekty Look. Klasa Look to fabryka abstrakcyjna (s. 101) pobierająca określony obiekt Layout za pomocą takich operacji, jak GetButtonLayout, GetMenuBarLayout itd. Dla każdego standardu wyglądu i stylu istnieje odpowiednia podklasa klasy Look (na przykład MotifLook lub OpenLook) udostępniająca właściwe obiekty Layout.

Przy okazji warto wspomnieć, że obiekty Layout są w istocie strategiami (zobacz wzorzec Strategia, s. 321). Stanowią one przykład obiektów strategii zaimplementowanych jako pyłki.

POWIĄZANE WZORCE

Wzorzec Pyłek często stosuje się razem ze wzorcem Kompozyt (s. 170) w celu zaimplementowania logicznie hierarchicznej struktury za pomocą acyklicznego grafu skierowanego ze wspólnymi węzłami-liśćmi.

Obiekty stanu (s. 312) i strategii (s. 321) często najlepiej jest implementować jako pyłki.

OMÓWIENIE WZORCÓW STRUKTURALNYCH

Możliwe, że zauważysz podobieństwa między poszczególnymi wzorcami strukturalnymi, zwłaszcza pod względem elementów i współdziałania. Prawdopodobnie wynika to z tego, że wzorce strukturalne oparte są na tym samym małym zestawie mechanizmów języka przeznaczonych do strukturyzowania kodu i obiektów — dziedziczeniu zwykłym i wielodziedziczeniu w przypadku wzorców klasowych oraz składaniu obiektów we wzorcach obiektowych. Jednak za tymi podobieństwami kryją się różne funkcje poszczególnych wzorców. W tym punkcie porównujemy grupy wzorców strukturalnych, aby przedstawić różnicujące je zalety.

ADAPTER I MOST

Wzorce Adapter (s. 141) i Most (s. 181) mają kilka cech wspólnych. Oba zwiększą elastyczność, ponieważ wprowadzają poziom pośredniości w komunikacji z innym obiektem. Ponadto oba polegają na przekazywaniu żądań do takiego obiektu za pomocą interfejsu różnego od wyjściowego.

Kluczowa różnica między tymi wzorcami związana jest z ich przeznaczeniem. Adapter ma przede wszystkim niwelować niezgodności między dwoma istniejącymi interfejsami. Najważniejsze nie są tu implementacje tych interfejsów lub sposoby ich niezależnego modyfikowania. Adapter umożliwia współdziałanie dwóch niezależnie zaprojektowanych klas bez konieczności ponownego implementowania jednej z nich. Natomiast Most łączy abstrakcję i jej (potencjalnie liczne) implementacje. Zapewnia stabilny interfejs klientom, a przy tym umożliwia modyfikowanie klas z jego implementacją. Most pozwala też dodawać nowe implementacje wraz z rozwijaniem systemu.

Z uwagi na te różnice wzorce Adapter i Most często stosuje się na różnych etapach cyklu rozwijania oprogramowania. Adapter nieraz jest konieczny, kiedy programista odkryje, że dwie niezgodne klasy powinny ze sobą współdziałać. Wykorzystanie adaptera zwykle ma na celu uniknięcie powielania kodu. Powiązanie między klasami jest wtedy wykrywane, kiedy są już one gotowe. Z kolei użytkownik wzorca Most od początku wie, że abstrakcja musi mieć kilka implementacji i że oba te elementy mogą być modyfikowane niezależnie od siebie. Wzorzec Adapter pozwala zapewnić współdziałanie klas *po* ich zaprojektowaniu, a Most — *przed* utworzeniem projektu. Nie oznacza to, że Adapter jest pod jakimś względem gorszy od Mostu. Po prostu każdy z tych wzorców rozwiązuje inny problem.

Niektórzy mogą uznać fasadę (zobacz wzorzec Fasada, s. 161) za adapter dla grupy innych obiektów. Jednak takie podejście nie uwzględnia faktu, że fasada definiuje *nowy* interfejs, natomiast adapter ponownie wykorzystuje stary. Warto pamiętać, że adapter umożliwia współdziałanie dwóch *istniejących* interfejsów bez definiowania nowego.

KOMPOZYT, DEKORATOR I PEŁNOMOCNIK

Diagramy strukturalne wzorców Kompozyt (s. 170) i Dekorator (s. 152) są podobne do siebie. Dzieje się tak, ponieważ w obu tych wzorcach zastosowano rekurencyjne składanie do uporządkowania dowolnej liczby obiektów. Ta wspólna cecha może skłaniać do traktowania obiektu dekoratora jak uproszczonego kompozytu, jednak nie jest to zgodne z celem stosowania

wzorca Dekorator. Podobieństwa kończą się na rekurencyjnym składaniu, a różnicą także tu jest inne przeznaczenie wzorców.

Dekorator ma umożliwiać dodawanie zadań do obiektów bez tworzenia podklas. Pozwala to uniknąć wzrostu liczby podklas, co może mieć miejsce przy próbie statycznego uwzględnienia każdej kombinacji zadań. Kompozyty są przeznaczone do czegoś innego. Służą głównie do porządkowania klas, aby umożliwić jednolite traktowanie wielu powiązanych obiektów i używanie wielu obiektów jak jednego. Najważniejsze są tu nie ozdobniki, ale reprezentacja.

Wzorce te mają odmienne przeznaczenie, ale się uzupełniają. Powoduje to, że wzorce Kompozyt i Dekorator często stosuje się razem. Oba prowadzą do powstania projektu, który umożliwia rozwijanie aplikacji przez łączenie obiektów bez konieczności definiowania nowych klas. Taki projekt obejmuje klasę abstrakcyjną. Część jej podklas to kompozyty, inne to dekoratory, a jeszcze inne obejmują implementację podstawowych bloków konstrukcyjnych systemu. W tym przypadku kompozyty i dekoratory będą miały wspólny interfejs. W kontekście wzorca Dekorator kompozyt to obiekt `ConcreteComponent`. Z perspektywy wzorca Kompozyt dekorator jest obiektem `Leaf`. Oczywiście wzorców tych nie *trzeba* używać razem i — jak pokazaliśmy — są one przeznaczone do czego innego.

Innym wzorcem o strukturze podobnej do Dekoratora jest Pełnomocnik (s. 191). Oba te wzorce opisują, jak dodać poziom pośredniości w komunikacji z obiektem, a w implementacjach tych wzorców przechowywana jest referencja do innego obiektu, do którego przekazywane są żądania. Jednak także te wzorce mają różne funkcje.

Wzorzec Pełnomocnik — podobnie jak Dekorator — wymaga złożenia obiektu i zapewnia klientom identyczny interfejs. Jednak inaczej niż we wzorcu Dekorator nie ma tu znaczenia dynamiczne dołączanie lub odłączanie właściwości. Ponadto wzorzec Pełnomocnik nie jest zaprojektowany pod kątem rekurencyjnego składania. Ma jedynie zapewniać zastępstwo dla podmiotu, kiedy bezpośredni dostęp do niego jest niewygodny lub niepożądany (na przykład podmiot działa na zdalnej maszynie, jest obiektem trwałym lub dostęp do niego jest ograniczony).

We wzorcu Pełnomocnik obiekt definiuje kluczowe funkcje, a pełnomocnik umożliwia (lub blokuje) dostęp do niego. We wzorcu Dekorator komponent udostępnia tylko część funkcji, a dekoratory obsługują pozostałe. Jest to przydatne w warunkach, kiedy w czasie komplilacji nie można ustalić wszystkich funkcji obiektu, a przynajmniej nie można tego zrobić w wygodny sposób. To nieokreślenie powoduje, że rekurencyjne składanie to kluczowy aspekt wzorca Dekorator. We wzorcu Pełnomocnik jest inaczej, ponieważ najważniejsza jest tu jedna relacja — między pełnomocnikiem i powiązanym z nim obiektem. Relację tę można przedstawić statycznie.

Te różnice mają duże znaczenie, ponieważ dotyczą rozwiązań specyficznych powtarzających się problemów w projektowaniu obiektowym. Jednak nie oznacza to, że opisanych wzorców nie można łączyć ze sobą. Wyobraź sobie pełnomocnik-dekorator, który dodaje funkcje do pełnomocnika, lub dekorator-pełnomocnik ozdabiający zdalny obiekt. Choć takie hybrydy mogą okazać się przydatne (nie znamy żadnych przykładów z praktyki), można je rozdzielić na wzorce, które są użyteczne.

Rozdział 5.

Wzorce operacyjne

Wzorce operacyjne dotyczą algorytmów i podziału zadań między obiekty. Opisują nie tylko modele obiektów i klas, ale też modele komunikacji między nimi. Wzorce operacyjne określają złożone przepływy sterowania, które trudno jest śledzić w czasie wykonywania programu. Pozwala to skoncentrować się na powiązaniach między obiekttami, a nie na przepływie sterowania.

Klasowe wzorce operacyjne polegają na stosowaniu dziedziczenia do podziału zachowań między klasy. W tym rozdziale opisujemy dwa takie wzorce. Metoda szablonowa (s. 264) to prostszy i częściej spotykany z nich. Taka metoda jest abstrakcyjną definicją algorytmu. Określa ona działanie algorytmu krok po kroku. Każdy etap polega na wywołaniu abstrakcyjnej lub prostej operacji. W podklasach algorytm jest uzupełniany o definicje operacji abstrakcyjnych. Drugi klasowy wzorzec operacyjny to Interpreter (s. 217). Dotyczy on reprezentowania gramatyki jako hierarchii klas. Interpreter jest tu implementowany jako operacja egzemplarzy klas z tej hierarchii.

W obiektowych wzorcach operacyjnych zamiast dziedziczenia stosuje się składanie obiektów. Niektóre z tych wzorców opisują współdziałanie grup równorzędnych obiektów przy wykonywaniu zadań, których żaden z obiektów nie potrafi ukończyć samodzielnie. Ważnym zagadnieniem jest tu dowiadывanie się równorzędnych obiektów o sobie. Takie obiekty mogą przechowywać referencje do siebie, jednak zwiększa to powiązanie między nimi. W krańcowym przypadku każdy obiekt wie o istnieniu wszystkich pozostałych. Wzorzec Mediator (s. 254) pozwala tego uniknąć przez wprowadzenie obiektu mediatora pomiędzy równorzędnymi obiektami. Taki mediator zapewnia pośredniość potrzebną do uzyskania luźnego powiązania.

Wzorzec Łańcuch zobowiązań (s. 244) umożliwia otrzymywanie jeszcze luźniejszego powiązania. Pozwala przesyłać żądania do obiektu pośrednio — poprzez łańcuch obiektów-kandydatów. W zależności od warunków w czasie wykonywania programu żądanie może zostać obsłużone przez każdego z kandydatów. Ich liczba jest nieokreślona, a obiekty włączane w łańcuch można wybierać w czasie wykonywania programu.

Wzorzec Obserwator (s. 269) opisuje definiowanie i konserwowanie zależności między obiekttami. Klasycznym przykładem zastosowania tego wzorca jest architektura MVC w języku Smalltalk, gdzie wszystkie widoki modelu są powiadamiane o zmianie jego stanu.

Inne obiektowe wzorce operacyjne dotyczą kapsulkowania zachowań w obiekcie i delegowania do niego żądań. Wzorzec Strategia (s. 321) polega na kapsulkowaniu algorytmu w obiekcie. Wzorzec ten ułatwia określanie i zmienianie algorytmu używanego przez obiekt. Wzorzec Polecenie (s. 302) polega na kapsulkowaniu żądania w obiekcie, aby można je przekazać jako parametr, zapisać w historii żądań lub manipulować nim w inny sposób. Wzorzec Stan (s. 312) dotyczy kapsulkowania stanów obiektu, tak aby można zmodyfikować zachowanie obiektu, kiedy zmianie ulegnie obiekt stanu. Wzorzec Odwiedzający (s. 280) opisuje kapsulkowanie zachowania, które w innych warunkach jest rozproszone po wielu klasach, a wzorzec Iterator (s. 230) pozwala abstrakcyjnie ująć sposób dostępu do obiektów w agregacie i przechodzenia po nich.

INTERPRETER (INTERPRETER)

klasowy, operacyjny

RZECZNACZENIE

Określa reprezentację gramatyki języka oraz interpreter, który wykorzystuje tę reprezentację do interpretowania zdań z danego języka.

IZASADNIENIE

Jeśli problem określonego rodzaju pojawia się dostatecznie często, czasem warto zapisać wystąpienia takiego problemu jako zdania w prostym języku. Następnie można zbudować interpreter rozwiązujejący problem przez interpretowanie tych zdań.

Przykładowym często spotykanym problemem jest wyszukiwaniełańcuchów znaków pasujących do wzorca. Standardowym językiem do określania takich wzorców są wyrażenia regularne. Zamiast tworzyć niestandardowe algorytmy porównujące każdy wzorzec złańcuchami znaków, można wykorzystać algorytmy wyszukiwania interpretujące wyrażenie regularne, które określa zbiór pasującychłańcuchów.

Wzorzec Interpreter pokazuje, jak zdefiniować gramatykę prostych języków, przedstawić zdania w tym języku i je interpretować. W omawianym przykładzie wzorzec ten określa, jak zdefiniować gramatykę wyrażeń regularnych, utworzyć reprezentację konkretnego wyrażenia i zinterpretować je.

Założymy, że wyrażenia regularne definiuje poniższa gramatyka:

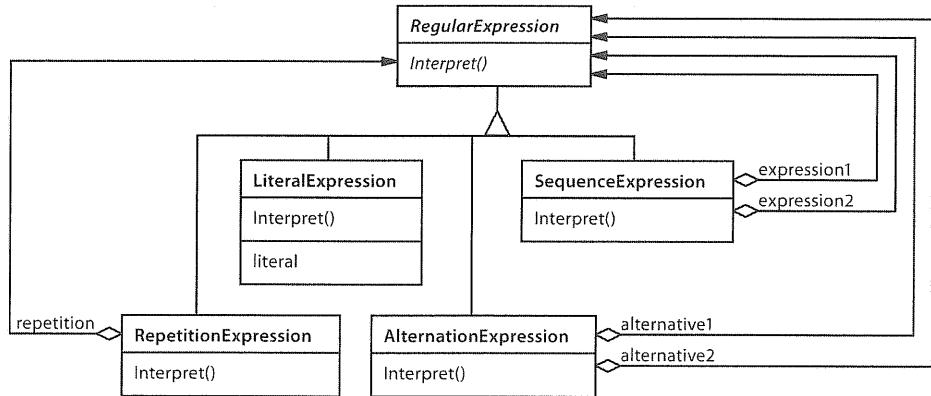
```

expression ::= literal | alternation | sequence | repetition |
              '(' expression ')'
alternation ::= expression '|' expression
sequence ::= expression '&' expression
repetition ::= expression '*'
literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*

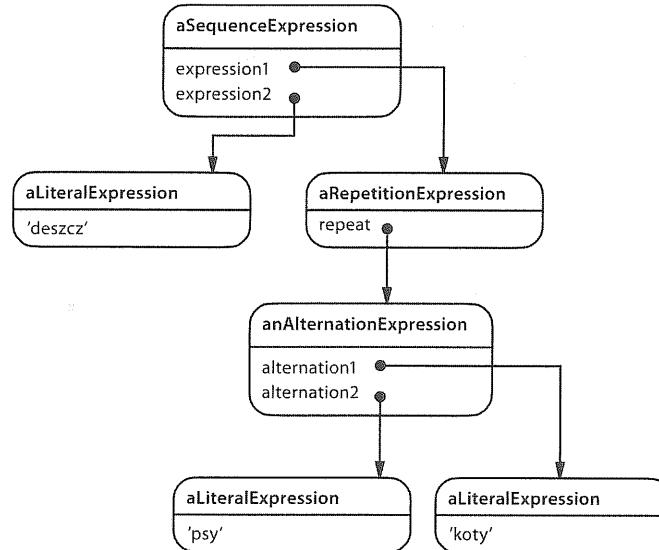
```

Słowo `expression` to symbol początkowy, a `literal` to symbol końcowy do definiowania prostych słów.

We wzorcu Interpreter do reprezentowania poszczególnych reguł gramatyki służą klasy. Symbole po prawej stronie reguły to zmienne egzemplarza z tych klas. Powyższa gramatyka składa się z pięciu klas: klasy abstrakcyjnej `RegularExpression` i czterech jej podklas — `LiteralExpression`, `AlternationExpression`, `SequenceExpression` i `RepetitionExpression`. Trzy ostatnie z tych klas służą do tworzenia zmiennych do przechowywania podwyrażeń.



Każde wyrażenie regularne zdefiniowane za pomocą tej gramatyki jest reprezentowane za pomocą drzewa składni abstrakcyjnej składającego się z egzemplarzy wymienionych klas. Oto przykładowe drzewo składni abstrakcyjnej:



Reprezentuje ono następujące wyrażenie regularne:

`deszcz & (psy | koty) *`

Można utworzyć interpreter takich wyrażeń regularnych przez zdefiniowanie operacji `Interpret` w każdej z podklas klasy `RegularExpression`. Operacja ta przyjmuje jako argument kontekst, w którym należy zinterpretować wyrażenie. Kontekst ten obejmuje wejściowy łańcuch znaków i informacje o tym, jak dużą jego część już dopasowano. Implementacja operacji `Interpret` w każdej z podklas klasy `RegularExpression` dopasowuje na podstawie bieżącego kontekstu następną część wejściowego łańcucha znaków. Na przykład:

- ▶ klasa `LiteralExpression` sprawdza, czy dane wejściowe pasują do literalu określonego w wyrażeniu;
- ▶ klasa `AlternationExpression` sprawdza, czy dane wejściowe pasują do jednego z alternatywnych wyrażeń;
- ▶ klasa `RepetitionExpression` sprawdza, czy dane wejściowe obejmują wiele kopii wyrażenia powtarzanego w danym wyrażeniu;

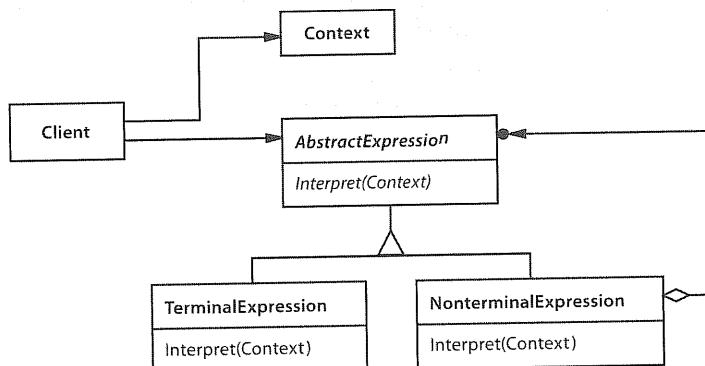
itd.

WARUNKI STOSOWANIA

Wzorzec Interpreter należy stosować, jeśli istnieje interpretowany język, a zdania z tego języka można przedstawić za pomocą drzewa składni abstrakcyjnej. Oto warunki, w których wzorzec ten jest najbardziej przydatny.

- ▶ Gramatyka jest prosta. W przypadku skomplikowanych gramatyk hierarchia klas staje się duża i trudna w zarządzaniu. Wtedy lepszym rozwiązaniem jest zastosowanie innych narzędzi, na przykład generatorów parserów. Parsery mogą interpretować wyrażenia bez budowania drzew składni abstrakcyjnej, co pozwala zmniejszyć ilość potrzebnej pamięci, a czasem także przyspieszyć działanie programu.
- ▶ Wydajność nie jest najważniejsza. Najwydajniejsze interpretery zwykle *nie* interpretują bezpośrednio drzew składni, ale najpierw tłumaczą je na inną postać. Na przykład wyrażenia regularne często są przekształcane na maszyny stanowe. Jednak nawet wtedy *translator* można zaimplementować za pomocą wzorca Interpreter, dlatego i tu jest on przydatny.

STRUKTURA



ELEMENTY

- ▶ **AbstractExpression** (`RegularExpression`), czyli wyrażenie abstrakcyjne:
 - obejmuje deklarację operacji abstrakcyjnej `Interpret` wspólnej wszystkim węzłom drzewa składni abstrakcyjnej.

- ▶ **TerminalExpression** (`LiteralExpression`), czyli wyrażenie końcowe:
 - obejmuje implementację operacji `Interpret` związaną z symbolami końcowymi z danej gramatyki;
 - dla każdego symbolu końcowego w zdaniu trzeba utworzyć egzemplarz tej klasy.
- ▶ **NonterminalExpression** (`AlternationExpression`, `RepetitionExpression`, `SequenceExpression`), czyli wyrażenie pośrednie:
 - dla każdej reguły typu $R ::= R_1R_2\dots R_n$ w gramatyce trzeba utworzyć jedną klasę tego rodzaju;
 - przechowuje zmienne egzemplarza typu `AbstractExpression` dla każdego z symboli od R_1 do R_n ;
 - obejmuje implementację operacji `Interpret` dla symboli pośrednich. Operacja ta zwykle rekurencyjnie wywołuje samą siebie dla zmiennych reprezentujących symbole od R_1 do R_n .
- ▶ **Context:**
 - przechowuje informacje globalne interpretera.
- ▶ **Client:**
 - tworzy (lub otrzymuje) drzewo składni abstrakcyjnej reprezentujące określone zdanie w języku zdefiniowanym przez gramatykę; takie drzewo składa się z egzemplarzy klas `NonterminalExpression` i `TerminalExpression`;
 - wywołuje operację `Interpret`.

WSPÓŁDZIAŁANIE

- ▶ Klient tworzy (lub otrzymuje) zdanie jako drzewo składni abstrakcyjnej obejmujące egzemplarze klas `NonterminalExpression` i `TerminalExpression`. Następnie klient inicjuje kontekst i wywołuje operację `Interpret`.
- ▶ Każdy węzeł klasy `NonterminalExpression` definiuje operację `Interpret` w kategoriach operacji `Interpret` poszczególnych podwyrażeń. Operacja `Interpret` w każdej klasie `TerminalExpression` określa przypadek bazowy w rekurencji.
- ▶ Operacje `Interpret` w każdym węźle wykorzystują kontekst do zapisywania i pobierania stanu interpretera.

KONSEKWENCJE

Wzorzec Interpreter ma następujące zalety i wady:

1. *Modyfikowanie i rozszerzanie gramatyki jest łatwe.* Ponieważ we wzorcu tym do reprezentowania reguł gramatyki służą klasy, można wykorzystać dziedziczenie do zmienienia lub wzbogacenia gramatyki. Istniejące wyrażenia można modyfikować stopniowo, a nowe wyrażenia można zdefiniować jako wersje starych.
2. *Implementowanie gramatyki także jest proste.* Klasy definiujące węzły drzewa składni abstrakcyjnej mają podobne implementacje. Klasy te łatwo jest napisać, a często można zautomatyzować ich generowanie za pomocą generatora kompilatorów lub parserów.

3. *Konserwowanie złożonych gramatyk jest trudne.* We wzorcu Interpreter trzeba zdefiniować przynajmniej jedną klasę dla każdej reguły gramatyki (reguły zdefiniowane za pomocą notacji BNF mogą wymagać wielu klas). Dlatego zarządzanie gramatykami obejmującymi wiele reguł i konserwowanie ich może sprawiać trudności. Aby złagodzić ten problem, można zastosować inne wzorce projektowe (zobacz punkt „Implementacja”). Jednak w przypadku bardzo złożonych gramatyk lepiej jest zastosować inne techniki, na przykład generatory parserów lub kompilatorów.
4. *Możliwość dodawania nowych sposobów interpretowania wyrażeń.* Wzorzec Interpreter ułatwia analizowanie wyrażenia w nowy sposób. Można na przykład dodać obsługę eleganckiego wyświetlanego wyrażenia lub sprawdzania jego typu przez zdefiniowanie nowej operacji w klasach reprezentujących wyrażenia. Jeśli często tworzysz nowe sposoby interpretowania wyrażenia, zastanów się nad zastosowaniem wzorca Odwiedzający (s. 280), aby uniknąć modyfikowania klas gramatyki.

IMPLEMENTACJA

Wiele zagadnień związanych z implementacją dotyczy zarówno wzorca Interpreter, jak i wzorca Kompozyt (s. 170). Poniższe kwestie są specyficzne dla wzorca Interpreter:

1. *Tworzenie drzewa składni abstrakcyjnej.* Wzorzec Interpreter nie określa, jak należy tworzyć drzewa składni abstrakcyjnej. Oznacza to, że nie dotyczy analizy składni. Drzewo składni abstrakcyjnej można utworzyć za pomocą parsera opartego na tablicach, ręcznie opracowanego parsera (zwykle stosującego przechodzenie rekurencyjne) lub bezpośrednio w kliencie.
2. *Definiowanie operacji Interpret.* Operacji Interpret nie trzeba definiować w klasach wyrażeń. Jeśli trzeba często tworzyć nowe interpretery, lepiej jest zastosować wzorzec Odwiedzający (s. 280) i umieścić operację Interpret w obiekcie odwiedzającym. Na przykład gramatyka języka programowania ma wiele operacji wykonywanych na drzewach składni abstrakcyjnej. Operacje te służą do sprawdzania typu, optymalizowania, generowania kodu itd. Korzystniejsze jest zastosowanie wzorca Odwiedzający, co pozwala uniknąć definiowania takich operacji dla każdej klasy gramatyki.
3. *Współużytkowanie symboli końcowych za pomocą wzorca Pytek.* Jeśli w zdaniach danej gramatyki pojawia się wiele wystąpień symbolu końcowego, korzystne może być współużytkowanie pojedynczej kopii tego symbolu. Dobrym przykładem są gramatyki języków programowania. Każda zmienna pojawia się w kodzie w wielu miejscach. W przykładzie z punktu „Uzasadnienie” zdanie może mieć wielokrotnie powtarzający się symbol końcowy pies (jego model określa klasa LiteralExpression).

Węzły końcowe zwykle nie przechowują informacji o ich pozycji w drzewie składni abstrakcyjnej. W czasie interpretowania zdań węzły nadrzędne przekazują do węzłów końcowych potrzebne dane. Dlatego występuje podział na stan współużytkowany (wewnętrzny) oraz przekazywany (zewnętrzny) i można zastosować wzorzec Pytek (s. 201).

Na przykład każdy egzemplarz klasy LiteralExpression reprezentujący słowo pies otrzymuje kontekst, który obejmuje dopasowany do tej pory podłańcuch. Każdy obiekt LiteralExpression wykonuje za pomocą operacji Interpret to samo zadanie — sprawdza, czy następny fragment danych wejściowych obejmuje słowo pies. Nie ma tu znaczenia, w którym miejscu drzewa pojawia się egzemplarz omawianej klasy.

PRZYKŁADOWY KOD

Przedstawiamy tu dwa przykłady. Pierwszy z nich to kompletny program napisany w języku Smalltalk. Służy on do sprawdzania, czy zdanie pasuje do wyrażenia regularnego. Drugi przykład to program w języku C++ przeznaczony do analizy wyrażeń logicznych.

Narzędzie do dopasowywania wyrażeń regularnych sprawdza, czy łańcuch znaków należy do języka zdefiniowanego przez wyrażenie regularne. Wyrażenie regularne jest zdefiniowane za pomocą poniższej gramatyki:

```
expression ::= literal | alternation | sequence | repetition |
              '(' expression ')'
alternation ::= expression ']' expression
sequence ::= expression '&' expression
repetition ::= expression 'repeat'
literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*
```

Ta gramatyka to zmodyfikowana wersja przykładu z punktu „Uzasadnienie”. Zmieniliśmy nieco składnię konkretną wyrażeń regularnych, ponieważ symbol „*” w języku Smalltalk nie może być operacją przyrostkową. Dlatego w zamian zastosowaliśmy słowo repeat. Przyjrzyjmy się przykładowemu wyrażeniu regularnemu:

```
(('pies ' | 'kot ') repeat & 'pogoda')
```

Pasuje ono do wejściowego łańcucha znaków pies pies kot pogoda.

Aby zaimplementować narzędzie do dopasowywania, zdefiniujemy pięć klas opisanych na stronie 243. Klasa SequenceExpression obejmuje zmienne egzemplarza expression1 i expression2. Posługują one do zapisania elementów podległych tej klasy w drzewie składni abstrakcyjnej. W klasie AlternationExpression pasujące alternatywne wyrażenia zapisane są w zmiennych egzemplarza alternative1 i alternative2, natomiast klasa RepetitionExpression przechowuje powtarzające się wyrażenie w zmiennej egzemplarza repetition. Klasa LiteralExpression ma zmienną egzemplarza components przechowującą listę obiektów (zwykle są nimi znaki). Reprezentują one literał znakowy, który musi pasować do wprowadzonego zdania.

Operacja match: obejmuje implementację interpretera wyrażeń regularnych. Każda z klas definiujących drzewo składni abstrakcyjnej musi zawierać implementację tej operacji. Przyjmuję ona argument inputState reprezentujący bieżący stan procesu dopasowywania (po wczytaniu fragmentu wejściowego łańcucha znaków).

Do przedstawienia bieżącego stanu służy zestaw wejściowych strumieni reprezentujących zbiór danych wejściowych, które wyrażenie regularne mogło do danego momentu zaakceptować. W przybliżeniu przypomina to rejestrowanie wszystkich stanów, w których mógłby znajdować się równoważny automat stanów skończonych po przetworzeniu wejściowego strumienia do danego miejsca.

Bieżący stan najważniejszy jest w operacji repeat. Założymy, że wyrażenie regularne wygląda tak:

```
'a' repeat
```

Interpreter dopasuje do niego łańcuchy a, aa, aaa itd. Jeśli wyrażenie ma następującą postać:

```
'a' repeat & 'bc'
```

wtedy interpreter dopasuje łańcuchy abc, aabc, aaabc itd. Jednak jeżeli wyrażenie regularne wygląda tak:

```
'a' repeat & 'abc'
```

wtedy przy dopasowywaniu wejściowego łańcucha aabc do podwyrażenia 'a' repeat wygenerowane zostaną dwa strumienie wejściowe — jeden z pasującym jednym znakiem danych wejściowych i drugi z pasującymi dwoma znakami. Do końcowego fragmentu abc pasował będzie tylko strumień z jednym zaakceptowanym znakiem.

Zastanówmy się teraz nad definicjami operacji `match`: w każdej klasie określającej wyrażenie regularne. Wersja dla klasy `SequenceExpression` dopasowuje po kolei każde z podwyrażeń z tej klasy. Zwykle usuwa też strumienie wejściowe ze zmiennej `inputState`.

```
match: inputState
  ^ expression2 match: (expression1 match: inputState).
```

Klasa `AlternationExpression` zwraca stan składający się z sumy stanów obu alternatywnych wyrażeń. Oto definicja operacji `match`: z tej klasy:

```
match: inputState
  | finalState |
  finalState := alternativel match: inputState.
  finalState addAll: (alternative2 match: inputState).
  ^ finalState
```

Operacja `match`: z klasy `RepetitionExpression` wyszukuje jak najwięcej pasujących stanów:

```
match: inputState
  | aState finalState |
  aState := inputState.
  finalState := inputState copy.
  [aState isEmpty]
    whileFalse:
      [aState := repetition match: aState.
       finalState addAll: aState].
  ^ finalState
```

Stan wyjściowy tej operacji zwykle obejmuje więcej stanów niż jej stan wejściowy, ponieważ klasa `RepetitionExpression` może dopasować różną liczbę (jedno, dwa lub więcej) wystąpień zmiennej `repetition` do stanu wejściowego. Stany wyjściowe reprezentują wszystkie te możliwości, co umożliwia dalszym elementom wyrażenia regularnego określenie, który z tych stanów jest właściwy.

Definicja operacji `match`: z klasy `LiteralExpression` dopasowuje jej komponenty do każdego możliwego strumienia wejściowego. Operacja zachowuje tylko pasujące strumienie wejściowe:

```
match: inputState
  | finalState tStream |
  finalState := Set new.
  inputState
  do:
    [:stream | tStream := stream copy.
     (tStream nextAvailable:
```

```

        components size
    ) = components
        ifTrue: [finalState add: tStream]
].
^ finalState

```

Komunikat `nextAvailable`: powoduje przejście do następnego strumienia wejściowego. Jest to jedyna operacja `match:`, która działa w ten sposób. Warto zauważyć, że zwracany stan obejmuje kopię strumienia wejściowego, co gwarantuje, że dopasowanie literalu nie wywoła zmiany takiego strumienia. Jest to ważne, ponieważ każde z alternatywnych wyrażeń w klasie `AlternationExpression` należy porównać z identyczną kopią strumienia wejściowego.

Po zdefiniowaniu klas składających się na drzewo składni abstrakcyjnej możemy opisać, jak je zbudować. Zamiast tworzyć parser wyrażeń regularnych, zdefiniujemy wybrane operacje klas `RegularExpression`, aby analiza wyrażenia w języku Smalltalk prowadziła do powstania drzewa składni abstrakcyjnej dla odpowiedniego wyrażenia regularnego. Umożliwi to wykorzystanie wbudowanego kompilatora języka Smalltalk w taki sposób, jakby był parserem wyrażeń regularnych.

Aby zbudować drzewo składni abstrakcyjnej, trzeba zdefiniować `|`, `repeat` i `&` jako operacje klasy `RegularExpression`. W klasie tej wymienione operacje zdefiniowaliśmy w następujący sposób:

```

& aNode
    ^ SequenceExpression new
        expression1: self expression2: aNode asRExp

repeat
    ^ RepetitionExpression new repetition: self

| aNode
    ^ AlternationExpression new
        alternativel: self alternative2: aNode asRExp

asRExp
    ^ self

```

Operacja `asRExp` przekształca literały na obiekty `RegularExpressions`. Oto definicje omawianych operacji w klasie `String`:

```

& aNode
    ^ SequenceExpression new
        expression1: self asRExp expression2: aNode asRExp

repeat
    ^ RepetitionExpression new repetition: self

| aNode
    ^ AlternationExpression new
        alternativel: self asRExp alternative2: aNode asRExp

asRExp
    ^ LiteralExpression new components: self

```

Gdybyśmy zdefiniowali te operacje na wyższym poziomie hierarchii klas (w klasie SequenceableCollection w języku Smalltalk-80 lub w klasie IndexedCollection w języku Smalltalk/V), byłoby zdefiniowane także dla takich klas, jak Array i OrderedCollection. Pozwoliłoby to dopasowywać wyrażenia regularne do sekwencji obiektów dowolnego rodzaju.

Drugi przykład to system do manipulowania wyrażeniami logicznymi i obliczania ich wartości zaimplementowany w języku C++. Symbole końcowe w tym języku to zmienne logiczne, czyli stałe true i false. Symbole pośrednie służą do reprezentowania wyrażeń obejmujących operatory and, or i not. Definicja gramatyki wygląda w następujący sposób¹:

```
BooleanExp ::= VariableExp | Constant | OrExp | AndExp | NotExp |
              '(' BooleanExp ')'
AndExp ::= BooleanExp 'and' BooleanExp
OrExp ::= BooleanExp 'or' BooleanExp
NotExp ::= 'not' BooleanExp
Constant ::= 'true' | 'false'
VariableExp ::= 'A' | 'B' | ... | 'X' | 'Y' | 'Z'
```

Zdefiniowaliśmy dwie operacje na wyrażeniach logicznych. Pierwsza z nich, Evaluate, oblicza wartość wyrażenia logicznego w kontekście, w którym każdej zmiennej przypisywana jest wartość true lub false. Druga operacja, Replace, tworzy nowe wyrażenie logiczne przez zastąpienie zmiennej wyrażeniem. Działanie tej operacji pokazuje, że wzorzec Interpreter można wykorzystać nie tylko do obliczania wartości wyrażeń. W tym przykładzie zastosowano go do manipulowania samym wyrażeniem.

W tym miejscu przedstawiamy szczegóły tylko kilku klas — BooleanExp, VariableExp i AndExp. Klassy OrExp i NotExp przypominają klasę AndExp. Klasa Constant reprezentuje stałe logiczne.

Klasa BooleanExp definiuje interfejs wszystkich klas określających wyrażenia logiczne:

```
class BooleanExp {
public:
    BooleanExp();
    virtual ~BooleanExp();

    virtual bool Evaluate(Context&) = 0;
    virtual BooleanExp* Replace(const char*, BooleanExp&) = 0;
    virtual BooleanExp* Copy() const = 0;
};
```

Klasa Context definiuje odwzorowanie ze zmiennych na wartości logiczne reprezentowane za pomocą stałych true i false z języka C++. Interfejs klasy Context wygląda tak:

```
class Context {
public:
    bool Lookup(const char*) const;
    void Assign(VariableExp*, bool);
};
```

¹ Dla uproszczenia ignorujemy tu priorytety operatorów i zakładamy, że ustala je obiekt tworzący drzewo składni.

Zmienna VariableExp reprezentuje nazwaną zmienną:

```
class VariableExp : public BooleanExp {
public:
    VariableExp(const char*);
    virtual ~VariableExp();

    virtual bool Evaluate(Context&);
    virtual BooleanExp* Replace(const char*, BooleanExp&);
    virtual BooleanExp* Copy() const;
private:
    char* _name;
};
```

Konstruktor tej klasy przyjmuje jako argument nazwę zmiennej:

```
VariableExp::VariableExp (const char* name) {
    _name = strdup(name);
}
```

Obliczenie wartości zmiennej polega na zwróceniu jej wartości w bieżącym kontekście.

```
bool VariableExp::Evaluate (Contexts aContext) {
    return aContext.Lookup(_name);
}
```

Skopiowanie zmiennej prowadzi do zwrócenia nowego obiektu VariableExp:

```
BooleanExp* VariableExp::Copy () const {
    return new VariableExp(_name);
}
```

Aby zastąpić zmienną wyrażeniem, należy sprawdzić, czy ma ona tę samą nazwę, co zmienna przekazana jako argument:

```
BooleanExp* VariableExp::Replace (
    const char* name, BooleanExp& exp
) {
    if (strcmp(name, _name) == 0) {
        return exp.Copy();
    } else {
        return new VariableExp(_name);
    }
}
```

Klasa AndExp reprezentuje wyrażenie utworzone przez połączenie dwóch wyrażeń logicznych operatorem AND:

```
class AndExp : public BooleanExp {
public:
    AndExp(BooleanExp*, BooleanExp*);
    virtual ~AndExp();

    virtual bool Evaluate(Context&);
    virtual BooleanExp* Replace(const char*, BooleanExp&);
    virtual BooleanExp* Copy() const;
```

```

private:
    BooleanExp* _operand1;
    BooleanExp* _operand2;
};

AndExp::AndExp (BooleanExp* op1, BooleanExp* op2) {
    _operand1 = op1;
    _operand2 = op2;
}

```

Operacja Evaluate klasy AndExp oblicza wartości jej operandów i zwraca ich iloczyn logiczny.

```

bool AndExp::Evaluate (Context& aContext) {
    return
        _operand1->Evaluate(aContext) &&
        _operand2->Evaluate(aContext);
}

```

W klasie AndExp operacje Copy i Replace są zaimplementowane za pomocą wywołań rekurencyjnych kierowanych do jej operandów:

```

BooleanExp* AndExp::Copy () const {
    return
        new AndExp(_operand1->Copy(), _operand2->Copy());
}

BooleanExp* AndExp::Replace (const char* name, BooleanExp& exp) {
    return
        new AndExp(
            _operand1->Replace(name, exp),
            _operand2->Replace(name, exp)
        );
}

```

Teraz możemy zdefiniować wyrażenie logiczne

$(\text{true} \text{ and } x) \text{ or } (y \text{ and } (\text{not } x))$

i obliczyć jego wartość po przypisaniu true lub false do zmiennych x i y:

```

BooleanExp* expression;
Context context;

VariableExp* x = new VariableExp("X");
VariableExp* y = new VariableExp("Y");

expression = new OrExp(
    new AndExp(new Constant(true), x),
    new AndExp(y, new NotExp(x))
);

context.Assign(x, false);
context.Assign(y, true);

bool result = expression->Evaluate(context);

```

Dla określonych tu wartości zmiennych x i y wyrażenie ma wartość `true`. Aby obliczyć wartość wyrażenia dla innych wartości zmiennych x i y , wystarczy zmienić kontekst.

Na zakończenie możemy zastąpić zmienną y nowym wyrażeniem i ponownie obliczyć wartość wcześniejszego wyrażenia:

```
VariableExp* z = new VariableExp("Z");
NotExp not_z(z);

BooleanExp* replacement = expression->Replace("Y", not_z);

context.Assign(z, true);

result = replacement->Evaluate(context);
```

Ten przykład ilustruje ważną cechę wzorca Interpreter — do „interpretowania” zdania można zastosować różnorodne operacje. Spośród trzech operacji zdefiniowanych w klasie BooleanExp najbardziej zgodna z naszym wyobrażeniem o działaniu interpretera jest operacja Evaluate — interpretuje ona program lub wyrażenie i zwraca prosty wynik.

Jednak jako interpreter można potraktować także operację Replace. Jest to interpreter, którego kontekst to nazwa zastępowanej zmiennej oraz wyrażenie zastępcze. Wynik działania tej operacji to nowe wyrażenie. Nawet operację Copy można uznać za interpreter działający przy pustym kontekście. Traktowanie operacji Replace i Copy jak interpreterów może wydawać się dziwne, ponieważ są to jedynie proste operacje na drzewach. Przykłady w opisie wzorca Odwiedzający (s. 280) ilustrują, jak można przeprowadzić refaktoryzację wszystkich trzech omówionych operacji i umieścić je w odrębnym odwiedzającym-interpreterze. Pokazuje to, że operacje te są bardzo podobne do siebie.

Wzorzec Interpreter dotyczy czegoś więcej niż operacji rozproszonej po hierarchii klas zgodnej ze wzorcem Kompozyt (s. 170). Uważamy operację Evaluate za interpreter, ponieważ hierarchia klas BooleanExp reprezentuje język. Gdyby podobna hierarchia klas reprezentowała części samochodowe, prawdopodobnie nie uznalibyśmy operacji w rodzaju Weight i Copy za interpretery, choć są rozproszone po hierarchii klas zgodnej ze wzorcem Kompozyt. Jednak zbioru części samochodowych nie uważamy za język. Istotny jest tu punkt widzenia. Gdybyśmy zaczęli publikować gramatykę części samochodowych, moglibyśmy uznać dotyczące ich operacje za mechanizmy interpretowania takiego języka.

ZNANE ZASTOSOWANIA

Wzorzec Interpreter jest powszechnie stosowany w kompilatorach zaimplementowanych za pomocą języków obiektowych (na przykład w kompilatorach języka Smalltalk). W języku SPECTalk wzorzec ten wykorzystano do interpretowania opisów formatów plików wejściowych [Sza92]. W pakiecie narzędziowym QOCA (jest on przeznaczony do rozwiązywania problemów z ograniczeniami) wzorzec ten służy do analizowania ograniczeń [HHMV92].

Wzorzec Interpreter w najbardziej ogólnej postaci (czyli w formie operacji rozproszonej po hierarchii klas zgodnej ze wzorcem Kompozyt) jest obecny w prawie każdym zastosowaniu wzorca Kompozyt. Jednak należy ograniczyć korzystanie z niego do tych przypadków, w których chcesz traktować hierarchię klas jak strukturę definiującą język.

POWIĄZANE WZORCE

Kompozyt (s. 170): drzewo składni abstrakcyjnej to przykład zastosowania wzorca Kompozyt.

Wzorzec Pyłek (s. 201) pokazuje, jak współużytkować symbole końcowe w drzewie składni abstrakcyjnej.

Iterator (s. 230): interpreter może korzystać z iteratora do przechodzenia po danej strukturze.

Wzorca Odwiedzający (s. 280) można użyć do umieszczenia w jednej klasie zachowania każdego węzła z drzewa składni abstrakcyjnej.

ITERATOR (ITERATOR)

obiektowy, operacyjny

PRZEZNACZENIE

Zapewnia sekwencyjny dostęp do elementów obiektu złożonego bez ujawniania jego wewnętrznej reprezentacji.

INNE NAZWY

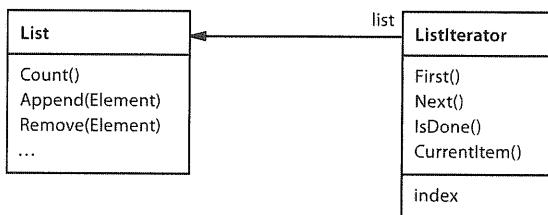
Kursor (ang. *cursor*).

UZASADNIENIE

Obiekty zagregowane, na przykład listy, powinny umożliwiać dostęp do swoich elementów bez ujawniania wewnętrznej struktury. Ponadto czasem możesz chcieć poruszać się po liście na różne sposoby w zależności od pożądanych efektów. Jednak prawdopodobnie nie chcesz zanadto powiększać interfejsu klasy **List** o operacje obsługujące różne metody przechodzenia po liście, nawet jeśli przewidujesz, że możesz ich potrzebować. Czasem potrzebna jest też możliwość jednoczesnego uruchomienia kilku procesów przechodzenia po liście.

Wzorzec Iterator umożliwia realizację wszystkich tych celów. Wzorzec ten oparto na pomyśle przeniesienia odpowiedzialności za udostępnianie elementów i przechodzenie po nich z obiektu listy do obiektu **iteratora**. Klasa **Iterator** definiuje interfejs udostępniający elementy listy. Obiekt iteratora odpowiada za śledzenie bieżącego elementu. Oznacza to, że wie, przez które elementy program już przeszedł.

Klasa **List** może na przykład wymagać klasy **ListIterator**. Oto relacje między tymi klasami:



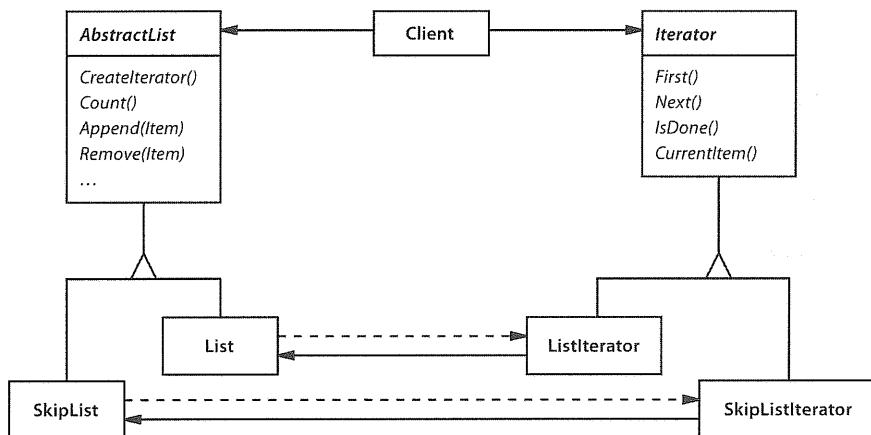
Przed utworzeniem obiektu **ListIterator** trzeba udostępnić obiekt **List**, po którego elementach iterator ma przechodzić. Kiedy egzemplarz klasy **ListIterator** jest już gotowy, można uzyskać sekwencyjny dostęp do elementów listy. Operacja `CurrentItem` zwraca bieżący element listy, operacja `First` inicjuje bieżący element za pomocą pierwszego elementu, operacja `Next` zmienia bieżący element na następny, a operacja `IsDone` sprawdza, czy program nie wyszedł poza ostatni element (czyli czy przechodzenie się nie zakończyło).

Wyodrębnienie mechanizmu przechodzenia po elementach z obiektu *List* umożliwia zdefiniowanie iteratorów dla różnych reguł przechodzenia bez konieczności wyliczania ich w interfejsie klasy *List*. Na przykład klasa *FilteringListIterator* może udostępniać tylko elementy spełniające określone warunki związane z filtrowaniem.

Zauważmy, że iterator i lista są powiązane, a klient musi wiedzieć, iż przechodzi po elementach *listy*, a nie innej zagregowanej struktury. Dlatego klient może korzystać tylko z wybranej struktury tego rodzaju. Korzystniejsza byłaby możliwość zmiany klasy agregatu bez modyfikowania kodu klienta. Można to zrobić przez uogólnienie iteratora i zapewnienie obsługi **iteracji polimorficznej**.

Załóżmy na przykład, że korzystamy z implementacji listy w postaci klasy *SkipList* (czyli lista z przeskokami). Lista z przeskokami [Pug90] to probabilistyczna struktura danych o właściwościach podobnych do drzew zrównoważonych. Chcemy mieć możliwość napisania kodu, który będzie działał zarówno dla obiektów *List*, jak i obiektów *SkipList*.

Zdefiniujmy klasę *AbstractList* udostępniającą wspólny interfejs do manipulowania listami. Potrzebujemy też klasy *Iterator* definiującej wspólny interfejs do przechodzenia po listach. Następnie możemy zdefiniować podklasy konkretne klasy *Iterator* dla różnych implementacji listy. W efekcie mechanizm poruszania się po listach będzie niezależny od klas konkretnych agregatów.



Do rozwiązania pozostał problem sposobu tworzenia iteratora. Ponieważ chcemy móc pisać kod niezależny od podklas konkretnych klasy *List*, nie możemy po prostu utworzyć egzemplarza określonej klasy. Zamiast tego sprawimy, aby to obiekty list tworzyły odpowiadające im iteratory. Wymaga to przygotowania operacji w rodzaju *CreateIterator*, której klienci będą mogli użyć do zażądania obiektu iteratora.

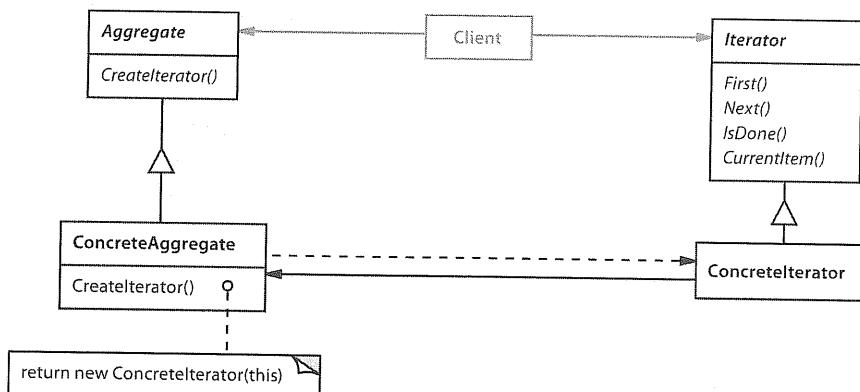
CreateIterator to przykład metody wytwarzającej (zobacz wzorzec Metoda wytwarzająca, s. 110). Stosujemy ją tu, aby umożliwić klientowi zażądanie odpowiedniego iteratora od obiektu listy. Podejście oparte na wzorcu Metoda wytwarzająca powoduje utworzenie dwóch hierarchii klas — jednej dla list i drugiej dla iteratorów. Metoda wytwarzająca *CreateIterator* łączy te dwie hierarchie.

WARUNKI STOSOWANIA

Ze wzorca Iterator należy korzystać w następujących warunkach:

- ▶ Kiedy chcesz uzyskać dostęp do zawartości obiektu zagregowanego bez ujawniania jego wewnętrznej reprezentacji.
- ▶ Jeśli chcesz umożliwić jednoczesne działanie wielu procesów przehodzenie po obiektach zagregowanych.
- ▶ Jeżeli chcesz udostępnić jednolity interfejs do poruszania się po różnych zagregowanych strukturach (czyli zapewnić obsługę iteracji polimorficznej).

STRUKTURA



ELEMENTY

- ▶ **Iterator:**
 - definiuje interfejs umożliwiający dostęp do elementów i przechodzenie po nich.
- ▶ **ConcreteIterator:**
 - obejmuje implementację interfejsu klasy **Iterator**;
 - śledzi bieżącą pozycję w czasie przechodzenia po agregacie.
- ▶ **Aggregate:**
 - definiuje interfejs do tworzenia obiektów **Iterator**.
- ▶ **ConcreteAggregate:**
 - obejmuje implementację interfejsu do tworzenia obiektów **Iterator** zwracającą egzemplarz odpowiedniej klasy **ConcreteIterator**.

WSPÓŁDZIAŁANIE

- ▶ Obiekt **ConcreteIterator** śledzi bieżący obiekt w agregacie i potrafi ustalić następny obiekt, do którego należy przejść.

KONSEKWENCJE

Stosowanie wzorca Iterator ma trzy ważne konsekwencje:

1. *Umożliwienie zmiany sposobu przechodzenia po agregacie.* Po skomplikowanych agregatach można poruszać się na wiele sposobów. Na przykład generowanie kodu i analiza semantyczna wymagają przechodzenia po drzewach składni. Przy generowaniu kodu można poruszać się po nich za pomocą metody inorder lub preorder. Iteratory sprawiają, że można łatwo zmienić algorytm przechodzenia. Wystarczy w tym celu zastąpić jeden egzemplarz iteratora innym. Można też zdefiniować podklasy klasy Iterator obsługujące nowe sposoby poruszania się po agregacie.
2. *Uproszczenie interfejsu klasy Aggregate.* Umieszczenie interfejsu umożliwiającego przechodzenie w klasie Iterator powoduje, że nie trzeba go tworzyć w klasie Aggregate, co upraszcza interfejs tej ostatniej.
3. *Możliwość jednoczesnego działania więcej niż jednego procesu przechodzenia po agregacie.* Iterator śledzi powiązany z nim stan przechodzenia, dlatego jednocześnie może działać wiele procesów przechodzenia po agregacie.

IMPLEMENTACJA

Istnieje wiele wersji i odmian implementacji iteratorów. Poniżej opisujemy niektóre z ważnych rozwiązań. Ich wady i zalety często zależą od struktur sterujących dostępnych w danym języku. Niektóre języki (na przykład CLU [LG86]) bezpośrednio obsługują omawiany wzorzec.

1. *Który element ma sterować iterowaniem?* Podstawową kwestią jest określenie, który element ma sterować iterowaniem — iterator czy korzystający z niego klient. Jeśli jest to klient, iterator jest nazywany **iteratorem zewnętrznym**. Jeżeli iterator steruje omawianym procesem, nazywamy go **iteratorem wewnętrznym**². Klienci korzystające z iteratora zewnętrznego muszą przejść do następnego elementu i jawnie zażądać go od iteratora. Natomiast w przypadku iteratorów wewnętrznych klient przekazuje im operację do wykonania, a iterator uruchamia ją dla każdego elementu z aggregatu.

Iteratory zewnętrzne są elastyczniejsze od wewnętrznych. Przy użyciu iteratorów zewnętrznych łatwo jest na przykład porównać dwie kolekcje, co jest praktycznie niemożliwe przy korzystaniu z iteratorów wewnętrznych. Te ostatnie są szczególnie ograniczone w językach w rodzaju C++, które — w odróżnieniu od języków Smalltalk i CLOS — nie udostępniają funkcji anonimowych, domknięć i kontynuacji. Jednak z drugiej strony iteratory wewnętrzne są łatwiejsze w użytkowaniu, ponieważ definiują logikę iterowania za programistę.

2. *Gdzie zdefiniowany jest algorytm przechodzenia?* Iterator to niejedyne miejsce, w którym można zdefiniować algorytm przechodzenia. Można go umieścić także w agregacie i stosować iterator do przechowywania samego stanu iteracji. Iteratory tego rodzaju nazywamy **kursorami**,

² Booch nazywa iteratory zewnętrzne i wewnętrzne odpowiednio **aktywnymi** oraz **pasywnymi** [Boo94]. Pojęcia „aktywny” i „pasywny” dotyczą tu roli klienta, a nie poziomu aktywności iteratora.

ponieważ jedynie wskazują bieżącą pozycję w agregacie. Klient może wywołać operację `Next` aggregatu z kursorem podanym jako argument, a operacja ta zmieni stan kursora³.

Jeśli to iterator odpowiada za algorytm przechodzenia, można łatwo zastosować różne algorytmy iteracji do tego samego aggregatu. Ponadto łatwiej jest powtórnie wykorzystać ten sam algorytm dla różnych aggregatów. Jednak w algorytmie przechodzenia potrzebny może być dostęp do zmiennych prywatnych aggregatu. Wtedy umieszczenie algorytmu przechodzenia w iteratorze naruszy kapsułkowanie aggregatu.

3. *Jak niezawodny jest iterator?* Modyfikowanie aggregatu w czasie przechodzenia po nim może być ryzykowne. Jeśli elementy są dodawane do aggregatu lub usuwane z niego, program może dwukrotnie wykorzystać dany element lub w ogóle go pominąć. Proste rozwiążanie polega na skopiowaniu aggregatu i przejściu po kopii, jednak zwykle jest to zbyt kosztowne.

Niezawodny iterator gwarantuje, że wstawianie i usuwanie nie zakłóci przechodzenia, przy czym kopianie aggregatu nie jest wtedy konieczne. Istnieje wiele sposobów na zaimplementowanie niezawodnych iteratorów. Większość programistów polega na rejestrowaniu iteratora w agregacie. Przy wstawieniu lub usunięciu elementu aggregat albo dostosowuje wewnętrzny stan utworzonych iteratorów, albo przechowuje potrzebne informacje wewnętrznie, aby zapewnić prawidłowy przebieg przechodzenia.

Kofler przedstawia ciekawą analizę implementacji niezawodnych iteratorów w platformie ET++ [Kof93]. Murray omawia implementację takich iteratorów na potrzeby klasy `List` (jest to jeden ze standardowych komponentów języka USL) [Mur93].

4. *Dodatkowe operacje klasy Iterator.* Minimalny interfejs klasy `Iterator` składa się z operacji `First`, `Next`, `IsDone` i `CurrentItem`⁴. Czasem przydatne są też inne operacje. Agregaty uporządkowane mogą na przykład udostępniać operację `Previous` powodującą przeniesienie iteratora do poprzedniego elementu. Operacja `SkipTo` jest użyteczna w posortowanych lub indeksowanych kolekcjach. Przenosi ona iterator do obiektu spełniającego określone kryteria.

5. *Stosowanie iteratorów polimorficznych w języku C++.* Iteratory polimorficzne powodują pewne koszty. Przy korzystaniu z nich obiekt iteratora trzeba zaalokować dynamicznie za pomocą metody wytwarzającej. Dlatego technikę tę należy stosować tylko wtedy, kiedy potrzebny jest polimorfizm. W innych sytuacjach należy korzystać z iteratorów konkretnych, które można zaalokować na stosie.

Iteratory polimorficzne mają jeszcze jedną wadę — klient odpowiada za ich usunięcie. Rozwiążanie to jest podatne na błędy, ponieważ łatwo jest zapomnieć o zwolnieniu iteratora zaalokowanego na stercie po zakończeniu korzystania z niego. Dzieje się tak szczególnie często, jeśli istnieje wiele punktów wyjścia z operacji. Ponadto po wystąpieniu wyjątku obiekt iteratora nigdy nie zostanie zwolniony.

³ Kursory są prostym przykładem zastosowania wzorca Pamiątki (s. 283), a wiele kwestii implementacyjnych dotyczy w równym stopniu kursorów, jak i pamiętek.

⁴ Można jeszcze bardziej zmniejszyć ten interfejs przez połączenie operacji `Next`, `IsDone` i `CurrentItem` w jedną, która przechodzi do następnego obiektu i zwraca go. Jeśli przechodzenie się zakończy, operacja ta zwróci specjalną wartość (na przykład 0) oznaczającą koniec iterowania.

Rozwiązaniem tego problemu jest wzorzec Pełnomocnik (s. 191). Można wykorzystać z alokowany na stosie pełnomocnik jako zastępnik rzeczywistego iteratora. W destruktorze pełnomocnika należy usunąć iterator. Dlatego kiedy pełnomocnik wyjdzie z zasięgu programu, rzeczywisty iterator zostanie usunięty wraz z nim. Pełnomocnik zapewnia prawidłowe wykonanie operacji porządkowych, i to nawet po wystąpieniu wyjątku. Jest to zastosowanie dobrze znanej techniki z języka C++ — przydziału zasobów w czasie inicjowania [ES90]. Ilustrujemy ją w punkcie „Przykładowy kod”.

6. *Iteratory mają uprzywilejowany dostęp.* Iterator można traktować jak rozszerzenie tworzącego go agregatu. Iterator i agregat są ściśle powiązane. W języku C++ można wyrazić tę bliską relację przez utworzenie iteratora jako klasy zaprzyjaźnionej (z modyfikatorem `friend`) agregatu. Nie trzeba wtedy definiować w agregacie operacji służących wyłącznie do tego, aby umożliwić wydajne zaimplementowanie przechodzenia w iteratorach.

Jednak taki uprzywilejowany dostęp może utrudnić definiowanie nowych metod przechodzenia, ponieważ trzeba będzie zmienić interfejs agregatu przez dodanie nowej klasy zaprzyjaźnionej. Aby uniknąć tego problemu, w klasie `Iterator` można umieścić operacje chronione (modyfikator `protected`) umożliwiające dostęp do ważnych, ale publicznie niedostępnych składowych agregatu. Podklasy klasy `Iterator` (i tylko one) będą mogły korzystać z tych chronionych operacji do uzyskania uprzywilejowanego dostępu do agregatu.

7. *Iteratory dla kompozytów.* Czasem trudno jest zaimplementować iteratory zewnętrzne dla rekurencyjnych struktur zagregowanych (występują one na przykład we wzorcu Kompozyt, s. 170). Dzieje się tak, ponieważ pozycja w strukturze może być określona na wielu poziomach zagnieżdżonych agregatów. Dlatego w iteratorze zewnętrznym trzeba zapisać ścieżkę przechodzenia po kompozycie, aby móc śledzić bieżący obiekt. Czasem łatwiej jest zastosować iterator wewnętrzny. Pozwala on rejestrować bieżącą pozycję przez rekurencyjne wywoływanie samego siebie, co prowadzi do niejawnego zapisywania ścieżki na stosie wywołań.

Jeśli węzły w kompozycie udostępniają interfejs do przechodzenia z węzła do elementów równorzędnych, nadrzędnych i podrzędnych, lepszym rozwiązaniem może się okazać iterator oparty na kurSORZE. KURSOR musi jedynie śledzić bieżący węzeł i może wykorzystać interfejs węzłów do poruszania się po kompozycie.

Często potrzebna jest możliwość poruszania się po kompozycie na różne sposoby. Powszechnie stosowane są metody preorder, postorder, inorder i wszerz. Do obsługi każdego sposobu przechodzenia można zastosować inną klasę iteratorów.

8. *Iteratory puste.* **NullIterator** to klasa uproszczonego iteratora pomocna przy obsłudze warunków brzegowych. Klasa ta z definicji zawsze informuje, że zakończyła przechodzenie. Oznacza to, że jej operacja `IsDone` zwraca wyłącznie wartość `true`.

Klasa `NullIterator` pozwala czasem uprościć przechodzenie po agregatach o strukturze drzewiastej (takich jak kompozyty). W każdym momencie poruszania się po agregacie możemy zażądać od bieżącego elementu iteratora dla jego elementów podrzędnych. Elementy agregatu zwracają standardowy iterator konkretny. Jednak liście zwracają egzemplarz klasy `NullIterator`. Pozwala to w jednolity sposób zaimplementować przechodzenie po całej strukturze.

PRZYKŁADOWY KOD

Przyjrzymy się implementacji prostej klasy `List`. Jest ona częścią opracowanej przez nas biblioteki podstawowej (zobacz dodatek C). Pokażemy dwie implementacje klasy `Iterator`. Jedna służy do przechodzenia po liście od początku do końca, a druga — do poruszania się od końca do początku (biblioteka podstawowa obejmuje tylko pierwszą z tych implementacji). Następnie pokażemy, jak wykorzystać te iteratory i jak uniknąć powiązania programu z określoną implementacją. Następnie zmodyfikujemy projekt, aby zagwarantować prawidłowe usuwanie iteratorów. Ostatni przykład ilustruje działanie iteratora wewnętrznego i pozwala porównać go z jego zewnętrznym odpowiednikiem.

1. *Interfejsy klas List i Iterator.* Najpierw przyjrzymy się fragmentowi interfejsu klasy `List` związanemu z implementacją iteratorów. Pełny interfejs przedstawia dodatek C.

```
template <class Item>
class List {
public:
    List(long size = DEFAULT_LIST_CAPACITY);

    long Count() const;
    Item& Get(long index) const;
    // ...
};
```

Klasa `List` udostępnia stosunkowo wydajny sposób obsługi iteracji za pośrednictwem jej interfejsu publicznego. Wystarczy on do zaimplementowania obu metod przechodzenia. Dlatego nie trzeba przypisywać iteratorom uprzywilejowanego dostępu do podstawowej struktury danych (oznacza to, że iteratory nie muszą być klasami zaprzyjaźnionymi z klasą `List`). Aby umożliwić niezauważalną zmianę metody przechodzenia, zdefiniujemy klasę abstrakcyjną `Iterator`. Obejmuje ona definicję interfejsu iteratora.

```
template <class Item>
class Iterator {
public:
    virtual void First() = 0;
    virtual void Next() = 0;
    virtual bool IsDone() const = 0;
    virtual Item CurrentItem() const = 0;
protected:
    Iterator();
};
```

2. *Implementacje podklas klasy Iterator.* `ListIterator` to podklasa klasy `Iterator`.

```
template <class Item>
class ListIterator : public Iterator<Item> {
public:
    ListIterator(const List<Item>* aList);
    virtual void First();
    virtual void Next();
    virtual bool IsDone() const;
    virtual Item CurrentItem() const;
```

```
private:
    const List<Item>* _list;
    long _current;
};
```

Implementacja klasy `ListIterator` jest prosta. Klasa ta przechowuje obiekt `List` wraz z indeksem `_current` dla tej listy:

```
template <class Item>
ListIterator<Item>::ListIterator (
    const List<Item>* aList
) : _list(aList), _current(0) { }
```

Operacja `First` ustawia iterator na pierwszy element:

```
template <class Item>
void ListIterator<Item>::First () {
    _current = 0;
}
```

Operacja `Next` zmienia bieżący element na następny:

```
template <class Item>
void ListIterator<Item>::Next () {
    _current++;
}
```

Operacja `IsDone` sprawdza, czy indeks wskazuje element z listy:

```
template <class Item>
bool ListIterator<Item>::IsDone () const {
    return _current >= _list->Count();
}
```

Ostatnia operacja, `CurrentItem`, zwraca element znajdujący się pod bieżącym indeksem. Jeśli iterowanie zostało już zakończone, operacja zgłasza wyjątek `IteratorOutOfBoundsException`:

```
template <class Item>
Item ListIterator<Item>::CurrentItem () const {
    if (IsDone()) {
        throw IteratorOutOfBoundsException();
    }
    return _list->Get(_current);
}
```

Implementacja klasy `ReverseListIterator` wygląda prawie identycznie. Różnice polegają na tym, że operacja `First` przypisuje do zmiennej `_current` ostatni element listy, a operacja `Next` zmniejsza wartość zmiennej `_current` i przechodzi w kierunku pierwszego elementu.

3. *Korzystanie z iteratorów.* Założymy, że mamy obiekt `List` przechowujący obiekty `Employee` (zawierają one informacje o pracownikach) i chcemy wyświetlić dane wszystkich pracowników z listy. Klasa `Employee` udostępnia przeznaczoną do tego operację `Print`. Na potrzeby wyświetlania listy zdefiniujemy operację `PrintEmployees` przyjmującą iterator jako argument. W operacji tej iterator posłuży do przejścia po liście i wyświetlenia jej elementów.

```

void PrintEmployees (Iterator<Employee*>& i) {
    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Print();
    }
}

```

Ponieważ dostępne są iteratory do przechodzenia od końca do początku listy i w drugą stronę, możemy ponownie wykorzystać tę operację do wyświetlania danych pracowników w obu porządkach.

```

List<Employee*>* employees;
// ...
ListIterator<Employee*> forward(employees);
ReverseListIterator<Employee*> backward(employees);

PrintEmployees(forward);
PrintEmployees(backward);

```

4. *Unikanie powiązania z określona implementacją listy.* Zastanówmy się, jak zastosowanie wersji klasy List z przeskokami wpłynie na kod do obsługi iterowania. Podklaśa SkipList klasy List musi udostępniać klasę SkipListIterator z implementacją interfejsu klasy Iterator. Wewnętrznie klasa SkipListIterator musi przechowywać także inne dane obok indeksu, aby można wydajnie przeprowadzić iterację. Jednak z uwagi na to, że klasa SkipListIterator jest zgodna z interfejsem klasy Iterator, operację PrintEmployees można wywołać także wtedy, kiedy dane pracowników są zapisane w obiekcie SkipList.

```

SkipList<Employee*>* employees;
// ...

SkipListIterator<Employee*> iterator(employees);
PrintEmployees(iterator);

```

Choć to rozwiązanie działa, powoduje powiązanie programu z określona implementacją klasy List (a dokładniej — z podklassą SkipList), czego warto unikać. Możemy dodać klasę AbstractList, aby utworzyć standardowy interfejs dla różnych implementacji list. Klasy List i SkipList będą wtedy podklasami klasy AbstractList.

Aby umożliwić iterowanie polimorficzne, w klasie AbstractList zdefiniujemy metodę wytwarzającą CreateIterator, którą przesłonimy w podklassach, tak żeby zwracała odpowiedni iterator:

```

template <class Item>
class AbstractList {
public:
    virtual Iterator<Item>* CreateIterator() const = 0;
    // ...
};

```

Inna możliwość to zdefiniowanie ogólnej klasy mieszanej Traversable określającej interfejs do tworzenia iteratorów. Klasę agregatów można połączyć z klasą Traversable, aby dodać obsługę iterowania polimorficznego.

W klasie List operacja CreateIterator jest przesłonięta, tak aby zwracała obiekt ListIterator:

```
template <class Item>
Iterator<Item>* List<Item>::CreateIterator () const {
    return new ListIterator<Item>(this);
}
```

Teraz możemy napisać kod do wyświetlania danych pracowników niezależnie od określonej reprezentacji.

```
// Znamy jedynie klasę AbstractList.
AbstractList<Employee*>* employees;
// ...

Iterator<Employee*>* iterator = employees->CreateIterator();
PrintEmployees(*iterator);
delete iterator;
```

5. *Gwarantowanie, że iteratory zostaną usunięte.* Warto zauważyc, że operacja CreateIterator zwraca nowo utworzony obiekt iteratora. Programista odpowiada za jego usunięcie. Jeśli o tym zapomni, nastąpi wyciekanie pamięci. Aby ułatwić pracę autorom klientów, udostępnimy klasę IteratorPtr pełniącą funkcję pełnomocnika dla iteratorów. Zajmie się ona usunięciem obiektu Iterator, kiedy ten znajdzie się poza zasięgiem programu.

Obiekty IteratorPtr zawsze są alokowane na stosie⁵. Język C++ automatycznie wywoła destruktor tej klasy, co spowoduje usunięcie rzeczywistego iteratora. W klasie IteratorPtr operatory operator-> i operator* są przeciążane w taki sposób, aby obiekt IteratorPtr można traktować jak wskaźnik do iteratora. Wszystkie składowe klasy IteratorPtr są zaimplementowane wewnętrznie, dlatego nie powodują dodatkowych kosztów.

```
template <class Item>
class IteratorPtr {
public:
    IteratorPtr(Iterator<Item>* i): _i(i) { }
    ~IteratorPtr() { delete _i; }

    Iterator<Item>* operator->() { return _i; }
    Iterator<Item>& operator*() { return *_i; }

private:
    // Należy uniemożliwić kopowanie i przypisywanie, aby
    // uniknąć wielokrotnego usuwania iteratora _i:

    IteratorPtr(const IteratorPtr&);
    IteratorPtr& operator=(const IteratorPtr&);

private:
    Iterator<Item>* _i;
};
```

⁵ Można to zagwarantować w czasie komplikacji przez zadeklarowanie prywatnych operatorów new i delete. Ich implementacja nie jest potrzebna.

Klasa `IteratorPtr` umożliwia uproszczenie kodu do wyświetlania listy:

```
AbstractList<Employee*>* employees;
// ...

IteratorPtr<Employee*> iterator(employees->CreateIterator());
PrintEmployees(*iterator);
```

6. *Iterator wewnętrzny ListIterator*. W ostatnim przykładzie przyjrzymy się możliwej implementacji wewnętrznej (pasywnej) klasy `ListIterator`. W tym rozwiążaniu to iterator kontroluje iterowanie i uruchamia operacje dla każdego elementu.

Problem polega tu na sparametryzowaniu iteratatora za pomocą operacji, którą chcemy uruchomić dla każdego elementu. Język C++ nie obsługuje funkcji anonimowych ani domknięć umożliwiających wykonanie tego zadania w innych językach. Istnieją jednak przynajmniej dwie możliwości — (1) przekazanie wskaźnika do funkcji (globalnej lub statycznej) lub (2) wykorzystanie podklas. W pierwszym z tych rozwiązań iterator wywołuje w każdym kroku iteracji przekazaną do niego operację. W drugiej technice iterator wywołuje operację przesłoniętą w podklasie w celu wykonania określonych zadań.

Żadne z tych rozwiązań nie jest doskonale. Często warto rejestrować stan w czasie iteracji, a funkcje nie są dobrze dostosowane do tego zadania, dlatego do przechowywania stanu trzeba zastosować zmienne statyczne. Podklasa klasy `Iterator` to wygodne miejsce na przechowywanie zarejestrowanego stanu, na przykład zmiennej egzemplarza. Jednak utworzenie podklasy dla każdej metody przechodzenia wymaga więcej pracy.

Oto zarys drugiego, opartego na podklasach rozwiązania. Nazwa iteratatora wewnętrznego to `ListTraverser`.

```
template <class Item>
class ListTraverser {
public:
    ListTraverser(List<Item>* aList);
    bool Traverse();
protected:
    virtual bool ProcessItem(const Item&) = 0;
private:
    ListIterator<Item> _iterator;
};
```

Klasa `ListTraverser` przyjmuje jako parametr egzemplarz klasy `List`. Wewnętrznie korzysta do przechodzenia po liście z zewnętrznego iteratatora `ListIterator`. Operacja `Traverse` rozpoczyna ten proces i wywołuje dla każdego elementu operację `ProcessItem`. Iteratator wewnętrzny może zakończyć przechodzenie przez zwrócenie wartości `false` w operacji `ProcessItem`. Operacja `Traverse` zwraca sterowanie, jeśli przechodzenie przedwcześnie się zakończy.

```
template <class Item>
ListTraverser<Item>::ListTraverser (
    List<Item>* aList
) : _iterator(aList) { }

template <class Item>
bool ListTraverser<Item>::Traverse () {
```

```

        bool result = false;

        for (
            _iterator.First();
            !_iterator.IsDone();
            _iterator.Next()
        ) {
            result = ProcessItem(_iterator.CurrentItem());

            if (result == false) {
                break;
            }
        }
    return result;
}

```

Użyjmy klasy `ListTraverser` do wyświetlenia danych pierwszych 10 pracowników z listy. W tym celu trzeba utworzyć podklasę klasy `ListTraverser` i przesłonić operację `ProcessItem`. Do zliczania wyświetlonych pracowników posłuży zmieniąca egzemplarza `_count`.

```

class PrintNEmployees : public ListTraverser<Employee*> {
public:
    PrintNEmployees(List<Employee*>* aList, int n) :
        ListTraverser<Employee*>(aList),
        _total(n), _count(0) {}

protected:
    bool ProcessItem(Employee* const&);

private:
    int _total;
    int _count;
};

bool PrintNEmployees::ProcessItem (Employee* const& e) {
    _count++;
    e->Print();
    return _count < _total;
}

```

Oto w jaki sposób operacje `PrintNEmployees` wyświetla dane pierwszych 10 pracowników z listy:

```

List<Employee*>* employees;
// ...

PrintNEmployees pa(employees, 10);
pa.Traverse();

```

Warto zauważyć, że klient nie musiał tworzyć pętli na potrzeby iteracji. Całą logikę procesu iteracji można wielokrotnie wykorzystać. Jest to główna zaleta stosowania iteratora wewnętrznego. Jednak jego utworzenie wymaga więcej pracy niż zbudowanie iteratora zewnętrznego, ponieważ trzeba zdefiniować nową klasę. Porównajmy poprzedni fragment z korzystaniem z iteratora wewnętrznego:

```

ListIterator<Employee*> i(employees);
int count = 0;

for (i.First(); !i.IsDone(); i.Next()) {
    count++;
    i.CurrentItem()->Print();

    if (count >= 10) {
        break;
    }
}

```

Iteratory wewnętrzne mogą kapsułkować różne metody iteracji. Na przykład klasa `FilteringListTraverser` kapsułkuje iterację, która powoduje przetworzenie tylko elementów o określonych cechach:

```

template <class Item>
class FilteringListTraverser {
public:
    FilteringListTraverser(List<Item*>* aList);
    bool Traverse();
protected:
    virtual bool ProcessItem(const Item&) = 0;
    virtual bool TestItem(const Item&) = 0;
private:
    ListIterator<Item> _iterator;
};

```

Ten interfejs jest prawie taki sam jak interfejs klasy `ListTraverser`. Różnicą jest dodatkowa funkcja `TestItem` z definicją testu. W podklasach przesłoniliśmy tę funkcję, aby sprawdzać test.

Operacja `Traverse` na podstawie wyniku testu określa, czy należy kontynuować przejście:

```

template <class Item>
void FilteringListTraverser<Item>::Traverse () {
    bool result = false;

    for (
        _iterator.First();
        !_iterator.IsDone();
        _iterator.Next()
    ) {
        if (TestItem(_iterator.CurrentItem())) {
            result = ProcessItem(_iterator.CurrentItem());
            if (result == false) {
                break;
            }
        }
    }
    return result;
}

```

W innej wersji tej klasy można zdefiniować operację Traverse tak, aby zwracała sterowanie, jeśli przynajmniej jeden element przechodzi test⁶.

ZNANE ZASTOSOWANIA

Iteratory są powszechnie stosowane w systemach obiektowych. Większość bibliotek klas do obsługi kolekcji udostępnia iteratory w tej lub innej postaci.

Oto przykład z komponentów Boocha [Boo94] — popularna biblioteka klas do obsługi kolekcji. Udostępnia ona implementację kolejek o stałym rozmiarze (ograniczonych) i dynamicznie rozszerzanych (nieograniczonych). Interfejs kolejki jest zdefiniowany w klasie abstrakcyjnej Queue. Aby zapewnić obsługę polimorficznego iterowania dla różnych implementacji kolejek, iterator kolejek zaimplementowano w kategoriach interfejsu klasy abstrakcyjnej Queue. To rozwiązanie ma tę zaletę, że nie trzeba tworzyć metody fabrycznej, aby żądać odpowiednich iteratorów od implementacji kolejek. Jednak interfejs klasy abstrakcyjnej Queue musi być wystarczająco rozbudowany, aby można wydajnie zaimplementować iterator.

W języku Smalltalk iteratorów nie trzeba definiować w tak bezpośredni sposób. Standardowe klasy do obsługi kolekcji (Bag, Set, Dictionary, OrderedCollection, String itd.) definiują wewnętrzną metodę iteratora do: przyjmującą jako argument blok (na przykład domknięcie). Każdy element w kolekcji jest wiązany ze zmienną lokalną z bloku, po czym następuje wykonanie kodu bloku. Język Smalltalk obejmuje też zestaw klas Stream udostępniających interfejs podobny do interfejsów iteratorów. Klasa ReadStream to w zasadzie odpowiednik klasy Iterator — może działać jak iterator zewnętrzny dla wszystkich kolekcji sekwencyjnych. Nie istnieją natomiast standardowe iteratory zewnętrzne dla kolekcji niesekwencyjnych, takich jak Set i Dictionary.

Opisane wcześniej iteratory polimorficzne i porządkujący pełnomocnik są udostępniane przez klasy kontenerowe platformy ET++ [WGM88]. W klasach platformy Unidraw (służy ona do tworzenia edytorów graficznych) wykorzystano iteratory oparte na kurSORACH [VL90].

Biblioteka ObjectWindows 2.0 [Bor94] udostępnia hierarchię klas reprezentujących iteratory dla kontenerów. Po kontenerach różnych typów można przechodzić w taki sam sposób. Składnia iteracji w bibliotece ObjectWindows jest oparta na przeciążeniu operatora post-incrementacji (++), tak aby przechodził do następnego kroku iteracji.

POWIĄZANE WZORCE

Kompozyt (s. 170): iteratory często stosuje się do struktur rekurencyjnych, takich jak kompozyty.

Metoda wytwórcza (s. 110): iteratory polimorficzne korzystają z metod wytwórczych do generowania egzemplarzy odpowiednich podklas klasy Iterator.

Wzorzec Pamiątki (s. 294) jest często stosowany w połączeniu ze wzorcem Iterator. Iterator może korzystać z pamiętki do zapisywania stanu iteracji i wewnętrznie przechowuje pamiętkę.

⁶ Operacja Traverse w tym kodzie to przykład zastosowania wzorca Metoda szablonowa (s. 325) z prostymi operacjami TestItem i ProcessItem.

ŁAŃCUCH ZOBOWIAZAŃ (CHAIN OF RESPONSIBILITY) *obiektowy, operacyjny*

PRZEZNACZENIE

Pozwala uniknąć wiązania nadawcy żądania z jego odbiorcą, ponieważ umożliwia obsłużenie żądania więcej niż jednemu obiekowi. Łączy w łańcuch obiekty odbiorcze i przekazuje między nimi żądanie do momentu obsłużenia go.

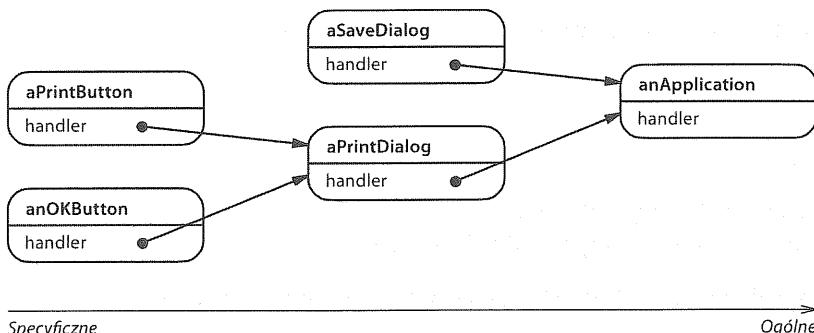
UZASADNIENIE

Rozważmy uwzględniający kontekst system pomocy w graficznym interfejsie użytkownika. Użytkownik możetrzymać informacje dotyczące dowolnej części interfejsu przez samo jej kliknięcie. Udostępniana pomoc zależy od wybranego elementu interfejsu i kontekstu. Na przykład widget w postaci przycisku w oknie dialogowym może być powiązany z innymi informacjami niż podobny przycisk w oknie głównym. Jeśli nie ma informacji na temat określonej części interfejsu, system pomocy powinien wyświetlić ogólniejszy komunikat na temat bezpośredniego kontekstu — na przykład całego okna dialogowego.

Dlatego naturalne jest uporządkowanie informacji na podstawie ich ogólności — od najbardziej do najmniej konkretnych. Ponadto oczywiste jest, że do obsługi żądania pomocy posłuży jeden z kilku obiektów interfejsu użytkownika. Ich wybór zależy od kontekstu i poziomu ogólności dostępnej pomocy.

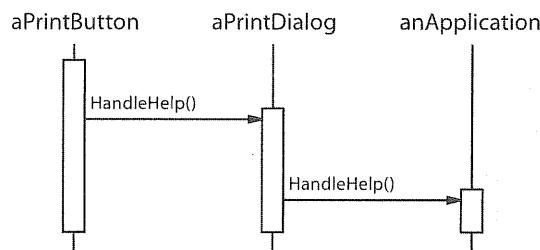
Problem polega na tym, że obiekt ostatecznie udostępniający pomoc nie jest bezpośrednio znany obiekowi (na przykład przyciskowi), który inicjuje żądanie. Potrzebny jest sposób na rozdzielenie przycisku zgłaszającego żądanie od obiektów, które mogą zwracać informacje. Wzorzec Łańcuch zobowiązań opisuje działanie takiego rozwiązania.

Pomysł, na którym oparty jest ten wzorzec, polega na rozdzieleniu nadawców od odbiorców przez umożliwienie obsługi żądania kilku obiektom. Żądanie jest przekazywane wzduż łańcucha obiektów do momentu obsłużenia go przez jeden z nich.



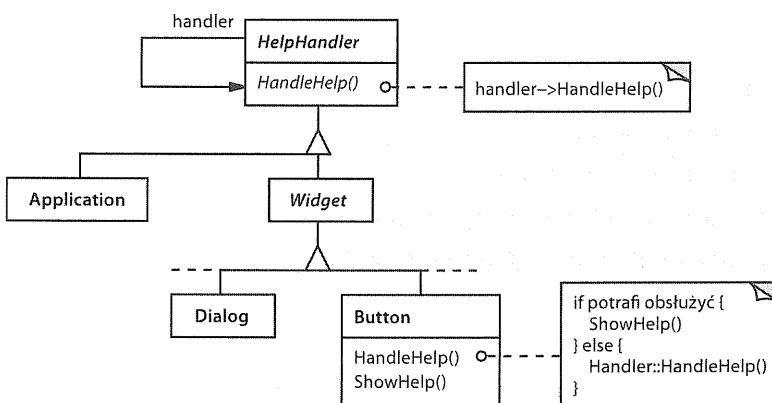
Pierwszy obiekt w łańcuchu odbiera żądanie i albo je obsługuje, albo przekazuje do następnego kandydata, a ten robi to samo. Obiekt zgłoszający żądanie nie wie, który obiekt je obsługuje. Można powiedzieć, że żądanie ma **niejawnego odbiorcę**.

Załóżmy, że użytkownik w celu uzyskania pomocy kliknął widget w postaci przycisku o etykiecie „Drukuj”. Przycisk ten znajduje się w egzemplarzu klasy PrintDialog znającym obiekt aplikacji, do którego należy (zobacz wcześniejszy diagram obiektów). Poniższy diagram interakcji ilustruje, w jaki sposób żądanie pomocy jest przekazywane wzduł łańcucha:



W tym przypadku ani obiekt aPrintButton, ani obiekt aPrintDialog nie obsługuje żądania. Dochodzi ono do obiektu anApplication, który może je obsługiwać lub zignorować. Klient zgłoszający żądanie nie obejmuje bezpośredniej referencji do obiektu, który ostatecznie je obsługuje.

Przekazywanie żądania wzduł łańcucha i zapewnienie, że odbiorca pozostanie niewidzialny, wymaga, aby każdy obiekt w łańcuchu miał taki sam interfejs do obsługi żądań i dostępu do swojego **następnika** w łańcuchu. W systemie pomocy można na przykład zdefiniować klasę HelpHandler z odpowiednią operacją HandleHelp. HelpHandler może być klasą mieszaną lub klasą nadziedną dla klas obiektów kandydatów. Następnie klasy, które mają obsługiwać żądania pomocy, można utworzyć jako podklasy klasy HelpHandler:



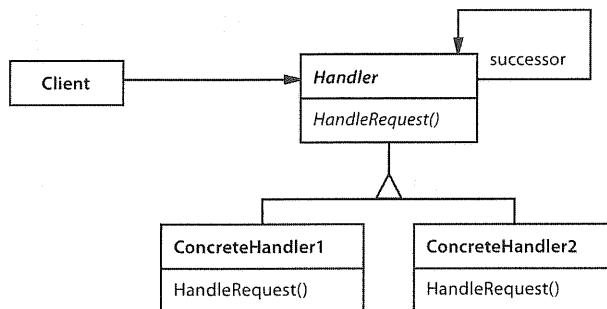
W klasach Button, Dialog i Application do obsługi żądań pomocy należy wykorzystać operacje klasy HelpHandler. Operacja HandleHelp klasy HelpHandler domyślnie przekazuje żądanie do następnika. W wybranych podkласach można przesłonić tę operację, aby we właściwych warunkach udostępnić pomoc. W pozostałych klasach można użyć implementacji domyślnej do przekazania żądania dalej.

WARUNKI STOSOWANIA

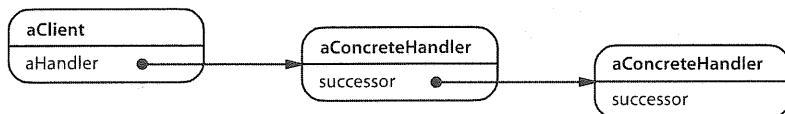
Wzorca Łańcuch zobowiązań należy używać w następujących warunkach:

- ▶ Kiedy więcej niż jeden obiekt może obsłużyć żądanie, a nie wiadomo z góry, który z nich to zrobi. Obiekt obsługujący żądanie powinien być ustalany automatycznie.
- ▶ Jeśli chcesz przesyłać żądanie do jednego z kilku obiektów bez bezpośredniego określania odbiorcy.
- ▶ Jeżeli zbiór obiektów, które mogą obsłużyć żądanie, należy określać dynamicznie.

STRUKTURA



Typowa struktura obiektów może wyglądać tak:



ELEMENTY

- ▶ **Handler (HelpHandler)**, czyli obiekt obsługujący:
 - definiuje interfejs do obsługi żądań;
 - (opcjonalnie) obejmuje implementację odwołania do następnika.
- ▶ **ConcreteHandler (PrintButton, PrintDialog)**, czyli konkretny obiekt obsługujący:
 - obsługuje żądania, za które odpowiada;
 - może uzyskać dostęp do następnika;
 - jeśli obiekt **ConcreteHandler** potrafi obsłużyć żądanie, robi to; w przeciwnym razie przekazuje je do następnika.
- ▶ **Client**:
 - inicjuje żądanie kierowane do obiektu **ConcreteHandler** wchodzącego w skład łańcucha.

WSPÓŁDZIAŁANIE

- Kiedy klient zgłosi żądanie, będzie ono przekazywane wzduż łańcucha do czasu, kiedy obiekt `ConcreteHandler` nie podejmie się jego obsługi.

KONSEKWENCJE

Wzorzec Łąćuch zobowiązań ma następujące zalety i wady:

7. *Zapewnia mniejsze powiązanie.* Wzorzec ten sprawia, że obiekty nie muszą wiedzieć, który z innych obiektów obsługuje żądanie. Obiekt potrzebuje jedynie informacji o tym, że żądanie zostanie właściwie obsłużone. Zarówno odbiorca, jak i nadawca nie wiedzą bezpośrednio o swoim istnieniu, a obiekty w łańcuchu nie znają jego struktury.
Powoduje to, że Łąćuch zobowiązań pozwala uprościć powiązania między obiektem. Zamiast przechowywać w obiekcie referencje do wszystkich potencjalnych odbiorców, wystarczy zapisać jedną referencję do następnika.
8. *Zwiększa elastyczność w zakresie przypisywania zadań obiektom.* Łąćuch zobowiązań zwiększa elastyczność przy podziale zadań między obiekty. Dodać lub zmodyfikować zadania związane z obsługą żądania można przez dołączenie obiektu do łańcucha lub zmodyfikowanie go w inny sposób w czasie wykonywania programu. Można to połączyć z tworzeniem podklas w celu statycznego budowania wyspecjalizowanych obiektów obsługujących żądania.
9. *Nie zapewnia gwarancji odbioru żądania.* Ponieważ żądanie nie ma jawnego odbiorcy, nie ma gwarancji, że zostanie obsłużone. Może ono zostać przekazane poza łańcuch bez wcześniejszego obsłużenia. Do braku obsługi może też doprowadzić niepoprawna konfiguracja łańcucha.

IMPLEMENTACJA

Oto kwestie implementacyjne, które należy rozważyć przy stosowaniu Łąćucha zobowiązań:

1. *Implementowanie łańcucha następników.* Łąćuch następników można zaimplementować na dwa sposoby:
 - a) przez zdefiniowanie nowych powiązań (zwykle w klasie `Handler`, ale można to zrobić także w klasach `ConcreteHandler`);
 - b) przez wykorzystanie istniejących powiązań.

W dotychczasowych przykładach definiowaliśmy nowe powiązania, jednak często do utworzenia łańcucha następników można wykorzystać istniejące referencje do obiektów. Na przykład do określenia następnika części w hierarchii część-całość można użyć referencji do elementu nadrzędnego. Struktura widgetu może standardowo obejmować potrzebne powiązania. Referencje do elementów nadrzędnych szczegółowo omawiamy w opisie wzorca Kompozyt (s. 170).

Wykorzystanie istniejących powiązań to dobre rozwiązanie, jeśli umożliwiają one zbudowanie potrzebnego łańcucha. Pozwala to zrezygnować z jawnego definiowania powiązań i zaoszczędzić pamięć. Jednak jeżeli dana struktura nie odzwierciedla łańcucha zobowiązań potrzebnego w aplikacji, trzeba zdefiniować dodatkowe powiązania.

2. *Łączenie następników.* Jeśli nie można zdefiniować łańcucha za pomocą istniejących referencji, trzeba je dodać samodzielnie. Wtedy klasa Handler nie tylko definiuje interfejs żądań, ale też zwykle przechowuje informacje o następcu. Umożliwia to umieszczenie w klasie Handler domyślnej implementacji operacji HandleRequest, przekazującej żądanie do następcu (jeśli taki istnieje). Jeżeli określona podkласa ConcreteHandler nie obsługuje danego żądania, nie trzeba przesyłać w niej operacji przekazywania, ponieważ domyślna implementacja bezwarunkowo przesyła żądanie dalej.

Oto klasa bazowa HelpHandler przechowująca powiązanie z następciem:

```
class HelpHandler {
public:
    HelpHandler(HelpHandler* s) : _successor(s) { }
    virtual void HandleHelp();
private:
    HelpHandler* _successor;
};

void HelpHandler::HandleHelp () {
    if (_successor) {
        _successor->HandleHelp();
    }
}
```

3. *Reprezentowanie żądań.* Żądania można reprezentować na różne sposoby. W najprostszej wersji żądanie jest zapisanym na stałe wywołaniem operacji (tak działa operacja HandleHelp). Jest to wygodne i bezpieczne, jednak pozwala przekazywać tylko stały zestaw żądań zdefiniowanych w klasie Handler.

Inna możliwość to zastosowanie pojedynczej funkcji obsługi żądania przyjmującej jako parametr kod żądania (na przykład stałą w postaci liczby całkowitej lub łańcucha znaków). To podejście pozwala stosować dowolnie duży zbiór żądań. Jedynym wymogiem jest to, że nadawca i odbiorca muszą uzgodnić sposób kodowania żądań.

To rozwiązanie jest elastyczniejsze, ale do przekazywania żądania trzeba zastosować instrukcję warunkową opartą na jego kodzie. Ponadto nie ma bezpiecznego ze względu na typ sposobu przekazywania parametrów, dlatego trzeba je pakować i wypakowywać ręcznie. Oczywiście jest to mniej bezpieczne niż bezpośrednie wywoływanie operacji.

Aby rozwiązać problem przekazywania parametrów, możemy użyć odrębnych obiektów żądań i umieścić w nich parametry żądania. Klasa Request będzie jawnie reprezentować żądania, a ich nowe rodzaje można zdefiniować przez utworzenie podklas tej klasy. Podklasy te mogą obejmować różne parametry. Obiekty obsługujące żądanie muszą znać jego rodzaj (czyli podklaś klasy Request), aby uzyskać dostęp do tych parametrów.

Na potrzeby identyfikowania żądań można w klasie Request zdefiniować akcesor zwracający identyfikator klasy. Inna możliwość to wykorzystanie przez odbiorcę informacji o typie w czasie wykonywania programu (jeśli języki użyte do implementacji to umożliwiają).

Oto zarys funkcji rozdzielającej żądania, która do ich identyfikowania korzysta z obiektów żądań. Operacja GetKind zdefiniowana w klasie bazowej Request określa rodzaj żądania:

```
void Handler::HandleRequest (Request* theRequest) {
    switch (theRequest->GetKind()) {
        case Help:
            // Rzutowanie argumentu na odpowiedni typ.
            HandleHelp((HelpRequest*) theRequest);
            break;

        case Print:
            HandlePrint((PrintRequest*) theRequest);
            // ...
            break;

        default:
            // ...
            break;
    }
}
```

W podklasach można rozszerzyć rozdzielanie żądań przez przesłonięcie operacji HandleRequest. Podklasy obsługują tylko te żądania, które są dla nich istotne. Pozostałe żądania należy przekazać do klasy nadzędnej. W ten sposób podklasy tak naprawdę rozszerzają operację HandleRequest, zamiast ją przesyłać. Na przykład w podklasie ExtendedHandler wersję operacji HandleRequest z klasy Handler rozszerzono w następujący sposób:

```
class ExtendedHandler : public Handler {
public:
    virtual void HandleRequest(Request* theRequest);
    // ...

void ExtendedHandler::HandleRequest (Request* theRequest) {
    switch (theRequest->GetKind()) {
        case Preview:
            // Obsługa żądania Preview.
            break;
        default:
            // Umożliwienie obsługi pozostałych żądań klasie Handler.
            Handler::HandleRequest(theRequest);
    }
}
```

4. *Automatyczne przekazywanie w języku Smalltalk.* Do przekazywania żądań można wykorzystać metodę doesNotUnderstand języka Smalltalk. Komunikaty, którym nie odpowiadają żadne metody, są przechwytywane w implementacji metody doesNotUnderstand. Można ją przesłonić, aby przekazywać komunikaty do następnika obiektu. Dzięki temu nie trzeba ręcznie implementować procesu przekazywania. Klasa będzie obsługiwać tylko istotne dla niej żądania, a za przekazywanie wszystkich pozostałych odpowiadać będzie metoda doesNotUnderstand.

PRZYKŁADOWY KOD

Poniższy przykład ilustruje, jak wykorzystać łańcuch zobowiązać do obsługi żądań w elektronicznych systemach pomocy, takich jak ten opisany wcześniej. Zgłoszenie żądania pomocy to jawną operacją. Do przekazywania żądań między widgetami z łańcucha użyjemy istniejących referencji do elementu nadrzędnego w hierarchii widgetów. Ponadto w klasie Handler zdefiniujemy referencję na potrzeby przekazywania żądań między tymi elementami łańcucha, które nie są widgetami.

Klasa HelpHandler definiuje interfejs do obsługi żądań pomocy. Przechowuje temat pomocy (domyślnie jest on pusty) i referencję do następnika w łańcuchu obiektów obsługujących żądania. Kluczową operacją jest tu HandleHelp (podklasy obejmują jej przesłonięte wersje). HasHelp to operacja pomocnicza sprawdzająca, czy istnieje temat pomocy powiązany z danym żądaniem.

```
typedef int Topic;
const Topic NO_HELP_TOPIC = -1;

class HelpHandler {
public:
    HelpHandler(HelpHandler* _successor, Topic _topic);
    virtual bool HasHelp();
    virtual void SetHandler(HelpHandler*, Topic);
    virtual void HandleHelp();
private:
    HelpHandler* _successor;
    Topic _topic;
};

HelpHandler::HelpHandler (
    HelpHandler* h, Topic t
) : _successor(h), _topic(t) { }

bool HelpHandler::HasHelp () {
    return _topic != NO_HELP_TOPIC;
}

void HelpHandler::HandleHelp () {
    if (_successor != 0) {
        _successor->HandleHelp();
    }
}
```

Wszystkie widgety to podklasy klasy abstrakcyjnej Widget. Z kolei Widget to podkلاsa klasy HelpHandler, ponieważ informacje z systemu pomocy mogą być dostępne dla wszystkich elementów interfejsu użytkownika. W tym miejscu moglibyśmy zastosować także implementację opartą na klasach mieszanych.

```
class Widget : public HelpHandler {
protected:
    Widget(Widget* parent, Topic t = NO_HELP_TOPIC);
private:
```

```

        Widget* _parent;
};

Widget::Widget (Widget* w, Topic t) : HelpHandler(w, t) {
    _parent = w;
}

```

W tym przykładzie przycisk to pierwszy obiekt obsługujący żądanie w łańcuchu. Klasa Button jest podklassą klasy Widget. Konstruktor klasy Button przyjmuje dwa parametry — referencję do zawierającego przycisk widgetu i temat pomocy.

```

class Button : public Widget {
public:
    Button(Widget* d, Topic t = NO_HELP_TOPIC);

    virtual void HandleHelp();
    // Operacje klasy Widget przesłonięte w klasie Button.
};

```

Wersja operacji HandleHelp z klasy Button najpierw sprawdza, czy istnieje temat pomocy dotyczący przycisków. Jeśli programista go nie przygotował, żądanie jest przekazywane do następnika za pomocą operacji HandleHelp z klasy HelpHandler. Jeśli *istnieje* odpowiedni temat pomocy, klasa Button wyświetla go, a wyszukiwanie zostaje zakończone.

```

Button::Button (Widget* h, Topic t) : Widget(h, t) { }

void Button::HandleHelp () {
    if (HasHelp()) {
        // Udostępnia informacje na temat przycisku.
    } else {
        HelpHandler::HandleHelp();
    }
}

```

W klasie Dialog zastosowaliśmy podobne rozwiązanie, jednak tu następnikiem nie jest widget, ale *dowolny* obiekt obsługujący żądania pomocy. W omawianej aplikacji takim następnikiem będzie egzemplarz klasy Application.

```

class Dialog : public Widget {
public:
    Dialog(HelpHandler* h, Topic t = NO_HELP_TOPIC);
    virtual void HandleHelp();

    // Operacje z klasy Widget przesłonięte w klasie Dialog.
    // ...
};

Dialog::Dialog (HelpHandler* h, Topic t) : Widget(0) {
    SetHandler(h, t);
}

void Dialog::HandleHelp () {
    if (HasHelp()) {

```

```

    // Udostępnia informacje o oknie dialogowym.
} else {
    HelpHandler::HandleHelp();
}
}

```

Na końcu łańcucha znajduje się egzemplarz klasy Application. Aplikacja nie jest widgetem, dlatego Application to bezpośrednia podklasa klasy HelpHandler. Jeśli żądanie pomocy dojdzie do końca łańcucha, aplikacja może udostępnić ogólne informacje na swój temat lub wyświetlić listę różnych tematów pomocy:

```

class Application : public HelpHandler {
public:
    Application(Topic t) : HelpHandler(0, t) { }

    virtual void HandleHelp();
    // Operacje specyficzne dla aplikacji.
};

void Application::HandleHelp () {
    // Wyświetlanie listy tematów pomocy.
}

```

Poniższy kod tworzy i łączy wymienione obiekty. W tym przypadku okno dialogowe dotyczy drukowania, dlatego obiekty są powiązane z tematami z zakresu drukowania.

```

const Topic PRINT_TOPIC = 1;
const Topic PAPER_ORIENTATION_TOPIC = 2;
const Topic APPLICATION_TOPIC = 3;

Application* application = new Application(APPLICATION_TOPIC);
Dialog* dialog = new Dialog(application, PRINT_TOPIC);
Button* button = new Button(dialog, PAPER_ORIENTATION_TOPIC);

```

Żądanie pomocy można zgłosić przez wywołanie operacji HandleHelp dla dowolnego obiektu z łańcucha. Aby rozpocząć wyszukiwanie od obiektu reprezentującego przycisk, wystarczy wywołać jego operację HandleHelp:

```
button->HandleHelp();
```

Tu przycisk natychmiast obsługuje żądanie. Warto zauważyć, że następcą klasy Dialog może być dowolna klasa z rodziny HelpHandler. Ponadto następnik można zmieniać dynamicznie. Dlatego niezależnie od miejsca użycia okna dialogowego aplikacja wyświetli odpowiednie zależne od kontekstu informacje na jego temat.

ZNANE ZASTOSOWANIA

W kilku bibliotekach klas wzorzec Łąncuch zobowiązań zastosowano do obsługi zdarzeń wywoływanych przez użytkownika. Odpowiedniki klasy Handler mają różne nazwy, ale ogólna zasada ich działania jest taka sama — kiedy użytkownik kliknie przycisk myszy lub wciśnie klawisz, zdarzenie zostanie wygenerowane i przekazane wzdułż łańcucha. W MacApp [App89] i ET++ [WGM88] klasa ta nosi nazwę EventHandler, w bibliotece TCL firmy Symantec [Sym93b] jest to klasa Bureaucrat, a w AppKit firmy NeXT [Add94] — Responder.

W platformie Unidraw (służy ona do tworzenia edytorów graficznych) zdefiniowane są obiekty Command. Kapsułkują one żądania kierowane do obiektów Component i ComponentView [VL90]. Obiekty Command to żądania w tym sensie, że komponent lub widok komponentu może w wyniku interpretacji polecenia wykonać operację. Odpowiada to technice tworzenia żądań jako obiektów opisanej w punkcie „Implementacja”. Komponenty i widoki komponentów mogą mieć strukturę hierarchiczną. Komponent lub widok może przekazać zadanie interpretacji polecenia do elementu nadzawanego, który przekaże je do swojego elementu nadzawanego i tak dalej, co prowadzi do powstania łańcucha zobowiązań.

W platformie ET++ wzorzec łańcucha zobowiązań zastosowano do obsługi aktualizowania elementów graficznych. Obiekt graficzny zawsze, kiedy musi zaktualizować część swojego wyglądu, wywołuje operację `InvalidateRect`. Taki obiekt nie może samodzielnie obsłużyć operacji `InvalidateRect`, ponieważ nie ma wystarczających informacji o kontekście. Obiekt graficzny może na przykład znajdować się w obiektach `Scroller` lub `Zoomer`, które zmieniają jego układ współrzędnych. Oznacza to, że obiekt można przewinąć lub przybliżyć, tak że stanie się częściowo niewidoczny. Dlatego domyślna implementacja operacji `InvalidateRect` przekazuje żądanie do zewnętrznego obiektu kontenerowego. Ostatnim obiektem w łańcuchu jest egzemplarz klasy `Window`. Do czasu dotarcia żądania do takiego egzemplarza współrzędne odświeżanego prostokąta zawsze są prawidłowo przekształcane. Klasa `Window` obsługuje operację `InvalidateRect` przez przekazanie powiadomienia do interfejsu systemu okienkowego i żądanie zaktualizowania obrazu.

POWIĄZANE WZORCE

Łańcuch zobowiązań często stosuje się razem ze wzorcem Kompozyt (s. 170). Wtedy obiekt nadzawany komponentu może pełnić funkcję jego następnika.

MEDIATOR (MEDIATOR)

obiektowy, operacyjny

PRZEZNACZENIE

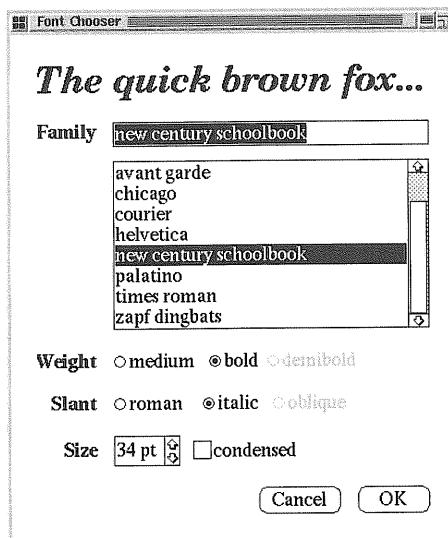
Określa obiekt kapsułkujący informacje o interakcji między obiektem a innymi obiektami z danego zbioru. Wzorzec ten pomaga zapewnić luźne powiązanie, ponieważ zapobiega bezpośredniemu odwoływaniu się obiektów do siebie i umożliwia niezależne modyfikowanie interakcji między nimi.

UZASADNIENIE

Projekt obiektowy zachęca do rozmieszczania zachowań w różnych obiektach. Może to prowadzić do powstania struktury obiektów z wieloma powiązaniami. W najgorszym przypadku każdy obiekt wie o istnieniu wszystkich pozostałych.

Choć podział systemu na wiele obiektów zwykle ułatwia jego powtórne wykorzystanie, liczne powiązania przeważnie je utrudniają. Duża liczba połączeń zmniejsza prawdopodobieństwo tego, że obiekt będzie działał bez pozostałych. System działa wtedy jak monolit. Ponadto znacząca zmiana działania systemu może być trudna, ponieważ zachowanie jest rozproszone między wiele obiektów. Powoduje to, że czasem konieczne jest zdefiniowanie wielu podklas w celu dostosowania działania systemu do potrzeb.

Rozważmy na przykład implementację okien dialogowych w graficznym interfejsie użytkownika. Okno dialogowe wykorzystuje zwykłe okno do wyświetlania kolekcji widgetów, takich jak przyciski, menu i pola na dane wejściowe, jak ilustruje to poniższy rysunek:



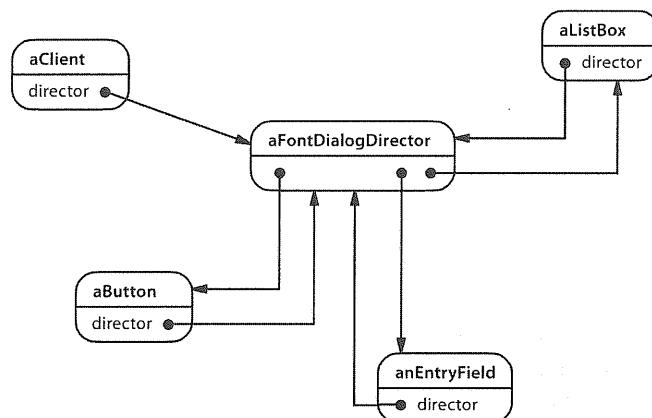
Miedzy widgetami z okna dialogowego często występują zależności. Na przykład przycisk może być nieaktywny, jeśli określone pole na dane wejściowe jest puste. Zaznaczenie pozycji na liście opcji (tak zwanej **liście rozwijanej**) może spowodować zmianę zawartości pola na

dane wejściowe. I na odwrót — wprowadzenie tekstu w takim polu może spowodować automatyczne zaznaczenie jednej lub kilku pozycji na liście rozwijanej. Kiedy w polu na dane wejściowe pojawi się tekst, system może udostępnić inne przyciski umożliwiające użytkownikowi wykorzystanie tego tekstu, na przykład zmodyfikowanie lub usunięcie elementu, którego on dotyczy.

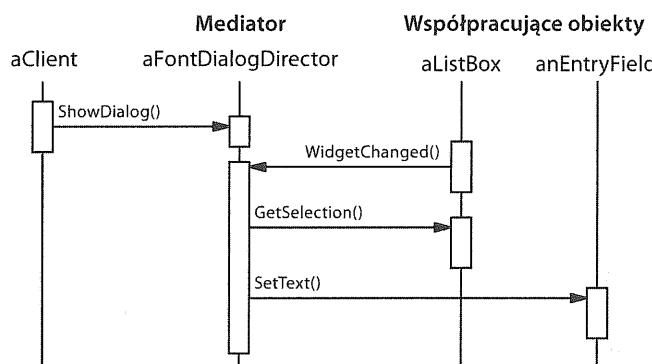
W poszczególnych oknach dialogowych zależności między widgetami mogą być różne. Dlatego choć okna dialogowe wyświetlają widgety tego samego rodzaju, nie można w nich łatwo powtórnie wykorzystać gotowych klas widgetów. Klasy te trzeba dostosować do zależności specyficznych dla okna dialogowego. Modyfikowanie poszczególnych klas przez tworzenie ich podklas jest żmudne z uwagi na ich liczbę.

Można uniknąć tych problemów przez zakapsułkowanie wspólnego zachowania w odrębnym obiekcie **mediatora**. Mediator odpowiada za kontrolowanie i koordynowanie interakcji w grupie obiektów. Działa jak pośrednik, dzięki któremu obiekty w grupie nie muszą odwoływać się do siebie bezpośrednio. Obiekty znają jedynie mediatora, co zmniejsza liczbę powiązań między nimi.

Na przykład klasa **FontDialogDirector** może być mediatorem pomiędzy widgetami z okna dialogowego. Obiekt **FontDialogDirector** zna widgety z okna i koordynuje interakcję między nimi. Obiekt ten działa jak centrala komunikacyjna dla widgetów:



Poniższy diagram interakcji ilustruje, w jaki sposób obiekty współpracują ze sobą przy obsłudze zmieniania opcji na liście rozwijanej:

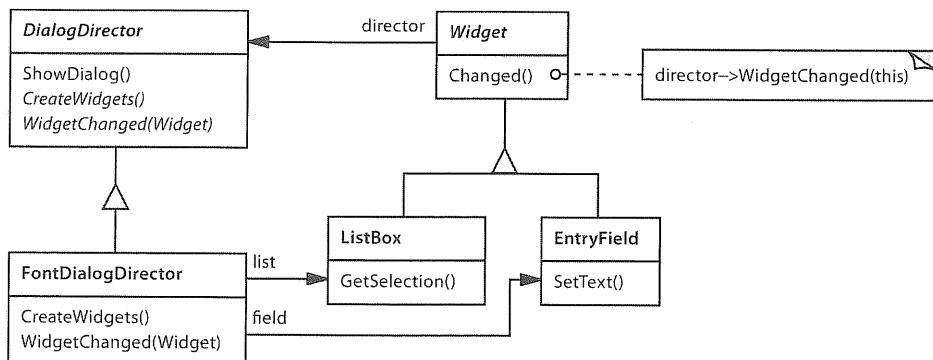


Oto kolejne zdarzenia, które powodują zapisanie w polu na dane wejściowe opcji wybranej na liście rozwijanej:

1. Lista rozwijana (obiekt `aListBox`) informuje kierownika (obiekt `director`) o zmianie.
2. Kierownik pobiera wybraną opcję z listy rozwijanej.
3. Kierownik przekazuje wybraną opcję do pola na dane wejściowe (`anEntryField`).
4. Kiedy pole na dane wejściowe zawiera tekst, kierownik włącza przyciski do inicjowania działań (na przykład „demibold” lub „oblique”).

Warto zauważyć, w jaki sposób kierownik pośredniczy w komunikacji między listą rozwijaną i polem na dane wejściowe. Widgety komunikują się ze sobą tylko pośrednio, poprzez kierownika. Nie muszą wiedzieć o swoim istnieniu i znają tylko kierownika. Ponadto z uwagi na umieszczenie zachowania w jednej klasie można je zmienić lub zastąpić przez rozszerzenie lub podmienienie klasy.

Oto, w jaki sposób abstrakcję `FontDialogDirector` można zintegrować z biblioteką klas:



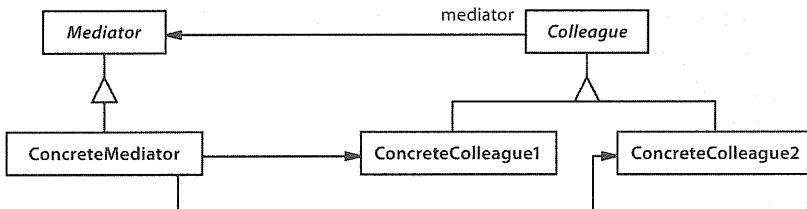
`DialogDirector` to klasa abstrakcyjna definiująca ogólne zachowanie okna dialogowego. Klienci wywołują operację `ShowDialog`, aby wyświetlić okno na ekranie. `CreateWidgets` to operacja abstrakcyjna do tworzenia widgetów okna dialogowego. `WidgetChanged` to następna operacja abstrakcyjna. Widgety wywołują ją, aby poinformować kierownika o zmianie. W podklasach klasy `DialogDirector` operacja `CreateWidgets` jest przesłonięta, tak aby tworzyła odpowiednie widgety. Przesłonięta jest też operacja `WidgetChanged`, co umożliwia obsługę wprowadzonych zmian.

WARUNKI STOSOWANIA

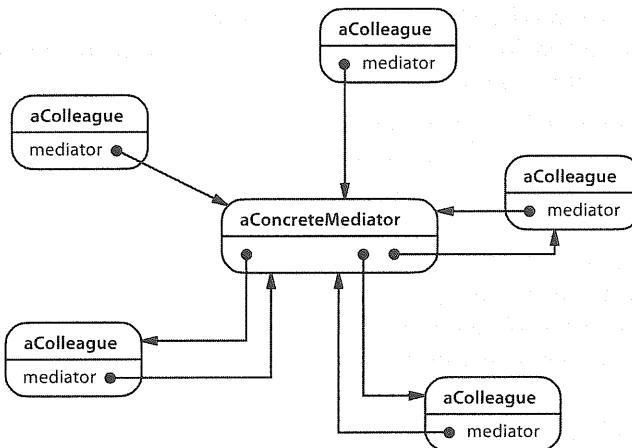
Oto warunki, w których należy stosować wzorzec Mediator:

- ▶ Zestaw obiektów komunikuje się w dobrze zdefiniowany, ale skomplikowany sposób. Powstałe w wyniku tego zależności są niestrukturyzowane i trudne do zrozumienia.
- ▶ Powtórne wykorzystanie obiektu jest trudne, ponieważ odwołuje się on do wielu innych obiektów i komunikuje się z nimi.
- ▶ Dostosowanie zachowania rozproszonego po kilku klasach nie powinno wymagać tworzenia wielu podklas.

STRUKTURA



Typowa struktura obiektów może wyglądać tak:



ELEMENTY

- ▶ **Mediator (DialogDirector):**
 - definiuje interfejs do komunikowania się z obiektami współpracującymi.
- ▶ **ConcreteMediator (FontDialogDirector):**
 - zapewnia współdziałanie przez koordynowanie obiektów współpracujących;
 - zna powiązane z nim obiekty współpracujące i zarządza nimi.
- ▶ **Klasy z rodziną Colleague (ListBox, EntryField), czyli klasy współpracujące:**
 - każdy obiekt współpracujący zna powiązany z nią obiekt Mediator;
 - każdy obiekt współpracujący komunikuje się z mediatorem zamiast z innymi takimi obiektami.

WSPÓŁDZIAŁANIE

- ▶ Obiekty współpracujące wysyłają i przyjmują żądania za pośrednictwem obiektu Mediator. Mediator zapewnia współdziałanie przez przekazywanie żądań między odpowiednimi współpracownikami.

KONSEKWENCJE

Wzorzec Mediator ma następujące zalety i wady:

1. *Ogranicza tworzenie podklas.* Mediator pozwala umieścić w jednym miejscu zachowanie, które standardowo trzeba rozproszyć między kilka obiektów. Zmodyfikowanie tego zachowania wymaga utworzenia podklasy tylko dla klasy Mediator. Obiekty współpracujące można powtórnie wykorzystać w ich pierwotnej postaci.
2. *Rozdziela obiekty współpracujące.* Mediator ułatwia zachowanie luźnego powiązania między obiektami współpracującymi. Klasy Colleague i Mediator można modyfikować oraz powtórnie użytkować niezależnie od siebie.
3. *Upraszczają protokoły obiektów.* Mediator pozwala zastąpić interakcje typu wiele do wielu komunikacją typu jeden do wielu (między mediatorem i powiązanymi z nim współpracownikami). Relacje jeden do wielu są łatwiejsze do zrozumienia, konserwowania i wzbogacania.
4. *Ujmuje abstrakcyjne współdziałanie obiektów.* Potraktowanie pośrednictwa jak niezależnego elementu i zakapsułkowanie tego procesu w obiekcie pozwala skoncentrować się na interakcji między obiektami w oderwaniu od ich zachowania. Może to pomóc w precyzyjniejszym przedstawieniu współdziałania obiektów w systemie.
5. *Centralizuje sterowanie.* Wzorzec Mediator pozwala zamienić złożoność interakcji na złożoność samego mediatora. Ponieważ mediator kapsułkuje protokoły, może być bardziej złożony niż którykolwiek z poszczególnych współpracowników. Może to spowodować, że sam mediator stanie się trudnym w konserwacji monolitem.

IMPLEMENTACJA

Ze wzorcem Mediator związane są poniższe zagadnienia implementacyjne:

1. *Pominiecie klasy abstrakcyjnej Mediator.* Jeśli obiekty współpracujące korzystają z tylko jednego mediatora, nie trzeba definiować klasy abstrakcyjnej Mediator. Abstrakcyjne powiązanie zapewniane przez klasę Mediator umożliwia współpracownikom korzystanie z różnych podklas klasy Mediator i na odwrót.
2. *Komunikacja między współpracownikami i mediatorem.* Obiekty współpracujące muszą komunikować się z mediatorem, kiedy wystąpi istotne zdarzenie. Jedną z możliwości jest zaimplementowanie klasy Mediator jako obserwatora (zobacz wzorzec Obserwator, s. 269). Klasa współpracownika pełni funkcję podmiotów i kiedy zmieni się ich stan, wysyłają do mediatora powiadomienia. Mediator reaguje na nie przez przekazanie efektów zmiany do innych współpracowników.

Inne rozwiązanie polega na zdefiniowaniu w klasie Mediator wyspecjalizowanego interfejsu do powiadamiania umożliwiającego współpracownikom bardziej bezpośrednią komunikację. W języku Smalltalk/V dla systemu Windows zastosowano pewną formę delegowania. Przy komunikowaniu się z mediatorem współpracownik przekazuje sam siebie jako argument, co umożliwia mediatorowi zidentyfikowanie nadawcy. Tę technikę zastosowaliśmy w punkcie „Przykładowy kod”, a implementację z języka Smalltalk/V omawiamy dokładniej w punkcie „Znane zastosowania”.

PRZYKŁADOWY KOD

W klasie `DialogDirector` zaimplementujemy okno dialogowe do zmiany czcionek pokazane w punkcie „Uzasadnienie”. Klasa abstrakcyjna `DialogDirector` definiuje interfejs kierownika.

```
class DialogDirector {
public:
    virtual ~DialogDirector();

    virtual void ShowDialog();
    virtual void WidgetChanged(Widget*) = 0;

protected:
    DialogDirector();
    virtual void CreateWidgets() = 0;
};
```

`Widget` to abstrakcyjna klasa bazowa widgetów. `Widget` zna powiązanego z nim kierownika.

```
class Widget {
public:
    Widget(DialogDirector*);
    virtual void Changed();

    virtual void HandleMouse(MouseEvent& event);
    // ...
private:
    DialogDirector* _director;
};
```

Operacja `Changed` wywołuje operację `WidgetChanged` kierownika. Widgety wywołują operację `WidgetChanged` kierownika, aby poinformować go o istotnym zdarzeniu.

```
void Widget::Changed () {
    _director->WidgetChanged(this);
}
```

W podklasach klasy `DialogDirector` należy przesłonić operację `WidgetChanged`, aby oddziaływała na odpowiednie widgety. Widget przekazuje referencję do samego siebie jako argument operacji `WidgetChanged`, co pozwala kierownikowi zidentyfikować zmieniony widget. W podklasach klasy `DialogDirector` czysto wirtualna operacja `CreateWidgets` jest przeddefiniowana, tak żeby tworzyła widgety w danym oknie dialogowym.

`ListBox`, `EntryField` i `Button` to podklasy klasy `Widget` reprezentujące wyspecjalizowane elementy interfejsu użytkownika. Klasa `ListBox` udostępnia operację `GetSelection` do pobierania zaznaczonej opcji, a operacja `SetText` klasy `EntryField` umieszcza nowy tekst w polu na dane wejściowe.

```
class ListBox : public Widget {
public:
    ListBox(DialogDirector*);

    virtual const char* GetSelection();
```

```

        virtual void SetList(List<char*>* listItems);
        virtual void HandleMouse(MouseEvent& event);
        // ...
    };

    class EntryField : public Widget {
public:
    EntryField(DialogDirector*);

    virtual void SetText(const char* text);
    virtual const char* GetText();
    virtual void HandleMouse(MouseEvent& event);
    // ...
};


```

Button to prosty widget wywołujący operację Changed przy każdym wciśnięciu przycisku. Wywołanie to znajduje się w implementacji operacji HandleMouse w tej klasie:

```

class Button : public Widget {
public:
    Button(DialogDirector*);

    virtual void SetText(const char* text);
    virtual void HandleMouse(MouseEvent& event);
    // ...
};

void Button::HandleMouse (MouseEvent& event) {
    // ...
    Changed();
}

```

Klasa FontDialogDirector (jest to podklasa klasy DialogDirector) pośredniczy w komunikacji między widgetami okna dialogowego:

```

class FontDialogDirector : public DialogDirector {
public:
    FontDialogDirector();
    virtual ~FontDialogDirector();
    virtual void WidgetChanged(Widget*);

protected:
    virtual void CreateWidgets();

private:
    Button* _ok;
    Button* _cancel;
    ListBox* _fontList;
    EntryField* _fontName;
};


```

Klasa `FontDialogDirector` śledzi wyświetlane przez nią widgety. Operacja `CreateWidgets` jest w niej przedefiniowana, tak aby tworzyła widgety i inicjowała prowadzące do nich referencje:

```
void FontDialogDirector::CreateWidgets () {
    _ok = new Button(this);
    _cancel = new Button(this);
    _fontList = new ListBox(this);
    _fontName = new EntryField(this);

    // Zapelnianie pola listBox nazwami dostepnych czcionek.

    // Laczenie widgetow w okno dialogowe.
}
```

Operacja `WidgetChanged` gwarantuje, że widgety będą prawidłowo współdziałać ze sobą:

```
void FontDialogDirector::WidgetChanged (
    Widget* theChangedWidget
) {
    if (theChangedWidget == _fontList) {
        _fontName->SetText(_fontList->GetSelection());

    } else if (theChangedWidget == _ok) {
        // Modyfikowanie czcionki i zamkniecie okna dialogowego.
        // ...
    } else if (theChangedWidget == _cancel) {
        // Zamkniecie okna dialogowego.
    }
}
```

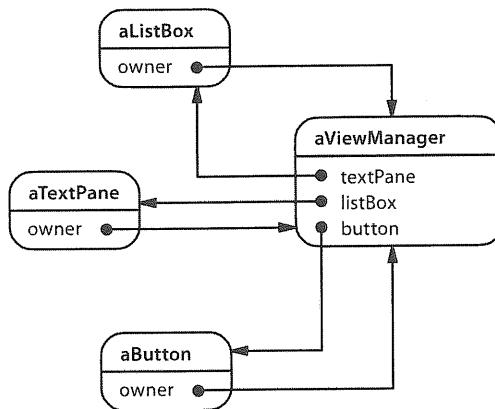
Złożoność operacji `WidgetChanged` rośnie proporcjonalnie do złożoności okna dialogowego. Rozbudowane okna dialogowe są oczywiście niepożądane także z innych przyczyn, jednak złożoność mediatora może zmniejszyć przydatność omawianego wzorca w innych zastosowaniach.

ZNANE ZASTOSOWANIA

Zarówno w platformie ET++ [WGM88], jak i w bibliotece klas THINK języka C [Sym93b] w oknach dialogowych zastosowano obiekty przypominające kierownika jako mediatory pośredniczące między widgetami.

Architektura aplikacji w języku Smalltalk/V dla systemu Windows jest oparta na strukturze mediatora [LaL94]. W tym środowisku aplikacja składa się z okna obejmującego zbiór pól. Biblioteka zawiera kilka gotowych reprezentujących pola obiektów Pane — `TextPane`, `ListBox`, `Button` itd. Można z nich korzystać bez tworzenia podklas. Autor aplikacji przygotowuje tylko podklasy klasy `ViewManager` odpowiadającej za koordynowanie interakcji między polami. Klasa `ViewManager` pełni funkcję mediatora, a każde pole zna tylko powiązany z nim obiekt `ViewManager`, który jest uważany za „właściciela” danego pola. Pola nie odwołują się bezpośrednio do siebie.

Poniższy diagram obiektów przedstawia obraz aplikacji w czasie wykonywania programu:



W języku Smalltalk/V do komunikacji między obiektami Pane i ViewManager służą zdarzenia. Pole generuje zdarzenie, kiedy chce pobrać informacje od mediatora lub poinformować go o tym, że zdarzyło się coś ważnego. W zdarzeniu zdefiniowany jest symbol (na przykład `#select`) identyfikujący dane zdarzenie. Aby obsłużyć określone zdarzenie, w obiekcie ViewManager trzeba zarejestrować selektor metod powiązany z wybranym polem. Ten selektor to mechanizm obsługi zdarzeń wywoływanego przy każdym wystąpieniu danego zdarzenia.

Poniższy fragment kodu pokazuje, w jaki sposób w podklasie klasy ViewManager powstaje obiekt ListPane i jak obiekt ViewManager rejestruje mechanizm obsługi zdarzenia `#select`:

```

self addSubpane: (ListPane new
    paneName: 'myListPane';
    owner: self;
    when: #select perform: #listSelect:).
  
```

Wzorzec Mediator można też wykorzystać do koordynowania złożonych aktualizacji. Służy do tego na przykład klasa ChangeManager z opisu wzorca Obserwator (s. 269). Klasa ta pośredniczy w komunikacji między podmiotami i obserwatorami, co pozwala uniknąć zbędnych aktualizacji. Kiedy dany obiekt się zmieni, powiadomi o tym obiekt ChangeManager, a ten skoordynuje aktualizację przez powiadomienie jednostek zależnych od zmodyfikowanego obiektu.

W podobny sposób wzorzec Mediator zastosowano w platformie do tworzenia edytorów graficznych Unidraw [VL90]. Wykorzystano w niej klasę CSolver do wymuszania ograniczeń łączności między konektorami. Obiekty w edytorek graficznych mogą wyglądać jak przyklejone do siebie. Konektory są przydatne w aplikacjach, które automatycznie podtrzymują takie połączenia, na przykład w edytorek diagramów i systemach do projektowania obwodów elektronicznych. CSolver pełni funkcję mediatora pomiędzy konektorami. Realizuje ograniczenia łączności i aktualizuje pozycje konektorów pod kątem tych ograniczeń.

POWIĄZANE WZORCE

Fasada (s. 161) różni się od wzorca Mediator, ponieważ pozwala abstrakcyjnie przedstawić podsystem obiektów i udostępnić dla niego wygodniejszy interfejs. Protokół fasady jest jednokierunkowy. Oznacza to, że obiekty reprezentujące fasadę mogą kierować żądania do klas podsystemu, ale już nie na odwrót. Wzorzec Mediator działa inaczej — umożliwia wspólne realizowanie zachowań, których obiekty współpracujące nie udostępniają lub nie mogą obsługiwać. Ponadto protokół jest tu wielokierunkowy.

Obiekty współpracujące mogą komunikować się z mediatorem za pomocą wzorca Obserwator (s. 269).

METODA SZABLONOWA (TEMPLATE METHOD)

klasowy, operacyjny

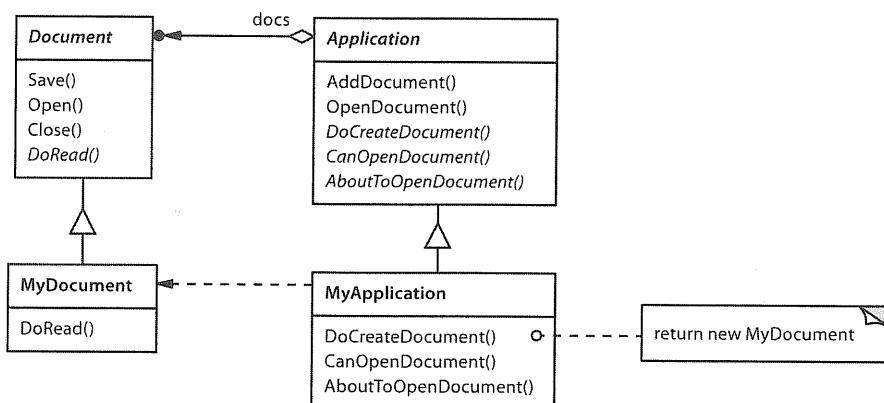
PRZEZNACZENIE

Określa szkielet algorytmu i pozostawia doprecyzowanie niektórych jego kroków podklassom. Umożliwia modyfikację niektórych etapów algorytmu w podklassach bez zmiany jego struktury.

UZASADNIENIE

Zastanówmy się nad platformą do tworzenia aplikacji udostępniającą klasy Application i Document. Klasa Application odpowiada za otwieranie istniejących dokumentów zapisanych w formacie zewnętrznym, na przykład w pliku. Obiekt Document reprezentuje informacje z dokumentu po ich wczytaniu z pliku.

W aplikacjach budowanych za pomocą wspomnianej platformy można utworzyć podklasy klas Application i Document dostosowane do specyficznych potrzeb. Na przykład w aplikacji do rysowania zdefiniowane zostaną podklasy DrawApplication i DrawDocument, a w aplikacji do przetwarzania arkuszy kalkulacyjnych — podklasy SpreadsheetApplication i SpreadsheetDocument.



Algorytm do otwierania i wczytywania dokumentu zdefiniowany jest w operacji OpenDocument klasy abstrakcyjnej Application:

```

void Application::OpenDocument (const char* name) {
    if (!CanOpenDocument(name)) {
        // Nie potrafi obsługiwać danego dokumentu.
        return;
    }

    Document* doc = DoCreateDocument();

    if (doc) {
        _docs->AddDocument(doc);
        AboutToOpenDocument(doc);
        doc->Open();
    }
}
  
```

```
    doc->DoRead();  
}
```

Operacja OpenDocument definiuje każdy etap otwierania dokumentu. Sprawdza, czy dokument można otworzyć, generuje specyficzny dla aplikacji obiekt Document, dodaje go do zestawu dokumentów i wczytuje treść dokumentu z pliku.

OpenDocument to tak zwana **metoda szablonowa**. Definiuje ona algorytm w kategoriach operacji abstrakcyjnych przesyłanych w podklasach w celu udostępniania konkretnego zachowania. Podklasy klasy Application definiują etapy algorytmu, który sprawdza, czy dokument można otworzyć (operacja CanOpenDocument), oraz tworzy obiekt Document (operacja DoCreateDocument). Klasy Document definiują proces wczytywania dokumentu (DoRead). Metoda szablonowa definiuje też operację AboutToOpenDocument powiadamiającą podklasy klasy Application o tym, kiedy dokument będzie otwierany (jeśli ma to znaczenie).

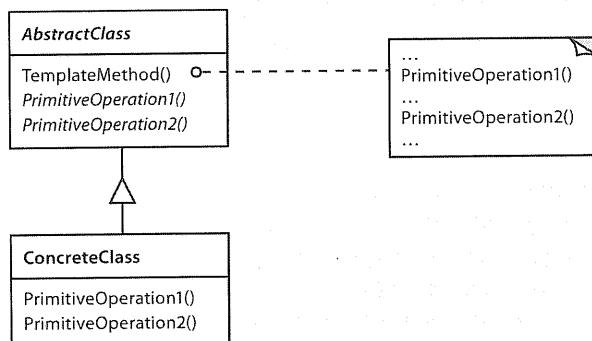
Przez zdefiniowanie wybranych etapów algorytmu za pomocą operacji abstrakcyjnych można w metodzie szablonowej określić ich kolejność, a jednocześnie dostosować w podklasach klas Application i Document kroki algorytmu do potrzeb tych podklas.

WARUNKI STOSOWANIA

Ze wzorca Metoda szablonowa należy korzystać w następujących warunkach:

- ▶ Do jednorazowego implementowania niezmiennych części algorytmu i umożliwienia implementowania zmieniających się zachowań w podklasach.
 - ▶ Kiedy zachowanie wspólne dla podklas należy wyodrębnić i umieścić w jednej klasie, aby uniknąć powielania kodu. Jest to dobry przykład zastosowania podejścia „refaktoryzacja w celu uogólniania”, opisanego przez Opdyke’ a i Johnsona [OJ93]. Najpierw należy wykryć różnice w istniejącym kodzie, a następnie umieścić je w nowych operacjach. Na zakończenie trzeba zastąpić różniące się fragmenty kodu metodą szablonową wywołującą jedną z nowych operacji.
 - ▶ Do kontrolowania rozszerzania podklas. Można zdefiniować metodę szablonową wywołującą w odpowiednich miejscach operacje stanowiące „punkty zaczepienia” (zobacz punkt „Konsekwencje”), co umożliwia rozszerzanie podklas tylko w tych punktach.

STRUKTURA



ELEMENTY

► **AbstractClass** (Application):

- określa abstrakcyjne **operacje proste** definiowane w podklasach konkretnych w celu zaimplementowania etapów algorytmu;
- obejmuje implementację metody szablonowej definiującej szkielet algorytmu; metoda szablonowa wywołuje operacje proste, a także operacje zdefiniowane w klasie **AbstractClass** lub w innych klasach.

► **ConcreteClass** (MyApplication):

- obejmuje implementację operacji prostych realizujących specyficzne dla podklasy etapy algorytmu.

WSPÓŁDZIAŁANIE

- Obiekt **ConcreteClass** polega na obiekcie **AbstractClass** w zakresie implementacji nieniemiennych kroków algorytmu.

KONSEKWENCJE

Metody szablonowe to podstawowa technika powtórnego wykorzystania kodu. Są szczególnie istotne w bibliotekach klas, ponieważ stanowią narzędzie do wyodrębniania wspólnego zachowania w klasach biblioteki.

Metody szablonowe prowadzą do odwrócenia struktury sterowania. Czasem nazywa się to „zasadą Hollywoodu” lub podejściem „nie dzwoń do nas — to my zadzwonimy do ciebie” [Swe85]. Wynika to z tego, że klasa nadziedziona wywołuje operacje podklasy, a nie na odwrót.

Metody szablonowe wywołują operacje następujących rodzajów:

- operacje konkretne (albo klasy **ConcreteClass**, albo klas klienta);
- konkretne operacje klasy **AbstractClass** (to znaczy operacje przydatne w podklasach);
- operacje proste (czyli operacje abstrakcyjne);
- metody wytwarzające (zobacz wzorzec Metoda wytwarzająca, s. 110);
- **operacje stanowiące „punkty zaczepienia”**. Udostępniają one zachowanie domyślne, które w razie potrzeby można rozszerzyć w podklasach (domyślnie operacje stanowiące „punkty zaczepienia” nie wykonują żadnych działań).

Ważne jest, aby w metodzie szablonowej określić, które operacje to „punkty zaczepienia” (*można je przesłonić*), a które są operacjami abstrakcyjnymi (*trzeba je przesłonić*). Aby skutecznie powtórnie wykorzystać klasę abstrakcyjną, autor podklasy musi zrozumieć, które operacje zaprojektowano w celu ich przesłonięcia.

W podklasie można *rozszerzyć* działanie operacji z klasy nadziedznej przez przesłonięcie tej operacji i jawne wywołanie jej wersji z klasy nadziedznej:

```
void DerivedClass::Operation () {
    ParentClass::Operation();
    // Zachowanie rozszerzone w klasie DerivedClass.
}
```

Niestety, łatwo jest zapomnieć o wywołaniu odziedziczonej operacji. Można przekształcić taką operację w metodę szablonową, aby umożliwić kontrolowanie w klasie nadzędnej rozszerzania tej operacji w podklasach. Pomysł polega na wywoływaniu operacji stanowiących „punkt zaczepienia” w metodzie szablonowej w klasie nadzędnej. Następnie w podklasach można przesłonić taką operację:

```
void ParentClass::Operation () {
    // Zachowanie z klasy ParentClass.
    HookOperation();
}
```

Operacja `HookOperation` w klasie `ParentClass` nie wykonuje żadnych zadań:

```
void ParentClass::HookOperation () { }

void DerivedClass::HookOperation () {
    // Rozszerzenie w podklasie.
}
```

IMPLEMENTACJA

Warto wspomnieć o trzech kwestiach implementacyjnych:

1. *Stosowanie kontroli dostępu w języku C++.* W języku C++ operacje proste wywoływane przez metodę szablonową można zadeklarować jako składowe chronione. Gwarantuje to, że będą one wywoływanie wyłącznie przez metodę szablonową. Operacje proste, które trzeba przesłonić, są deklarowane jako czysto wirtualne. Samej metody szablonowej nie należy przesłaniać, dlatego powinna być niewirtualną funkcją składową.
2. *Minimalizowanie liczby operacji prostych.* Ważnym celem w czasie projektowania metod szablonowych jest zminimalizowanie liczby operacji prostych, które trzeba przesłonić w podklasie w celu utworzenia ciała algorytmu. Im więcej operacji trzeba przesłonić, tym żmudniejsza będzie praca autorów kodu klienta.
3. *Konwencje nazewnicze.* Można wyróżnić przeznaczone do przesłonięcia operacje przez dodanie przedrostka do ich nazw. Na przykład w platformie MacApp [App89] (służy ona do tworzenia aplikacji na komputery Macintosh) nazwy metod szablonowych mają przedrostek „Do” — `DoCreateDocument`, `DoRead` itd.

PRZYKŁADOWY KOD

Przedstawiony tu przykład w języku C++ pokazuje, jak można wymusić w klasie nadzędnej przestrzeganie niezmienników w jej podklasach. Kod pochodzi z pakietu AppKit firmy NeXT [Add94]. Zastanówmy się nad klasą `View` obsługującą wyświetlanie elementów na ekranie. Klasa ta wymusza przestrzeganie niezmiennika, zgodnie z którym podklasy mogą wyświetlać elementy w widoku tylko wtedy, kiedy ten stanie się aktywny. Wymaga to odpowiedniego skonfigurowania stanu wyświetlania (na przykład kolorów i czcionek).

Do konfigurowania stanu możemy wykorzystać metodę szablonową `Display`. W klasie `View` zdefiniowano dwie operacje konkretne — `SetFocus` i `ResetFocus` — służące do konfigurowania i zerowania stanu wyświetlanego. Za samo wyświetlanie odpowiada operacja `DoDisplay` klasy `View` stanowiąca „punkt zaczepienia”. Metoda `Display` w celu skonfigurowania stanu wyświetlanego wywołuje najpierw operację `SetFocus`, a następnie `DoDisplay`. W dalszej kolejności metoda `Display` wywołuje operację `ResetFocus`, aby zwolnić zasoby przydzielone stanowi.

```
void View::Display () {
    SetFocus();
    DoDisplay();
    ResetFocus();
}
```

Aby zachować zgodność z niezmiennikiem, w klientach klasy `View` zawsze należy wywoływać metodę `Display`, a w podklasach klasy `View` — przesyłać operację `DoDisplay`.

W klasie `View` operacja `DoDisplay` nie wykonuje żadnych zadań:

```
void View::DoDisplay () { }
```

W podklasach operacja ta jest prześloniona, co pozwala dodać specyficzne zachowanie związane z wyświetlaniem:

```
void MyView::DoDisplay () {
    // Wyświetlanie zawartości widoku.
}
```

ZNANE ZASTOSOWANIA

Metody szablonowe są tak podstawową techniką, że można je znaleźć w niemal każdej klasie abstrakcyjnej. Wirfs-Brock i współpracownicy [WBBW90, WBJ90] przedstawiają dobre omówienie oraz analizę metod szablonowych.

POWIĄZANE WZORCE

Metody wytwarzane (s. 110) często są wywoływane przez metody szablonowe. W przykładzie z punktu „Uzasadnienie” metoda wytwarzana `DoCreateDocument` jest wywoływana przez metodę szablonową `OpenDocument`.

Strategia (s. 321): w metodach szablonowych wykorzystywane jest dziedziczenie do modyfikowania fragmentów algorytmu. W przypadku strategii za pomocą delegowania zmieniany jest cały algorytm.

OBSERWATOR (OBSERVER)

obiektowy, operacyjny

PRZEZNACZENIE

Określa zależność jeden do wielu między obiekty. Kiedy zmieni się stan jednego z obiektów, wszystkie obiekty zależne od niego są o tym automatycznie powiadamiane i aktualizowane.

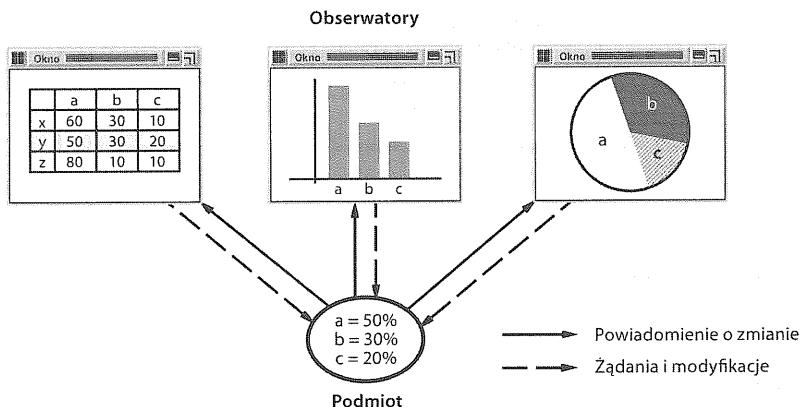
INNE NAZWY

Obiekty zależne (ang. *dependents*), publikuj-subskrybuj (ang. *publish-subscribe*).

UZASADNIENIE

Typowym efektem ubocznym podziału systemu na kolekcję współdziałających klas jest konieczność zachowania spójności między powiązanymi obiekty. Niepożądane jest jednak osiąganie tego przez tworzenie ścisłe powiązanych klas, ponieważ zmniejsza to możliwość ich powtórnego wykorzystania.

Na przykład w wielu pakietach do tworzenia graficznych interfejsów użytkownika aspekty związane z warstwą prezentacji są oddzielone od danych aplikacji [KP88, LVC89, P-88, WGM88]. Klasy definiujące dane i warstwę prezentacji aplikacji można powtórnie wykorzystać niezależnie od siebie. Mogą one też współdziałać ze sobą. Zarówno obiekt reprezentujący arkusz kalkulacyjny, jak i obiekt reprezentujący wykres słupkowy mogą w różnej formie przedstawiać informacje z tego samego obiektu danych aplikacji. Arkusz kalkulacyjny i wykres słupkowy nie wiedzą o swoim istnieniu, co umożliwia ponowne wykorzystanie tylko potrzebnego elementu. Jednak *działają* one tak, jakby były powiązane. Kiedy użytkownik zmieni informacje w arkuszu kalkulacyjnym, modyfikacje zostaną natychmiast odzwierciedlone na wykresie słupkowym (i na odwrót).



Takie działanie oznacza, że arkusz kalkulacyjny i wykres słupkowy są zależne od obiektu danych, dlatego należy je powiadomić o każdej zmianie stanu tego obiektu. Nie ma też powodu, aby ograniczać liczbę obiektów zależnych do dwóch. Te same dane można powiązać z dowolną liczbą interfejsów użytkownika.

Wzorzec Obserwator opisuje, jak nawiązać takie relacje. Kluczowe elementy tego wzorca to **podmiot** i **obserwator**. Z podmiotem można powiązać dowolną liczbę zależnych obserwatorów. Wszystkie obserwatory trzeba powiadomić o zmianie stanu podmiotu. W odpowiedzi na takie powiadomienie każdy obserwator kieruje zapytanie do podmiotu, aby zsynchronizować z nim swój stan.

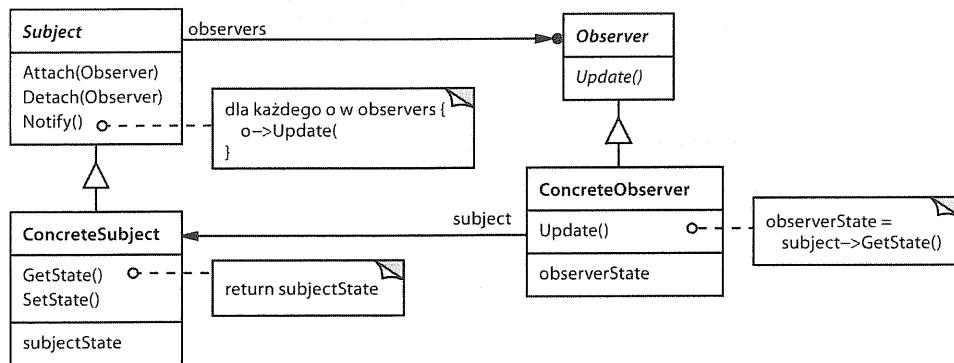
Ten typ interakcji jest też nazywany modelem **publikuj-subskrybuj**. Podmiot odpowiada za publikowanie powiadomień. Nie musi przy tym wiedzieć, kim są obserwatory. Powiadomienia może subskrybować dowolna liczba obserwatorów.

WARUNKI STOSOWANIA

Ze wzorca Obserwator można korzystać w dowolnej z poniższych sytuacji:

- ▶ Kiedy abstrakcja ma dwa aspekty, a jeden z nich zależy od drugiego. Zakapsułkowanie tych aspektów w odrębnych obiektach umożliwia modyfikowanie i wielokrotne użytkowanie ich niezależnie od siebie.
- ▶ Jeśli zmiana w jednym obiekcie wymaga zmodyfikowania drugiego, a nie wiadomo, ile obiektów trzeba przekształcić.
- ▶ Jeżeli obiekt powinien móc powiadomić inne bez określania ich rodzaju. Oznacza to, że obiekty nie powinny być ścisłe powiązane.

STRUKTURA



ELEMENTY

- ▶ **Subject**, czyli podmiot:
 - zna powiązane z nim obserwatory; podmiot może obserwować dowolna liczba obiektów **Observer**;
 - udostępnia interfejs do dołączania i odłączania obiektów **Observer**.

► **Observer:**

- definiuje interfejs do aktualizacji dla obiektów, które należy powiadamiać o zmianach podmiotu.

► **ConcreteSubject**, czyli podmiot konkretny:

- przechowuje stan istotny dla obiektów **ConcreteObserver**;
- kiedy zmieni się jego stan, wysyła powiadomienie do powiązanych z nim obserwatorów.

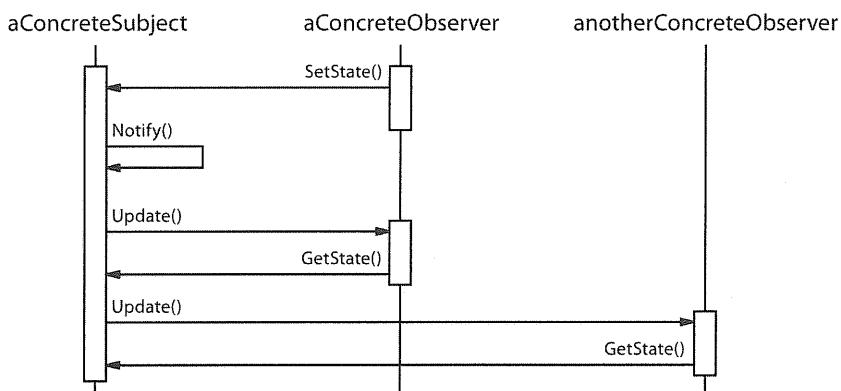
► **ConcreteObserver:**

- przechowuje referencję do obiektu **ConcreteSubject**;
- przechowuje stan, który powinien być spójny ze stanem podmiotu;
- obejmuje implementację interfejsu do aktualizacji z klasy **Observer** potrzebną do tego, aby zachować spójność stanu obiektu **ConcreteObserver** ze stanem podmiotu.

WSPÓŁDZIAŁANIE

- Obiekt **ConcreteSubject** powiadamia obserwatory o każdej zmianie, która mogłyby spowodować, że stan tego obiektu będzie niespójny ze stanem obserwatorów.
- Obiekt **ConcreteObserver** po otrzymaniu powiadomienia o zmianie obiektu **ConcreteSubject** może wysłać do niego zapytanie o informacje. Na podstawie tych danych obiekt **ConcreteObserver** dostosowuje swój stan do stanu podmiotu.

Poniższy diagram interakcji ilustruje współdziałanie między podmiotem i dwoma obserwatorami:



Warto zauważyć, że obiekt **Observer** inicjujący żądanie zmiany odracza aktualizację do momentu otrzymania powiadomienia od podmiotu. Operacja **Notify** nie zawsze jest wywoływaną przez podmiot. Może to zrobić obserwator lub zupełnie inny rodzaj obiektu. W punkcie „Implementacja” omawiamy kilka często stosowanych technik.

KONSEKWENCJE

Wzorzec Obserwator umożliwia niezależne modyfikowanie podmiotów i obserwatorów. Można powtórnie wykorzystać podmioty bez ponownego zastosowania ich obserwatorów i na odwrót. Umożliwia to dodanie obserwatorów bez modyfikowania podmiotu lub innych obserwatorów.

Do innych zalet i wad wzorca Obserwator należą między innymi:

1. *Abstrakcyjne powiązanie między obiektyami Subject i Observer.* Podmiot zna tylko listę obserwatorów, a każdy z nich jest zgodny z prostym interfejsem klasy abstrakcyjnej *Observer*. Podmiot nie zna klasy konkretnej żadnego z obserwatorów. Dlatego powiązanie między podmiotami i obserwatorami jest abstrakcyjne i minimalne.

Ponieważ klasy *Subject* i *Observer* nie są ściśle powiązane, mogą należeć do różnych warstw abstrakcji w systemie. Działający na niższym poziomie podmiot może komunikować się z funkcjonującym na wyższym poziomie obserwatorem, przez co podział systemu na warstwy zostaje zachowany. Jeśli klasy *Subject* i *Observer* połączymy ze sobą, utworzony w ten sposób obiekt będzie musiał obejmować dwie warstwy (co narusza podział na warstwy) lub działać tylko w jednej z nich (co może negatywnie wpływać na abstrakcję podziału na warstwy).

2. *Obsługa rozsyłania grupowego komunikatów.* Podmiot przy wysyłaniu powiadomień — inaczej niż w przypadku zwykłych żądań — nie musi określać odbiorcy. Powiadomienie jest automatycznie rozsyłane do wszystkich zainteresowanych obiektów, które je subskrybują. Z perspektywy podmiotu liczba takich obiektów nie ma znaczenia. Odpowiada on tylko za powiadomienie obserwatorów. Pozwala to swobodnie dodawać i usuwać obserwatory w dowolnym momencie. To one same określają, czy powinny obsłużyć powiadomienie czy je zignorować.

3. *Nieoczekiwane aktualizacje.* Ponieważ obserwatory nie wiedzą o swoim istnieniu, mogą nie znać ostatecznych kosztów zmodyfikowania podmiotu. Na pozór bezpieczna operacja na obiekcie może spowodować kaskadę aktualizacji w obserwatorach i zależnych od nich obiektach. Ponadto źle zdefiniowane lub zarządzane kryteria zależności zwykle prowadzą do błędnych aktualizacji, które trudno jest wykryć.

Problem ten jest dodatkowo powiększany przez fakt, że prosty protokół aktualizacji nie udostępnia szczegółowych informacji o tym, co zmieniło się w podmiocie. Bez dodatkowego protokołu pomagającego określić zakres modyfikacji obserwatory mogą być zmuszone do skomplikowanego ustalania tego, co się zmieniło.

IMPLEMENTACJA

W tym punkcie omawiamy kilka zagadnień związanych z implementacją mechanizmu zarządzania zależnościami.

1. *Odwzorowywanie podmiotów na obserwatory.* Najprostszy sposób na śledzenie w podmiocie obserwatorów otrzymujących powiadomienia polega na przechowywaniu referencji do nich bezpośrednio w podmiocie. Jednak to podejście może okazać się zbyt kosztowne, jeśli istnieje wiele podmiotów i nieliczne obserwatory. Jednym z rozwiązań jest zamiana kosztów pamięci na koszty w postaci czasu przetwarzania przez zastosowanie wyszukiwania

asocjacyjnego (na przykład tablic haszujących) do przechowywania odwzorowań podmiotów na obserwatory. Wtedy podmiot bez obserwatorów nie będzie powodował dodatkowych narzutów związanych z pamięcią. Jednak to podejście zwiększa koszt uzyskania dostępu do obserwatora.

2. *Obserwowanie więcej niż jednego podmiotu.* W niektórych sytuacjach uzasadnione jest uzależnienie obserwatora od więcej niż jednego podmiotu. Na przykład arkusz kalkulacyjny może zależeć od kilku źródeł danych. Konieczne jest wtedy rozszerzenie interfejsu aktualizacji, aby umożliwić obserwatorowi ustalenie, który podmiot wysłał powiadomienie. Podmiot może po prostu przekazać samego siebie jako parametr operacji *Update* i poinformować w ten sposób obserwatora o tym, który podmiot wymaga zbadania.
3. *Który obiekt uruchamia aktualizację?* Podmiot i powiązane z nim obserwatory korzystają z mechanizmu powiadomień do zachowania spójności. Jednak który obiekt powinien wywoływać operację *Notify*, aby uruchomić aktualizację? Oto dwie możliwości:
 - c) Wywoływanie operacji *Notify* w operacjach ustawiających stan podmiotu po jego zmodyfikowaniu. Zaletą tego podejścia jest to, że nie trzeba pamiętać o wywołaniu w klientach operacji *Notify* podmiotu. Wada polega na tym, że kilka kolejnych operacji spowoduje kolejne aktualizacje, co może okazać się niewygodne.
 - d) Wywoływanie operacji *Notify* w odpowiednim momencie w klientach. Zaletą jest możliwość uruchomienia aktualizacji przez klienta po wprowadzeniu kilku zmian w stanie, co pozwala uniknąć zbędnych aktualizacji pośrednich. Wadą jest dodatkowe zadanie dla klientów — uruchamianie aktualizacji. Zwiększa to prawdopodobieństwo popełnienia błędu, ponieważ autorzy klientów mogą zapomnieć o wywołaniu operacji *Notify*.
4. *Wiszące referencje do usuniętych podmiotów.* Usunięcie podmiotu nie powinno prowadzić do powstawania wiszących referencji w jego obserwatorach. Jednym ze sposobów na uniknięcie wiszących referencji jest powiadamianie przez podmiot obserwatorów o swoim usunięciu, co umożliwia wyzerowanie odwołań do niego. Zwykle samo usunięcie obserwatora nie jest dobrym rozwiązaniem, ponieważ inne obiekty mogą przechowywać referencje do niego, a sam obserwator może obserwować także inne podmioty.
5. *Upewniać się przed wysłaniem powiadomienia, że stan obiektu Subject jest wewnętrznie spójny.* Ważne jest, aby przed wywołaniem operacji *Notify* sprawdzić, czy stan obiektu *Subject* jest wewnętrznie spójny. Wynika to z tego, że obserwatory w procesie aktualizowania swojego stanu żądają od podmiotu określenia jego bieżącego stanu.

Nietrudno jest przypadkowo naruszyć regułę wewnętrznej spójności, jeśli operacje podklas klasy *Subject* wywołują odziedziczone operacje. Na przykład poniższy kod zgłasza powiadomienie, kiedy stan podmiotu jest niespójny:

```
void MySubject::Operation (int newValue) {
    BaseClassSubject::Operation(newValue);
    // Zgłaszenie powiadomienia.

    _myInstVar += newValue;
    // Aktualizowanie stanu podklasy (za późno!).
}
```

Można uniknąć tej pułapki przez wysyłanie powiadomień w metodach szablonowych (Metoda szablonowa, s. 264) klas abstrakcyjnych **Subject**. Należy zdefiniować proste operacje na potrzeby przesłaniania ich w podklasach i umieścić **Notify** jako ostatnią operację w metodzie szablonowej. Gwarantuje to, że kiedy w podklasach przesłonimy operacje klasy **Subject**, obiekt będzie miał wewnętrznie spójny stan.

```
void Text::Cut (TextRange r) {
    ReplaceRange(r); // Przedefiniowana w podklasach.
    Notify();
}
```

Przy okazji uwaga — zawsze warto udokumentować, które operacje klasy **Subject** zgłaszają powiadomienia.

6. *Unikanie specyficznych dla obserwatora protokołów aktualizacji — modele wypychania i wyciągania.* W implementacjach wzorca **Obserwator** podmiot często rozsyła dodatkowe informacje o zmianie. Podmiot przekazuje je jako argument operacji **Update**. Ilość tych informacji jest bardzo różna.

W jednym z krańcowych rozwiązań, w tak zwanym **modelu wypychania**, podmiot wysyła do obserwatorów szczegółowe informacje o zmianie niezależnie od tego, czy obserwatory tego oczekują. Przeciwieństwem tego jest **model wyciągania**, w którym podmiot wysyła tylko minimalne powiadomienie, a następnie obserwatory jawnie żądają szczegółów.

W modelu wyciągania nacisk położony jest na nieznajomość obserwatorów przez podmiot, natomiast w modelu wypychania zakładamy, że podmioty znają potrzeby powiązanych z nimi obserwatorów. Model wypychania może utrudnić wielokrotne użytkowanie obserwatorów, ponieważ w klasach **Subject** czynione są różne — nie zawsze prawdziwe — założenia na temat klas **Observer**. Z drugiej strony model wyciągania może okazać się nie-wydajny, ponieważ w klasach **Observer** trzeba bez pomocy ze strony klas **Subject** ustalić, co się zmieniło.

7. *Jawne określanie istotnych modyfikacji.* Można zwiększyć wydajność aktualizacji przez rozszerzenie interfejsu służącego do rejestracji w podmiotach i umożliwić rejestrowanie przez obserwatory zainteresowania tylko określonymi zdarzeniami. Kiedy wystąpi takie zdarzenie, podmiot poinformuje o nim wyłącznie zainteresowane nim obserwatory. Jednym ze sposobów obsługi tego mechanizmu jest wykorzystanie tak zwanych **aspektów** obiektów **Subject**. Aby zarejestrować zainteresowanie określonymi zdarzeniami, obserwatory należy połączyć z podmiotami w następujący sposób:

```
void Subject::Attach(Observer*, Aspect& interest);
```

Argument **interest** określa tu interesujące zdarzenie. W czasie powiadamiania podmiot prześle do obserwatorów zmieniony aspekt jako parametr operacji **Update**, na przykład:

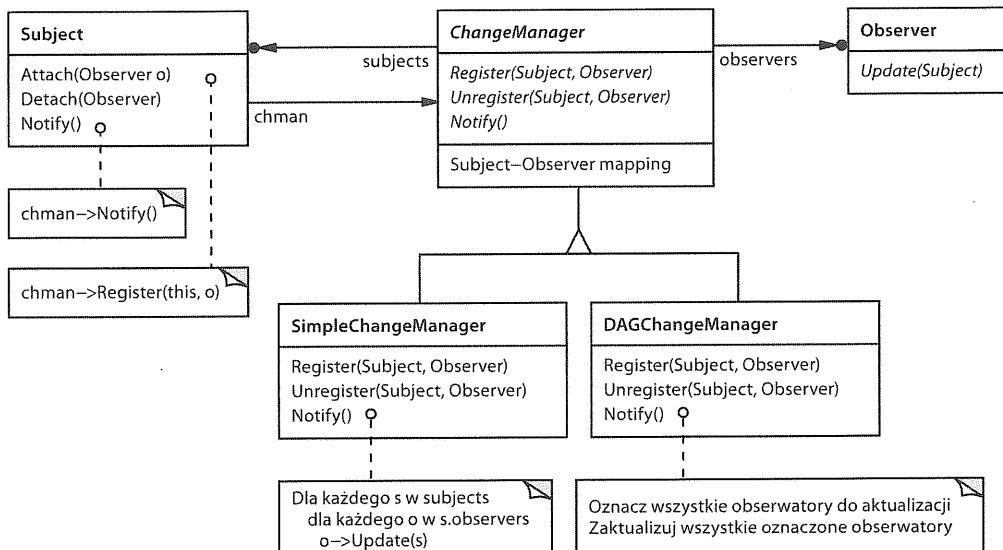
```
void Observer::Update(Subject*, Aspect& interest);
```

8. *Kapsulkowanie złożonej semantyki aktualizacji.* Jeśli relacja zależności między podmiotami i obserwatorami jest wyjątkowo złożona, czasem potrzebny jest obiekt do zarządzania takimi związkami. Nazwijmy taki obiekt **ChangeManager**. Służy on do minimalizowania pracy niezbędnej do odzwierciedlenia w obserwatorach zmian wprowadzonych w powiązanym z nimi podmiotem. Na przykład jeśli operacja modyfikuje kilka zależnych od siebie podmiotów, czasem trzeba zagwarantować, że obserwatory otrzymają powiadomienie dopiero po zmienieniu *wszystkich* podmiotów. Pozwala to uniknąć wielokrotnego powiadamiania obserwatorów.

Obiekt ChangeManager ma trzy zadania:

- Odwzorowuje podmiot na powiązane z nim obserwatory i udostępnia interfejs do zarządzania tym odwzorowaniem. Eliminuje to konieczność przechowywania referencji do obserwatorów w podmiotach i w drugą stronę.
- Definiuje określoną strategię aktualizacji.
- Na żądanie podmiotu aktualizuje wszystkie obserwatory zależne.

Poniższy diagram ilustruje prostą implementację wzorca Obserwator opartą na klasie ChangeManager. Istnieją dwie wyspecjalizowane klasy tego rodzaju. SimpleChangeManager to klasa „naiwna”, ponieważ zawsze aktualizuje wszystkie obserwatory każdego podmiotu. Z kolei klasa DAGChangeManager obsługuje acykliczne grafy skierowane reprezentujące zależności między podmiotami i powiązanymi z nimi obserwatorami. Jeśli obserwator obserwuje więcej niż jeden podmiot, lepiej jest zastosować klasę DAGChangeManager. Zastosowanie w takiej sytuacji klasy SimpleChangeManager może przy modyfikacji więcej niż jednego podmiotu spowodować zbędne aktualizacje. Klasa DAGChangeManager gwarantuje, że nastąpi tylko jedna aktualizacja obserwatora. Klasa SimpleChangeManager jest odpowiednia, jeżeli wielokrotne aktualizacje nie występują.



Klasa ChangeManager to przykład zastosowania wzorca Mediator (s. 254). Zwykle istnieje tylko jeden globalnie znany obiekt tej klasy. Wtedy użyteczny może być wzorzec Singleton (s. 130).

9. **Łączenie klas Subject i Observer.** W bibliotekach klas napisanych w językach bez obsługi wielodziedziczenia (na przykład w języku Smalltalk) zwykle nie definiuje się odrębnych klas **Subject** i **Observer**. Ich interfejsy zazwyczaj połączone są w jedną klasę. Umożliwia to zdefiniowanie bez korzystania z wielodziedziczenia obiektu, który pełni jednocześnie funkcje podmiotu i obserwatora. Na przykład w języku Smalltalk interfejsy klas **Subject** i **Observer** zdefiniowane są w klasie głównej **Object**, dlatego są dostępne we wszystkich klasach.

PRZYKŁADOWY KOD

Interfejs obserwatora jest zdefiniowany za pomocą klasy abstrakcyjnej:

```
class Subject;

class Observer {
public:
    virtual ~Observer();
    virtual void Update(Subject* theChangedSubject) = 0;
protected:
    Observer();
};
```

Ta implementacja umożliwia połączenie z każdym obserwatorem wielu podmiotów. Jeśli obserwator obserwuje więcej niż jeden podmiot, może na podstawie podmiotu przekazanego do operacji `Update` ustalić, który z nich się zmienił.

Także interfejs podmiotu jest zdefiniowany za pomocą klasy abstrakcyjnej:

```
class Subject {
public:
    virtual ~Subject();

    virtual void Attach(Observer* );
    virtual void Detach(Observer* );
    virtual void Notify();
protected:
    Subject();
private:
    List<Observer*> *_observers;
};

void Subject::Attach (Observer* o) {
    _observers->Append(o);
}

void Subject::Detach (Observer* o) {
    _observers->Remove(o);
}

void Subject::Notify () {
    ListIterator<Observer*> i(_observers);

    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Update(this);
    }
}
```

`ClockTimer` to podmiot konkretny służący do zapisywania i przechowywania godziny. Wysyła on co sekundę powiadomienie do obserwatorów. Klasa `ClockTimer` udostępnia interfejs do pobierania wartości poszczególnych jednostek czasu — godziny, minuty i sekundy.

```
class ClockTimer : public Subject {
public:
    ClockTimer();

    virtual int GetHour();
    virtual int GetMinute();
    virtual int GetSecond();

    void Tick();
};
```

Operację `Tick` wywołuje w regularnych odstępach czasu wewnętrzny zegar, co zapewnia precyzyjną podstawę pomiaru czasu. Operacja ta aktualizuje stan wewnętrzny obiektu `ClockTimer` i wywołuje operację `Notify`, aby powiadomić obserwatory o zmianie:

```
void ClockTimer::Tick () {
    // Aktualizacja wewnętrznego stanu związanego z pomiarem czasu.
    // ...
    Notify();
}
```

Teraz możemy zdefiniować klasę `DigitalClock` przeznaczoną do wyświetlania czasu. Dziedziczy ona funkcje graficzne po klasie `Widget` udostępnianej przez pakiet do tworzenia interfejsów użytkownika. Interfejs klasy `Observer` jest dołączany do interfejsu klasy `DigitalClock` przez dziedziczenie po klasie `Observer`.

```
class DigitalClock: public Widget, public Observer {
public:
    DigitalClock(ClockTimer*);
    virtual ~DigitalClock();

    virtual void Update(Subject*);
    // Przesłonięcie operacji z klasy Observer.

    virtual void Draw();
    // Przesłonięcie operacji z klasy Widget.
    // Definiowanie procesu wyświetlania cyfrowego zegara.

private:
    ClockTimer* _subject;
};

DigitalClock::DigitalClock (ClockTimer* s) {
    _subject = s;
    _subject->Attach(this);
}

DigitalClock::~DigitalClock () {
    _subject->Detach(this);
}
```

Operacja *Update* przed wyświetleniem tarczy zegara sprawdza, czy podmiot wysyłający powiadomienie to zegar:

```
void DigitalClock::Update (Subject* theChangedSubject) {
    if (theChangedSubject == _subject) {
        Draw();
    }
}

void DigitalClock::Draw () {
    // Pobieranie nowych wartości z podmiotu.

    int hour = _subject->GetHour();
    int minute = _subject->GetMinute();
    // Itd.

    // Wyświetlanie cyfrowego zegara.
}
```

Klasę *AnalogClock* można zdefiniować w taki sam sposób.

```
class AnalogClock : public Widget, public Observer {
public:
    AnalogClock(ClockTimer*);
    virtual void Update(Subject*);
    virtual void Draw();
    // ...
};
```

Poniższy kod tworzy obiekty *AnalogClock* i *DigitalClock* zawsze wyświetlające ten sam czas:

```
ClockTimer* timer = new ClockTimer;
AnalogClock* analogClock = new AnalogClock(timer);
DigitalClock* digitalClock = new DigitalClock(timer);
```

Po każdym „tyknięciu” obiektu *timer* oba zegary są zaktualizowane i wyświetlają prawidłową godzinę.

ZNANE ZASTOSOWANIA

Pierwszy i prawdopodobnie najbardziej znany przykład zastosowania wzorca *Obserwator* pojawia się w Smalltalk Model/View/Controller (MVC) — platformie do tworzenia interfejsów użytkownika działających w środowisku języka Smalltalk [KP88]. Klasa *Model* z MVC pełni funkcję podmiotu, natomiast *View* to klasa bazowa obserwatorów. Smalltalk, platforma ET++ [WGM88] i biblioteka klas THINK [Sym93b] zapewniają ogólny mechanizm zarządzania zależnościami przez udostępnienie interfejsów podmiotu oraz obserwatora w klasie nadzędnej wszystkich pozostałych klas systemu.

Inne pakiety do tworzenia interfejsów użytkownika, w których zastosowano ten wzorzec, to między innymi InterViews [LVC89], The Andrew Toolkit [P-88] i Unidraw [VL90]. W InterViews klasy *Observer* i *Observable* (klasa podmiotów) są zdefiniowane jawnie. W pakiecie Andrew ich nazwy to „view” i „data object”. W Unidraw obiekty edytora graficznego są podzielone na rodziny *View* (klasa obserwatorów) i *Subject*.

POWIĄZANE WZORCE

Mediator (s. 254): z uwagi na kapsułkowanie złożonej semantyki aktualizacji klasa ChangeManager działa jak mediator między podmiotami i obserwatorami.

Singleton (s. 130): do utworzenia klasy ChangeManager można zastosować wzorzec Singleton, aby jej egzemplarz był niepowtarzalny i globalnie dostępny.

ODWIEDZAJĄCY (VISITOR)

obiektowy, operacyjny

PRZEZNACZENIE

Reprezentuje operację wykonywaną na elementach struktury obiektów. Wzorzec ten umożliwia zdefiniowanie nowej operacji bez zmieniania klas elementów, na których działa.

UZASADNIENIE

Zastanówmy się nad kompilatorem reprezentującym programy jako drzewa składni abstrakcyjnej. Musi on wykonywać operacje na takich drzewach w czasie analizowania „semantyki statycznej”, na przykład przy sprawdzaniu, czy wszystkie zmienne są zdefiniowane. Kompilator musi też generować kod. Dlatego trzeba zdefiniować w nim operacje do sprawdzania typów, optymalizowania kodu, przeprowadzania analiz przepływu, określania, czy do zmiennych przed ich użyciem przypisano wartości itd. Ponadto drzewa składni abstrakcyjnej można wykorzystać do eleganckiego wyświetlanego kodu, zmieniania struktury programu, instrumentacji kodu i obliczania różnych wskaźników dotyczących programu.

W większości tych operacji węzły reprezentujące instrukcje przypisania należy traktować inaczej niż węzły przedstawiające zmienne lub wyrażenia arytmetyczne. Dlatego potrzebna jest jedna klasa dla instrukcji przypisania, inna do dostępu do zmiennych, jeszcze następna dla wyrażeń arytmetycznych itd. Zestaw klas reprezentujących węzły zależy oczywiście od kompilowanego języka, jednak jest stosunkowo stały w ramach każdego z nich.

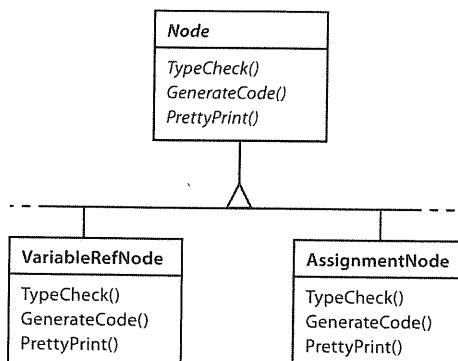
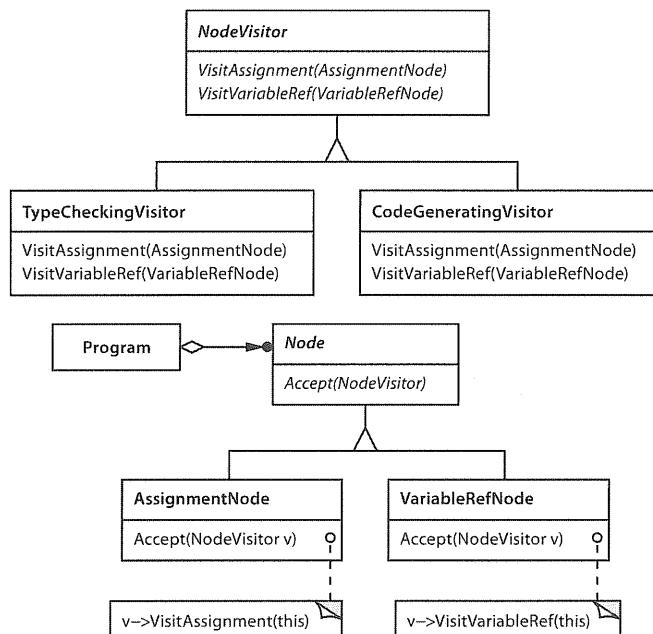


Diagram ilustruje część hierarchii klasy Node. Problem polega tu na tym, że rozproszenie wszystkich operacji po różnych klasach węzłów prowadzi do powstania systemu trudnego do zrozumienia, konserwowania i modyfikowania. Łączenie kodu do sprawdzania typów z kodem do obsługi eleganckiego wyświetlanego lub analizy przepływu jest nieintuicyjne. Ponadto przy tym podejściu dodanie nowej operacji wymaga zwykle ponownego skompilowania wszystkich klas. Lepiej jest zapewnić możliwość dodawania każdej operacji osobno i zachować niezależność klas węzłów od stosowanych do nich operacji.

Oba wspomniane cele można osiągnąć przez umieszczenie powiązanych ze sobą operacji z każdej klasy w odrębnym obiekcie, tak zwanym **odwiedzającym**, i przekazywanie do niego elementów drzewa składni abstrakcyjnej w czasie przechodzenia po tym drzewie. Jeśli element „przyjmuje” odwiedzającego, wysyła do niego żądanie z zakodowaną klasą. Żądanie obejmuje też argument w postaci samego elementu. Odwiedzający wykonuje następnie dla danego elementu operację, która wcześniej znajdowała się w klasie tego elementu.

Na przykład kompilator bez odwiedzających może sprawdzać typy w procedurze przez wywołanie operacji TypeCheck na drzewie składni abstrakcyjnej. W każdym węźle operacja TypeCheck jest zaimplementowana w postaci wywołania operacji TypeCheck jego komponentów (zobacz wcześniejszy diagram klas). Jeśli kompilator sprawdza typy w procedurze za pomocą odwiedzających, może utworzyć obiekt TypeCheckingVisitor i wywołać na drzewie składni abstrakcyjnej operację Accept ze wspomnianym obiektem jako argumentem. W każdym z węzłów operacja Accept jest zaimplementowana w formie wywołania zwrótnego do odwiedzającego. Węzeł reprezentujący przypisanie wywołuje na odwiedzającym operację VisitAssignment, natomiast węzeł referencji do zmiennej wywołuje operację VisitVariableReference. Wcześniejsa operacja TypeCheck klasy AssignmentNode staje się operacją VisitAssignment klasy TypeCheckingVisitor.

Aby wykorzystać odwiedzających nie tylko do sprawdzania typów, musimy utworzyć nadrzędna klasę abstrakcyjną NodeVisitor dla wszystkich odwiedzających powiązanych z drzewem składni abstrakcyjnej. W klasie NodeVisitor trzeba zadeklarować operacje dla wszystkich klas węzłów. Aplikacja, która ma obliczać miary dotyczące programu, wymaga zdefiniowania nowych podklas klasy NodeVisitor, natomiast nie trzeba wtedy dodawać kodu specyficznego dla aplikacji do klas węzłów. Wzorzec Odwiedzający ilustruje, jak zakapsułkować operacje dotyczące każdego etapu komplikacji w klasach Visitor powiązanych z poszczególnymi fazami.



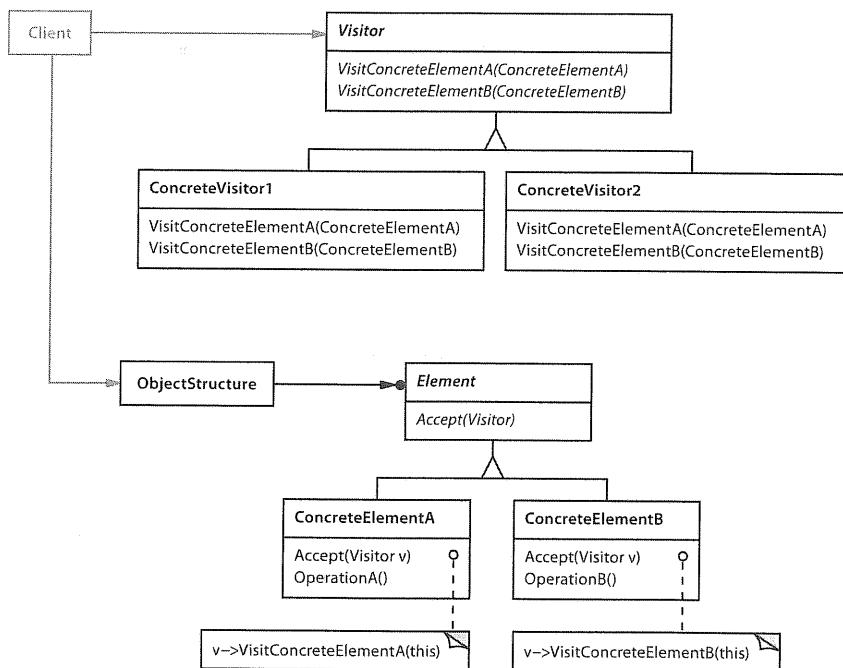
Przy stosowaniu wzorca Odwiedzający należy zdefiniować dwie hierarchie klas — jedną dla przetwarzanych elementów (hierarchia klas *Node*) i drugą dla odwiedzających z definicjami operacji wykonywanych na tych elementach (hierarchia klas *NodeVisitor*). Aby utworzyć nową operację, należy dodać nową podklasę do hierarchii klas odwiedzających. Dopóki gramatyka akceptowana przez kompilator się nie zmienia (czyli nie powstają nowe podklasy klasy *Node*), możemy dodawać nowe funkcje przez definiowanie nowych podklas klasy *NodeVisitor*.

WARUNKI STOSOWANIA

Wzorzec Odwiedzający należy stosować w następujących warunkach:

- ▶ Jeśli struktura obiektów obejmuje wiele klas o różnych interfejsach, a chcesz wykonywać na tych obiektach operacje zależne od ich klas konkretnych.
- ▶ Kiedy na obiektach z ich struktury trzeba wykonywać wiele różnych i niepowiązanych operacji, a chcesz uniknąć „zaśmiecania” klas tymi operacjami. Wzorzec Odwiedzający umożliwia przechowywanie powiązanych operacji razem przez zdefiniowanie ich w jednej klasie. Jeżeli struktura obiektów jest współużytkowana przez wiele aplikacji, należy zastosować wzorzec Odwiedzający do umieszczenia operacji w tylko tych programach, w których są potrzebne.
- ▶ Gdy klasy definiujące strukturę obiektów rzadko się zmieniają, ale często chcesz definiować nowe operacje dla tej struktury. Zmodyfikowanie klas ze struktury obiektów wymaga przeddefiniowania interfejsu wszystkich odwiedzających, co może okazać się kosztowne. Jeśli klasy ze struktury obiektów zmieniają się często, prawdopodobnie lepiej będzie zdefiniować operacje w tych klasach.

STRUKTURA



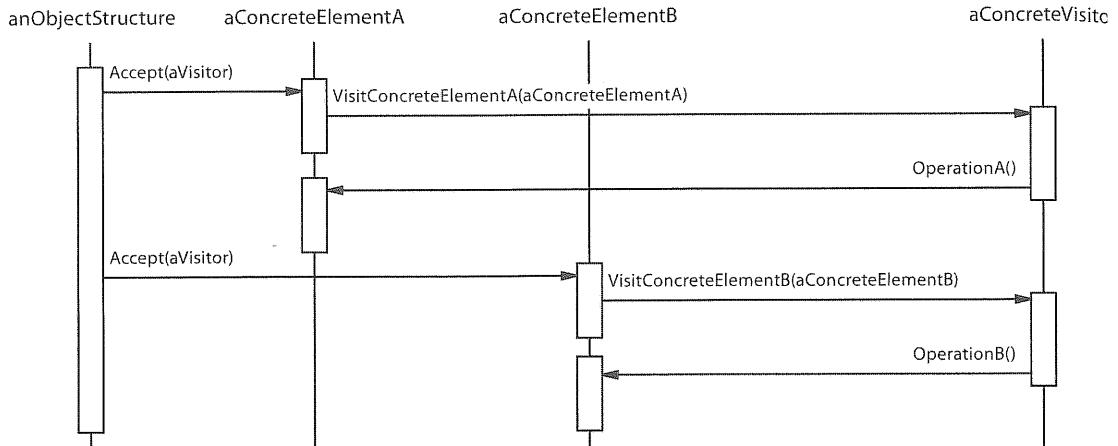
ELEMENTY

- ▶ **Visitor** (`NodeVisitor`), czyli odwiedzający:
 - obejmuje deklarację operacji `Visit` dla każdej klasy `ConcreteElement` ze struktury obiektów. Nazwa i sygnatura operacji określają klasę wysyłającą do odwiedzającego żądanie `Visit`. Umożliwia to odwiedzającym ustalenie klas konkretnego odwiedzanego elementu. Następnie odwiedzający może bezpośrednio uzyskać dostęp do elementu poprzez jego interfejs.
- ▶ **ConcreteVisitor** (`TypeCheckingVisitor`), czyli odwiedzający konkretny:
 - obejmuje implementację wszystkich operacji zadeklarowanych w klasie `Visitor`. Każda operacja realizuje fragment algorytmu zdefiniowanego na potrzeby odpowiedniej klasy obiektu ze struktury. Klasa `ConcreteVisitor` udostępnia kontekst działania algorytmu i przechowuje jego lokalny stan. W stanie często akumulowane są wyniki w czasie przechodzenia po danej strukturze.
- ▶ **Element** (`Node`):
 - definiuje operację `Accept` przyjmującą odwiedzającego jako argument.
- ▶ **ConcreteElement** (`AssignmentNode`, `VariableRefNode`):
 - obejmuje implementację operacji `Accept` przyjmującej odwiedzającego jako argument.
- ▶ **ObjectStructure** (`Program`):
 - może wyliczać zapisane w niej elementy;
 - może udostępniać wysokopoziomowy interfejs, który umożliwia odwiedzającym odwiedzenie elementów struktury;
 - może być albo kompozytem (zobacz wzorzec Kompozyt, s. 170), albo kolekcją, taką jak lista lub zbiór.

WSPÓŁDZIAŁANIE

- ▶ Klient korzystający ze wzorca Odwiedzający musi utworzyć obiekt `ConcreteVisitor`, a następnie przejść po strukturze obiektów i odwiedzić każdy element za pomocą odwiedzającego.
- ▶ Element w czasie odwiedzin wywołuje operację `Visitor` odpowiednią dla jego klasy. Element przekazuje sam siebie jako argument tej operacji, aby umożliwić odwiedzającemu dostęp do stanu, jeśli jest to potrzebne.

Poniższy diagram interakcji ilustruje współdziałanie między strukturą obiektów, odwiedzającym i dwoma elementami:



KONSEKWENCJE

Oto niektóre zalety i wady stosowania wzorca Odwiedzający:

- Ułatwia dodawanie nowych operacji.* Wzorzec ten sprawia, że można łatwo dodawać operacje zależne od komponentów złożonych obiektów. Można zdefiniować nową operację dla struktury obiektów przez proste dodanie nowego odwiedzającego. Natomiast w przypadku rozproszenia funkcji po wielu klasach trzeba zmodyfikować każdą z nich, aby zdefiniować nową operację.
- Umożliwia połączenia powiązanych operacji i wyodrębnienia niepowiązanych.* Powiązane zachowania nie są w omawianym wzorcu umieszczane w klasach definiujących strukturę obiektów. Zamiast tego znajdują się w odwiedzającym. Niepowiązane zestawy zachowań są wyodrębniane do osobnych podklas klasy `Visitor`. Upraszczają to zarówno klasy z definicjami elementów, jak i algorytmy zdefiniowane w odwiedzających. W odwiedzającym można ukryć dowolne specyficzne dla algorytmu struktury danych.
- Utrudnia dodawanie nowych klas `ConcreteElement`.* Wzorzec Odwiedzający sprawia, że trudno jest dodawać nowe podklasy klasy `Element`. Każda nowa klasa `ConcreteElement` powoduje powstanie nowej operacji abstrakcyjnej w klasie `Visitor` i powiązanej implementacji w każdej klasie `ConcreteVisitor`. Czasem w klasie `Visitor` można udostępnić implementację domyślną dziedziczoną przez większość klas `ConcreteVisitor`, jednak jest to raczej wyjątek od reguły.

Dlatego w czasie stosowania wzorca Odwiedzający trzeba koniecznie rozważyć, czy większe jest prawdopodobieństwo zmiany algorytmu stosowanego w strukturze obiektów czy klas obiektów składających się na tę strukturę. Hierarchia klas `Visitor` może być trudna w konserwacji, jeśli nowe klasy `ConcreteElement` są dodawane często. Wtedy prawdopodobnie łatwiej jest po prostu zdefiniować operacje w klasach składających się na strukturę. Jeśli hierarchia klas `Element` jest stabilna, jednak nieustannie dodajesz operacje lub zmieniasz algorytmy, wtedy wzorzec Odwiedzający pomoże w zarządzaniu zmianami.

4. Umożliwia odwiedzanie różnych hierarchii klas. Iterator (zobacz wzorzec Iterator, s. 230) pozwala odwiedzać obiekty w strukturze w czasie przechodzenia po nich przez wywoływanie ich operacji. Jednak iterator nie może działać dla różnych struktur obiektów z elementami innego typu. Na przykład interfejs klasy `Iterator` zdefiniowany na stronie 263 może uzyskać dostęp tylko do obiektów typu `Item`:

```
template <class Item>
class Iterator {
    // ...
    Item CurrentItem() const;
};
```

Na tej podstawie można wywnioskować, że wszystkie elementy, które iterator może odwiedzić, mają wspólną klasę nadziedziczoną `Item`.

Odwiedzający nie ma takich ograniczeń. Może odwiedzać obiekty, które nie mają wspólnej klasy nadziedziczonej. Do interfejsu klasy `Visitor` można dodać obiekt dowolnego typu. Oto przykład:

```
class Visitor {
public:
    // ...
    void VisitMyType(MyType* );
    void VisitYourType(YourType* );
};
```

Klasy `MyType` i `YourType` w ogóle nie muszą być powiązane relacją dziedziczenia.

5. Pozwala akumulować stan. Odwiedzający mogą akumulować stan w czasie odwiedzania każdego elementu w strukturze obiektów. Bez odwiedzającego ten stan byłby przekazywany jako dodatkowe argumenty do operacji odpowiedzialnych za przechodzenie lub mógłby mieć postać zmiennych globalnych.
6. Powoduje naruszenie kapsułkowania. Korzystanie ze wzorca Odwiedzający oparte jest na założeniu, że interfejs klasy `ConcreteElement` jest wystarczająco rozbudowany, aby odwiedzający mógł wykonywać swoje zadania. Powoduje to, że wzorzec ten często wymusza udostępnianie operacji publicznych mających dostęp do wewnętrznego stanu elementu, co może naruszać kapsułkowanie.

IMPLEMENTACJA

Dla każdej struktury obiektów należy utworzyć klasę `Visitor`. W tej klasie abstrakcyjnej zadeklarowane są operacje `VisitConcreteElement` dla wszystkich klas `ConcreteElement` definiujących strukturę obiektów. Każda operacja `Visit` w klasie `Visitor` ma argument określonego typu `ConcreteElement`, co umożliwia klasie `Visitor` bezpośredni dostęp do interfejsu danego typu. W klasach `ConcreteVisitor` każda operacja `Visit` jest przesłonięta i obejmuje implementacją specyficznego dla odwiedzającego zachowania dla odpowiedniej klasie `ConcreteElement`.

W języku C++ klasę Visitor można zadeklarować w następujący sposób:

```
class Visitor {
public:
    virtual void VisitElementA(ElementA* );
    virtual void VisitElementB(ElementB* );

    // Tak dalej dla innych elementów konkretnych.
protected:
    Visitor();
};
```

W każdej klasie ConcreteElement zaimplementowana jest operacja Accept wywołująca odpowiednią operację Visit... odwiedzającego dla danej klasy ConcreteElement. W ten sposób wywoływana ostatecznie operacja zależy zarówno od klasy elementu, jak i od klasy odwiedzającego⁷.

Elementy konkretne są zadeklarowane w następujący sposób:

```
class Element {
public:
    virtual ~Element();
    virtual void Accept(Visitor& ) = 0;
protected:
    Element();
};

class ElementA : public Element {
public:
    ElementA();
    virtual void Accept(Visitor& v) { v.VisitElementA(this); }
};

class Elements : public Element {
public:
    ElementB();
    virtual void Accept(Visitors v) { v.VisitElementB(this); }
};
```

W klasie CompositeElement operacja Accept może być zaimplementowana tak:

```
class CompositeElement : public Element {
public:
    virtual void Accept(Visitor& );
private:
    List<Element.*>* _children;
};
```

⁷ Aby nadać tym operacjom tę samą prostą nazwę, na przykład Visit, moglibyśmy zastosować przeciążanie funkcji, ponieważ operacje różnią się już ze względu na otrzymywany parametr. Takie przeciążanie ma zalety i wady. Z jednej strony podkreśla fakt, że każda operacja obejmuje te same analizy, choć na innym argumencie. Z drugiej strony może to utrudniać osobie czytającej kod zrozumienie, co dzieje się w miejscu wywołania. Tak naprawdę sprawdza się to do tego, czy programista uważa przeciążanie funkcji za korzystne.

```

void CompositeElement::Accept (Visitor& v) {
    ListIterator<Element*> i(_children);

    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Accept(v);
    }
    v.VisitCompositeElement(this);
}

```

Oto dwa zagadnienia implementacyjne, które mają znaczenie przy stosowaniu wzorca Odwiedzający:

1. *Dwukrotna dyspozycja* (ang. *double dispatch*). Wzorzec Odwiedzający umożliwia dodawanie operacji do klas bez modyfikowania tych ostatnich. W tym wzorcu uzyskano ten efekt przez zastosowanie techniki **dwukrotnej dyspozycji**. Jest to dobrze znana metoda. Niektóre języki programowania, na przykład CLOS, obsługują ją bezpośrednio. W innych językach, takich jak C++ i Smalltalk, stosowana jest **dyspozycja jednokrotna**.

W językach z dyspozycją jednokrotną dwa kryteria określają, która operacja posłuży do realizacji żądania. Są to: nazwa żądania i typ odbiorcy. Na przykład to, która operacja zostanie wywołana w wyniku zgłoszenia żądania `GenerateCode`, zależy od typu wybranego obiektu węzła. W języku C++ wywołanie żądania `GenerateCode` na egzemplarzu klasy `VariableRefNode` spowoduje wywołanie operacji `VariableRefNode::GenerateCode` (generuje ona kod dla referencji do zmiennej). Wywołanie żądania `GenerateCode` na obiekcie `AssignmentNode` doprowadzi do wywołania operacji `AssignmentNode::GenerateCode` (generuje ona kod dla przypisań). Wywoływana operacja zależy zarówno od rodzaju żądania, jak i od typu odbiorcy.

Przy dyspozycji dwukrotnej wykonywana operacja zależy od rodzaju żądania i typów dwóch odbiorców. `Accept` to operacja z dyspozycją dwukrotną. Jej znaczenie zależy od dwóch typów — odwiedzającego i elementu. Dwukrotna dyspozycja umożliwia odwiedzającym żądanie różnych operacji dla poszczególnych klas elementów⁸.

Oto kluczowy aspekt wzorca Odwiedzający — wykonywana operacja zależy zarówno od typu odwiedzającego, jak i od typu odwiedzanego elementu. Zamiast statycznie wiązać operacje w interfejsie elementu, można umieścić je w odwiedzającym i wykorzystać operację `Accept` do przeprowadzenia wiązania w czasie wykonywania programu. Rozszerzanie interfejsu klasy `Element` sprowadza się wtedy do zdefiniowania jednej nowej podklasy klasy `Visitor` zamiast wielu nowych podklas klasy `Element`.

2. Która jednostka odpowiada za poruszanie się po strukturze obiektów? Odwiedzający musi odwiedzić każdy element struktury obiektów. Jak jednak ma do nich dotrzeć? Obsługę przezechodzenia po strukturze można umieścić w dowolnym z trzech miejsc: w strukturze obiektów, w odwiedzającym lub w odrębnym obiekcie iteratora (zobacz wzorzec Iterator, s. 230).

⁸ Skoro można stosować dyspozycję dwukrotną, dlaczego nie wprowadzić dyspozycji trzykrotniej, czterokrotniej lub dla dowolnej innej liczby? W rzeczywistości dyspozycja dwukrotna jest specjalnym przypadkiem **dyspozycji wielokrotnej**, co polega na wyborze operacji na podstawie dowolnej liczby typów (technikę tę obsługuje język CLOS). W językach z obsługą dyspozycji dwukrotnej lub wielokrotnej wzorzec Odwiedzający jest potrzebny w mniejszym stopniu.

Za iterowanie często odpowiada struktura obiektów. Kolekcje po prostu przechodzą po swoich elementach i wywołują dla każdego z nich operację *Accept*. Przy poruszaniu się po kompozycie zwykle każda operacja *Accept* przechodzi po elementach podrzędnych kompozytu i rekurencyjnie wywołuje tę samą operację dla każdego z nich.

Inna możliwość to zastosowanie do odwiedzenia elementów iteratora. W języku C++ może to być iterator wewnętrzny lub zewnętrzny (zależy to od dostępnych elementów i wydajności). W języku Smalltalk przeważnie stosuje się iterator wewnętrzny za pomocą operacji do: i bloku. Ponieważ iteratory wewnętrzne są zaimplementowane w strukturze obiektów, ich użycie przypomina wykorzystanie takiej struktury do iterowania. Główna różnica między tymi podejściami polega na tym, że iterator wewnętrzny nie prowadzi do dwukrotnej dyspozycji, ale wywołuje operację na *odwiedzającym* z *elementem* jako argumentem, a nie na odwrót. Można jednak łatwo zastosować wzorzec Odwiedzający za pomocą iteratora wewnętrznego, jeśli operacja odwiedzającego wywołuje operację elementu bez rekurencji.

Algorytm przechodzenia po strukturze można umieścić nawet w odwiedzającym, choć powoduje to powielenie kodu tego algorytmu w każdej klasie *ConcreteVisitor* dla każdej zagregowanej klasy *ConcreteElement*. Głównym powodem umieszczenia strategii przechodzenia w odwiedzającym jest implementowanie wyjątkowo złożonego procesu poruszania się, zależnego od skutków uruchomienia operacji dla struktury obiektów. Taką sytuację opisujemy w punkcie „Przykładowy kod”.

PRZYKŁADOWY KOD

Ponieważ odwiedzający jest zwykle powiązany z kompozytem, do zilustrowania wzorca Odwiedzający wykorzystamy klasy *Equipment* zdefiniowane w punkcie „Przykładowy kod” w opisie wzorca Kompozyt (s. 170). W klasie *Visitor* zdefiniujemy operacje do generowania wykazu elementów i obliczania łącznego kosztu sprzętu. Klasy *Equipment* są tak proste, że stosowanie wzorca Odwiedzający nie jest konieczne, jednak pozwalają dobrze zilustrować mechanizmy związane z implementowaniem tego wzorca.

Jeszcze raz przedstawiamy tu klasę *Equipment* z opisu wzorca Kompozyt (s. 170). Wzbogaciliśmy ją o operację *Accept*, aby umożliwić współdziałanie tej klasy z odwiedzającym:

```
class Equipment {
public:
    virtual ~Equipment();

    const char* Name() { return _name; }

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();

    virtual void Accept(EquipmentVisitor&);

protected:
    Equipment(const char*);
```

private:

```
    const char* _name;
};
```

Operacje klasy `Equipment` zwracają cechy produktów, na przykład poziom zużycia prądu i cenę. W podklasach operacje te są odpowiednio przeddefiniowane pod kątem specyficznych rodzajów sprzętu (takich jak płyty montażowe, dyski i płyty główne).

Jak przedstawia to następny fragment, klasa abstrakcyjna dla wszystkich odwiedzających powiązanych ze sprzętem obejmuje funkcję wirtualną dla każdej podklasy klasy `Equipment`. Domyślnie funkcje te nie wykonują żadnych zadań.

```
class EquipmentVisitor {
public:
    virtual ~EquipmentVisitor();

    virtual void VisitFloppyDisk(FloppyDisk*);
    virtual void VisitCard(Card*);
    virtual void VisitChassis(Chassis*);
    virtual void VisitBus(Bus*);

    // I tak dalej dla pozostałych podklas konkretnych klasy Equipment.
protected:
    EquipmentVisitor();
};
```

W podklasach klasy `Equipment` operacja `Accept` jest zdefiniowana w zasadzie w ten sam sposób — wywołuje operację `EquipmentVisitor` odpowiadającą klasie, która otrzymała żądanie `Accept`, na przykład:

```
void FloppyDisk::Accept (EquipmentVisitor& visitor) {
    visitor.VisitFloppyDisk(this);
}
```

W klasach sprzętu składającego się z innych urządzeń (w szczególności w podklasach klasy `CompositeEquipment` we wzorcu Kompozyt) operacja `Accept` przechodzi po elementach podrzędnych i wywołuje operację `Accept` dla każdego z nich. Następnie wywołuje operację `Visit` w standardowy sposób. Na przykład operacja `Chassis::Accept` może przechodzić po wszystkich częściach płyty montażowej w następujący sposób:

```
void Chassis::Accept (EquipmentVisitor& visitor) {
    for (
        ListIterator<Equipment*> i(_parts);
        !i.IsDone();
        i.Next()
    ) {
        i.CurrentItem()->Accept(visitor);
    }
    visitor.VisitChassis(this);
}
```

Podklasy klasy `EquipmentVisitor` definiują określone algorytmy dla struktury reprezentującej sprzęt. Klasa `PricingVisitor` oblicza cenę na podstawie takiej struktury. Uwzględnia cenę netto wszystkich prostych urządzeń (na przykład stacji dyskietek) i cenę z rabatem sprzętu złożonego (na przykład płyt montażowych i magistral).

```

class PricingVisitor : public EquipmentVisitor {
public:
    PricingVisitor();

    Currency& GetTotalPrice();

    virtual void VisitFloppyDisk(FloppyDisk* );
    virtual void VisitCard(Card* );
    virtual void VisitChassis(Chassis* );
    virtual void VisitBus(Bus* );
    // ...
private:
    Currency _total;
};

void PricingVisitor::VisitFloppyDisk (FloppyDisk* e) {
    _total += e->NetPrice();
}

void PricingVisitor::VisitChassis (Chassis* e) {
    _total += e->DiscountPrice();
}

```

Klasa PricingVisitor oblicza łączną cenę na podstawie wszystkich węzłów ze struktury reprezentującej sprzęt. Warto zauważyć, że klasa ta wybiera właściwą regułę ustalania ceny dla klas sprzętu przez wywołanie odpowiedniej funkcji składowej. Co więcej, możemy zmodyfikować taką regułę dla struktury reprezentującej sprzęt przez zmianę klasy PricingVisitor.

Odwiedzającego do tworzenia wykazu urządzeń możemy zdefiniować w następujący sposób:

```

class InventoryVisitor : public EquipmentVisitor {
public:
    InventoryVisitor();

    Inventory& GetInventory();

    virtual void VisitFloppyDisk(FloppyDisk* );
    virtual void VisitCard(Card* );
    virtual void VisitChassis(Chassis* );
    virtual void VisitBus(Bus* );
    // ...

private:
    Inventory _inventory;
};

```

Klasa InventoryVisitor sumuje łączną cenę dla każdego rodzaju sprzętu w strukturze obiektów. W klasie InventoryVisitor wykorzystano klasę Inventory definiującą interfejs do dodawania sprzętu (nie przedstawiamy jej w tym miejscu).

```

void InventoryVisitor::VisitFloppyDisk (FloppyDisk* e) {
    _inventory.Accumulate(e);
}

```

```
void InventoryVisitor::VisitChassis (Chassis* e) {
    _inventory.Accumulate(e);
}
```

Klasę InventoryVisitor możemy zastosować do struktury reprezentującej sprzęt w następujący sposób:

```
Equipment* component;
InventoryVisitor visitor;

component->Accept(visitor);
cout << "Wykaz "
    << component->Name()
    << visitor.GetInventory();
```

Teraz pokażemy, jak za pomocą wzorca Odwiedzający zaimplementować przykład dotyczący języka Smalltalk ze wzorca Interpreter (zobacz stronę 248). Ten przykład, podobnie jak poprzedni, jest tak prosty, że wzorzec Odwiedzający nie przyniesie tu istotnych korzyści. Możemy jednak dobrze przedstawić sposób stosowania tego wzorca. Ponadto przykład ten ilustruje sytuację, w której przechodzenie po strukturze jest zadaniem odwiedzającego.

Struktura obiektów (wyrażenia regularne) składa się z czterech klas. Wszystkie one mają metodę accept: przyjmującą odwiedzającego jako argument. W klasie SequenceExpression metoda accept: wygląda tak:

```
accept: aVisitor
^ aVisitor visitSequence: self
```

W klasie RepeatExpression metoda accept: wysyła komunikat visitRepeat:. W klasie AlternationExpression jest to komunikat visitAlternation:, a w klasie LiteralExpression — komunikat visitLiteral:.

Cztery wymienione klasy muszą ponadto mieć funkcje dostępu przeznaczone do użytku przez odwiedzającego. W klasie SequenceExpression są to funkcje expression1 i expression2; w klasie AlternationExpression — funkcje alternative1 i alternative2; w klasie RepeatExpression — funkcja repetition; a w klasie LiteralExpression — components.

Odpowiednikiem klasy ConcreteVisitor jest tu klasa REMatchingVisitor. Odpowiada ona za przechodzenie po strukturze, ponieważ używany do tego algorytm działa zależnie od warunków. Związane jest to przede wszystkim z tym, że klasa RepeatExpression wielokrotnie przechodzi po komponencie. Klasa REMatchingVisitor obejmuje zmienną egzemplarza inputState. Metody w tej klasie są w zasadzie identyczne z metodami match: klas wyrażeń z opisu wzorca Interpreter. Różnica polega na tym, że argument o nazwie inputState zastąpiono tu dopasowywanym węzłem wyrażenia. Jednak metody te nadal zwracają zbiór strumieni dopasowywany w wyrażeniu w celu określenia jego bieżącego stanu.

```
visitSequence: sequenceExp
    inputState := sequenceExp expression1 accept: self.
    ^ sequenceExp expression2 accept: self.

visitRepeat: repeatExp
    | finalState |
```

```

finalState := inputState copy.
[inputState isEmpty]
whileFalse:
    [inputState := repeatExp repetition accept: self.
     finalState addAll: inputState].
^ finalState

visitAlternation: alternateExp
| finalState originalState |
originalState := inputState.
finalState := alternateExp alternative1 accept: self.
inputState := originalState.
finalState addAll: (alternateExp alternative2 accept: self).
^ finalState

visitLiteral: literalExp
| finalState tStream |
finalState := Set new.
inputState
do:
    [:stream | tStream := stream copy.
     (tStream nextAvailable:
      literalExp components size
     ) = literalExp components
      ifTrue: [finalState add: tStream]
    ] .
^ finalState

```

ZNANE ZASTOSOWANIA

W kompilatorze języka Smalltalk-80 znajduje się odpowiednik klasy *Visitor* — klasa *Program Enumerator*. Jest ona używana głównie w algorytmach analizujących kod źródłowy. Nie służy (choć mogłyby) do generowania kodu lub jego eleganckiego wyświetlania.

IRIS Inventor [Str93] to pakiet narzędziowy do tworzenia aplikacji z obszaru grafiki trójwymiarowej. W pakiecie tym trójwymiarowa scena jest przedstawiana jako hierarchia węzłów, z których każdy reprezentuje albo obiekt geometryczny, albo jego atrybut. Operacje, takie jak wyświetlanie sceny lub odwzorowywanie zdarzeń wejściowych, wymagają zastosowania różnych sposobów przechodzenia przez tę hierarchię. W pakiecie Inventor służą do tego odwiedzający nazywani akcjami. Istnieją różne rodzaje odwiedzających przeznaczone do wyświetlania, obsługi zdarzeń, wyszukiwania, wypełniania i określania ramek ograniczających.

Aby ułatwić dodawanie nowych węzłów, w pakiecie Inventor zaimplementowano system dwukrotnej dyspozycji dla języka C++. System ten jest oparty na informacjach o typie z czasu wykonywania programu i dwuwymiarowej tablicy, w której wiersze reprezentują odwiedzających, a kolumny — klasy węzłów. Komórki przechowują wskaźnik do funkcji powiązanej z odwiedzającym i klasą węzła.

Mark Linton wymyślił pojęcie „*Visitor*” (czyli odwiedzający) i zastosował je w specyfikacji pakietu Fresco Application Toolkit firmy X Consortium [LP93].

POWIĄZANE WZORCE

Kompozyt (s. 170): odwiedzających można wykorzystać do zastosowania operacji dla struktury obiektów zdefiniowanej za pomocą wzorca Kompozyt.

Interpreter (s. 217): odwiedzającego można zastosować do przeprowadzenia interpretacji.

PAMIĄTKA (MEMENTO)

obiektowy, operacyjny

PRZEZNACZENIE

Bez naruszania kapsułkowania rejestruje i zapisuje w zewnętrznej jednostce wewnętrzny stan obiektu, co umożliwia późniejsze przywrócenie obiektu według zapamiętanego stanu.

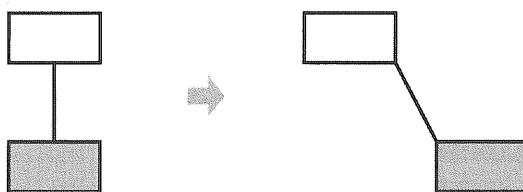
INNA NAZWA

Znacznik (ang. *token*).

UZASADNIENIE

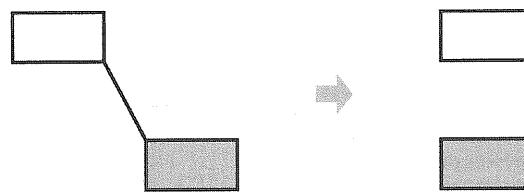
Czasem trzeba zarejestrować wewnętrzny stan obiektu. Jest to niezbędne przy implementowaniu punktów kontrolnych i mechanizmów cofania umożliwiających użytkownikom anulowanie operacji wykonanych na próbę lub przywrócenie stanu programu po wystąpieniu błędów. Aby można odtworzyć obiekty w ich wcześniejszej formie, trzeba zapisać informacje o stanie. Jednak zwykle obiekty kapsułkują swój stan (częściowo lub w całości), dlatego jest on niedostępny dla innych obiektów i nie można go zapisać zewnętrznie. Ujawnienie stanu spowodowałoby naruszenie kapsułkowania, co może negatywnie wpływać na niezawodność i możliwość rozszerzania aplikacji.

Rozważmy na przykład edytor graficzny z obsługą połączeń między obiektami. Kiedy użytkownik powiąże linią dwa prostokąty ze sobą, będą one połączone także po przeniesieniu jednego z nich. Edytor powoduje rozciągnięcie linii w celu zachowania powiązania.



Dobrze znanym sposobem na zachowywanie relacji połączenia między obiektami jest system rozwiązywania równań z ograniczeniami. Można zakapsułkować taki mechanizm w obiekcie **ConstraintSolver**. Obiekt ten rejestruje połączenia przy ich nawiązywaniu i generuje opisujące je równania matematyczne. Równania te rozwiązuje za każdym razem, kiedy użytkownik utworzy połączenie lub w inny sposób zmodyfikuje diagram. Obiekt **ConstraintSolver** wykorzystuje wyniki tych obliczeń do zmiany uporządkowania grafiki, aby zachowane zostały w niej właściwe połączenia.

Obsługa cofania działań w tej aplikacji nie jest tak prosta, jak się to może wydawać. Oczywisty sposób na anulowanie operacji polega na zapisaniu odległości, o którą przesunięto obiekt, i przeniesieniu go z powrotem o tę samą odległość. Jednak nie gwarantuje to, że wszystkie obiekty pojawią się w wyjściowych miejscach. Założymy, że połączenie jest luźne. Wtedy samo przeniesienie prostokąta w jego pierwotne położenie nie zawsze pozwoli uzyskać pożądany efekt.



Publiczny interfejs klasy `ConstraintSolver` może okazać się niewystarczający do precyzyjnego anulowania efektów jej działania. Mechanizm cofania musi ściślej współdziałać z obiektami `ConstraintSolver` przy przywracaniu wcześniejszego stanu, jednak powinniśmy też uniknąć udostępniania wspomnianemu mechanizmowi wewnętrznych elementów klasy `ConstraintSolver`.

Wzorzec Pamiątki pozwala rozwiązać ten problem. **Pamiątka** to obiekt przechowujący zapis wewnętrznego stanu innego obiektu — **źródła** pamiętki. Mechanizm cofania zażąda pamiętać od źródła, kiedy będzie potrzebował zarejestrować jego stan. Źródło inicjuje pamiętkę informacjami określającymi jego bieżący stan. Tylko źródło może zapisywać informacje w pamiętce i pobierać je z niej. Dla innych obiektów pamiętka jest nieprzezroczysta.

W omówionym wcześniej przykładowym edytorze graficznym obiekt `ConstraintSolver` może działać jak źródło. Poniższa sekwencja zdarzeń opisuje proces cofania:

1. Edytor żąda pamiętki od obiektu `ConstraintSolver` w ramach operacji przenoszenia.
2. Obiekt `ConstraintSolver` tworzy i zwraca pamiętkę. Tu jest to egzemplarz klasy `SolverState`. Pamiątka `SolverState` obejmuje struktury danych opisujące bieżący stan wewnętrznych równań i zmiennych obiektu `ConstraintSolver`.
3. Później, przy cofaniu przez użytkownika operacji przenoszenia, edytor przekazuje obiekt `SolverState` z powrotem do obiektu `ConstraintSolver`.
4. Na podstawie informacji z obiektu `SolverState` obiekt `ConstraintSolver` zmienia swoje wewnętrzne struktury, aby precyzyjnie przywrócić wcześniejszy stan równań i zmiennych.

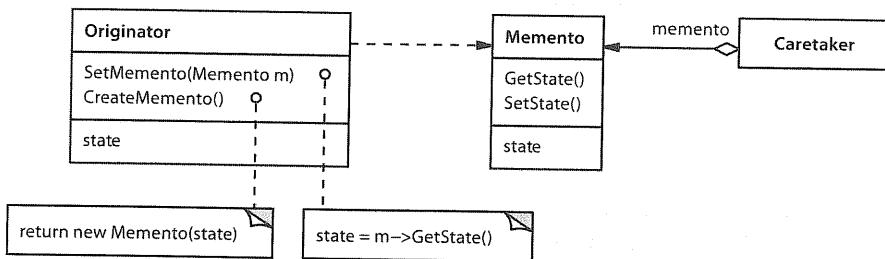
To rozwiązanie umożliwia obiektom `ConstraintSolver` powierzenie innym obiektom informacji potrzebnych do przywrócenia wcześniejszego stanu bez udostępniania przy tym wewnętrznej struktury i reprezentacji.

WARUNKI STOSOWANIA

Wzorca Pamiątki należy używać w następujących warunkach:

- Kiedy trzeba zachować obraz (części) stanu obiektu w celu jego późniejszego odtworzenia w tym stanie *oraz*
- bezpośredni interfejs do pobierania stanu spowodowałby ujawnienie szczegółów implementacji i naruszenie kapsułkowania obiektu.

STRUKTURA



ELEMENTY

► **Memento** (*SolverState*), czyli pamiętka:

- Przechowuje wewnętrzny stan obiektu **Originator**. Pamiątka może obejmować dowolną określana przez źródło część jego wewnętrznego stanu.
- Chroni dane przed dostępem przez obiekty inne niż źródło. Pamiątki mają w istocie dwa interfejsy. Zarządcy widzi *zawężony* interfejs pamiętek i może jedynie przekazywać ją innym obiektom, natomiast źródło ma dostęp do *pełnego* interfejsu, umożliwiającego dostęp do wszystkich danych potrzebnych do przywrócenia swojego wcześniejszego stanu. W idealnych warunkach dostęp do wewnętrznego stanu pamiętki ma tylko źródło, które ją utworzyło.

► **Originator** (*ConstraintSolver*), czyli źródło:

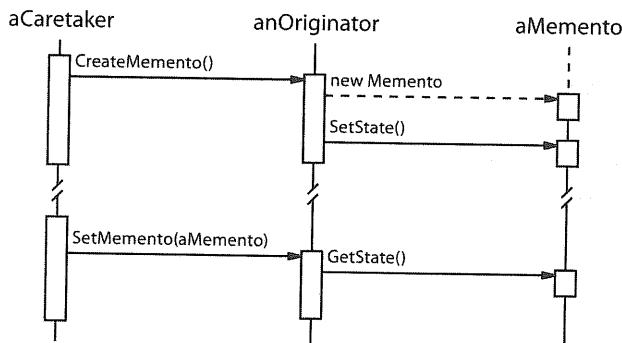
- tworzy pamiętkę obejmującą zapis wewnętrznego stanu źródła;
- korzysta z pamiętek do przywrócenia swojego wewnętrznego stanu.

► **Caretaker** (*mechanizm cofania*), czyli zarządcy:

- odpowiada za zarządzanie pamiętką;
- nigdy nie manipuluje zawartością pamiętki ani jej nie sprawdza.

WSPÓŁDZIAŁANIE

► Zarządcy żąda od źródła udostępnienia pamiętki, przechowuje ją przez określony czas i przekazuje z powrotem do źródła. Ilustruje to poniższy diagram interakcji:



Czasem zarządcą nie przekazuje pamiątki z powrotem do źródła, ponieważ nie trzeba przywracać jego wcześniejszego stanu.

- ▶ Pamiątki są pasywne. Tylko źródło, które utworzyło pamiątkę, przypisuje do niej stan i pobiera go.

KONSEKWENCJE

Stosowanie wzorca Pamiątka ma kilka konsekwencji:

1. *Zachowanie granic kapsułkowania.* Pamiątka pozwala uniknąć udostępniania informacji, które trzeba zapisać poza źródłem, choć tylko ono powinno nimi zarządzać. Wzorzec ten oddziela inne obiekty od potencjalnie złożonych mechanizmów wewnętrznych klasy Originator, co pozwala zachować granice kapsułkowania.
2. *Uproszczenie klasy Originator.* W innych projektach zachowujących kapsułkowanie to klasa Originator przechowuje wersje stanu wewnętrznego zażądane przez klienty. Wymaga to obsługi przez klasę Originator wszystkich zadań związanych z zarządzaniem pamięcią. Wykorzystanie klientów do zarządzania żądanym stanem pozwala uproszczyć klasę Originator, a ponadto klienty nie muszą powiadomić źródła o zakończeniu wykonywania swoich zadań.
3. *Korzystanie z pamiątek może być kosztowne.* Pamiątki mogą powodować znaczące koszty, jeśli obiekty Originator muszą kopiować duże ilości informacji, które trzeba zapisać w pamiątkę, lub jeżeli klienty często tworzą pamiątki i zwracają je do źródła. Jeśli kapsułkowanie i przywracanie stanu obiektów Originator jest kosztowne, wzorzec może okazać się nieodpowiedni. Zobacz analizę przyrostowych zmian w punkcie Implementacja.
4. *Definiowanie zawężonego i pełnego interfejsu.* W niektórych językach zagwarantowanie, że tylko źródło ma dostęp do stanu pamiątki, może okazać się trudne.
5. *Ukryte koszty zarządzania pamiątkami.* Zarządcą odpowiada za usuwanie powiązanych z nim pamiątek. Nie wie jednak, jak rozbudowany jest stan zapisany w pamiątkach. Dlatego prosty w innych aspektach zarządcą może spowodować duże koszty związane z pamięcią przy przechowywaniu pamiątki.

IMPLEMENTACJA

Oto dwie kwestie, które trzeba rozważyć przy implementowaniu wzorca Pamiątka:

1. *Pomocne mechanizmy w języku.* Pamiątki mają dwa interfejsy — pełny dla źródeł i zawężony dla pozostałych obiektów. W idealnych warunkach język użyty do implementacji powinien zapewniać dwa poziomy statycznej ochrony. Język C++ umożliwia uzyskanie takiego efektu przez zadeklarowanie klasy Originator jako zaprzyjaźnionej z klasą Memento i utworzenie pełnego interfejsu klasy Memento jako prywatnego. Tylko interfejs zawężony należy zadeklarować jako publiczny. Na przykład:

```
class State;

class Originator {
public:
    Memento* CreateMemento();
    void SetMemento(const Memento*);
```

```

    // ...
private:
    State* _state; // Wewnętrzne struktury danych.
    // ...
};

class Memento {
public:
    // Zawężony interfejs publiczny.
    virtual ~Memento();
friend class Originator;
private:
    // Składowe prywatne dostępne tylko dla obiektów Originator.
    friend class Originator;
    Memento();

    void SetState(State* );
    State* GetState();
    // ...
private:
    State* _state;
    // ...
};

```

2. *Zapisywanie przyrostowych zmian.* Kiedy pamiątki są tworzone i przekazywane z powrotem do źródła w przewidywalnej kolejności, wtedy w klasie *Memento* wystarczy zapisać *przyrostowe zmiany* w wewnętrznym stanie źródła.

Na przykład możliwe do cofnięcia polecenia w ich historii mogą korzystać z pamiątek, aby zagwarantować, że po anulowaniu zostaną dokładnie przywrócone do wcześniejszego stanu (zobacz wzorzec *Polecenie*, s. 302). Historia poleceń określa specyficzną kolejność ich cofania i powtarzania. Oznacza to, że w pamiątkach wystarczy zapisać przyrostowe zmiany wywoływanie przez polecenia. Nie trzeba rejestrować pełnego stanu każdego obiektu, na który wpływa dane polecenie. We wcześniejszym przykładzie z punktu „Uzasadnienie” w celu zachowania linii łączącej prostokąty wystarczy w obiekcie *ConstraintSolver* zapisać tylko te wewnętrzne struktury, które zmodyfikowano. Nie trzeba przechowywać pozycji zajmowanych przez te obiekty.

PRZYKŁADOWY KOD

Przedstawiony tu kod w języku C++ ilustruje opisany wcześniej przykład z klasą *ConstraintSolver*. Do przenoszenia obiektów graficznych z jednego miejsca w inne oraz cofania tego procesu używamy obiektów *MoveCommand* (zobacz wzorzec *Polecenie*, s. 302). Edytor graficzny przy przemieszczaniu obiektu graficznego wywołuje operację *Execute* polecenia, natomiast przy anulowaniu — operację *Unexecute*. Polecenie zapisuje obiekt docelowy, odległość jego przesunięcia i egzemplarz klasy *ConstraintSolverMemento* (jest to pamiątka ze stanem obiektu *ConstraintSolver*).

```

class Graphic;
// Klasa bazowa obiektów graficznych w edytorze graficznym.

class MoveCommand {
public:

```

```

MoveCommand(Graphic* target, const Point& delta);
void Execute();
void Unexecute();
private:
    ConstraintSolverMemento* _state;
    Point _delta;
    Graphic* _target;
};

```

Ograniczenia połączenia są określane przez klasę ConstraintSolver. Jej kluczową funkcją składową jest Solve. Przetwarza ona ograniczenia zarejestrowane za pomocą operacji AddConstraint. Aby umożliwić cofanie, stan obiektów ConstraintSolver można zapisać zewnętrznie w egzemplarzu klasy ConstraintSolverMemento za pomocą operacji CreateMemento. Obiekt ConstraintSolver można przywrócić do wcześniejszego stanu przez wywołanie operacji SetMemento. Klasa ConstraintSolver jest singletonem (s. 130).

```

class ConstraintSolver {
public:
    static ConstraintSolver* Instance();

    void Solve();
    void AddConstraint(
        Graphic* startConnection, Graphic* endConnection
    );
    void RemoveConstraint(
        Graphic* startConnection, Graphic* endConnection
    );

    ConstraintSolverMemento* CreateMemento();
    void SetMemento(ConstraintSolverMemento*);

private:
    // Złożone zmienne stanu i operacje potrzebne do
    // utrzymania połączeń.
};

class ConstraintSolverMemento {
public:
    virtual ~ConstraintSolverMemento();
private:
    friend class ConstraintSolver;
    ConstraintSolverMemento();

    // Prywatne zmienne stanu obiektu ConstraintSolver.
};

```

Na podstawie tych interfejsów możemy zaimplementować składowe Execute i Unexecute klasy MoveCommand w następujący sposób:

```

void MoveCommand::Execute () {
    ConstraintSolver* solver = ConstraintSolver::Instance();
    _state = solver->CreateMemento(); // Tworzenie pamiętki.
    _target->Move(_delta);
    solver->Solve();
}

```

```

    }

void MoveCommand::Unexecute () {
    ConstraintSolver* solver = ConstraintSolver::Instance();
    _target->Move(-_delta);
    solver->SetMemento(_state); // Przywrócenie stanu obiektu ConstraintSolver.
    solver->Solve();
}

```

Operacja Execute przed przesunięciem obiektu graficznego pobiera pamiętkę Constraint \rightarrow SolverMemento. Operacja Unexecute przenosi grafikę z powrotem, przywraca wcześniejszy stan obiektu ConstraintSolver i na zakończenie nakazuje temu obiekowi przetworzenie ograniczeń.

ZNANE ZASTOSOWANIA

Wcześniej przykładowy kod jest oparty na mechanizmie obsługi połączeń przez klasę CSolver w platformie Unidraw [VL90].

Kolekcje w języku Dylan [App92] udostępniają interfejs do iteracji zgodny ze wzorcem Pamiątka. W kolekcjach tych wykorzystywany jest obiekt stanu. Jest to pamiętka reprezentująca stan iteracji. Każda kolekcja może przedstawać bieżący stan iteracji w dowolny wybrany sposób. Reprezentacja ta jest całkowicie ukryta przed klientami. Mechanizm iteracji zastosowany w języku Dylan można przekształcić na następujący kod w języku C++:

```

template <class Item>
class Collection {
public:
    Collection();

    IterationState* CreateInitialState();
    void Next(IterationState* );
    bool IsDone(const IterationState*) const;
    Item CurrentItem(const IterationState*) const;
    IterationState* Copy(const IterationState*) const;

    void Append(const Item& );
    void Remove(const Item& );
    // ...
};


```

Operacja CreateInitialState zwraca zainicjowany obiekt IterationState powiązany z określoną kolekcją. Operacja Next przenosi obiekt stanu do następnej pozycji w iteracji, co powoduje zwiększenie indeksu. Operacja IsDone zwraca wartość true, jeśli operacja Next przeszła poza ostatni element kolekcji. Operacja CurrentItem przeprowadza dereferencję obiektu stanu i zwraca element kolekcji wskazywany przez ten obiekt. Operacja Copy zwraca kopię danego obiektu stanu. Jest to przydatne do zaznaczania miejsca w iteracji.

Załóżmy, że korzystamy z klasy `ItemType`. Możemy przejść po kolekcji jej egzemplarzy w następujący sposób⁹:

```
class ItemType {
public:
    void Process();
    // ...
};

Collection<ItemType*> aCollection;
IterationState* state;

state = aCollection.CreateInitialState();
while (!aCollection.IsDone(state)) {
    aCollection.CurrentItem(state)->Process();
    aCollection.Next(state);
}
delete state;
```

Interfejs do iterowania oparty na pamiętce ma dwie ciekawe zalety:

1. Można manipulować jedną kolekcją za pomocą kilku stanów (to samo dotyczy wzorca Iterator, s. 230).
2. Nie trzeba naruszać kapsułkowania kolekcji, aby umożliwić iterowanie. Pamiątkę przetwarza tylko sama kolekcja. Żaden inny obiekt nie ma dostępu do pamiętki. Pozostałe techniki iteracji wymagają naruszenia kapsułkowania przez utworzenie klas iteratora jako zaprzyjaźnionych z klasami kolekcji (zobacz wzorzec Iterator, s. 230). W implementacji opartej na pamiętce zastosowano odwrotne podejście — to `Collection` jest klasą zaprzyjaźnioną klasy `IteratorState`.

W pakiecie do rozwiązywania równań z ograniczeniami QOCA pamiętki służą do przechowywania informacji przyrostowych [HHMV92]. Klienci mogą otrzymać pamiętkę określającą bieżące rozwiązywanie systemu ograniczeń. Pamiątka obejmuje tylko te zmienne ograniczeń, które zmieniły się od czasu utworzenia ostatniego rozwiązania. Zwykle w każdym nowym rozwiązyaniu zmienia się tylko mały podzbiór zmiennych obiektu rozwiązyjącego równanie. Podzbiór ten pozwala takiemu obiekowi powrócić do poprzedniego rozwiązania. Wymaga to przywrócenia pamiętek reprezentujących rozwiązania pośrednie. Dlatego nie można ustawić pamiętek w dowolnej kolejności. W pakiecie QOCA do powracania do wcześniejszych rozwiązań wykorzystano mechanizm historii.

POWIĄZANE WZORCE

Polecenie (s. 302): polecenia mogą korzystać z pamiętek do zachowywania stanu na potrzeby operacji umożliwiających cofanie.

Iterator (s. 230): pamiętki można wykorzystać do iterowania w opisany wcześniej sposób.

Warto zauważyć, że przykładowy kod usuwa obiekt stanu po zakończeniu iterowania. Jednak jeśli operacja `ProcessItem` zgłosi wyjątek, operacja `delete` nie zostanie wywołana, co doprowadzi do zaśmiecenia pamięci. Jest to problem w języku C++, ale już nie w języku Dylan, w którym działa przywracanie pamięci. Rozwiązanie tego problemu omawiamy na stronie 266.

POLECENIE (COMMAND)

obiektowy, operacyjny

PRZEZNACZENIE

Kapsuluje żądanie w formie obiektu. Umożliwia to parametryzację klienta przy użyciu różnych żądań oraz umieszczanie żądań w kolejkach i dziennikach, a także zapewnia obsługę cofania operacji.

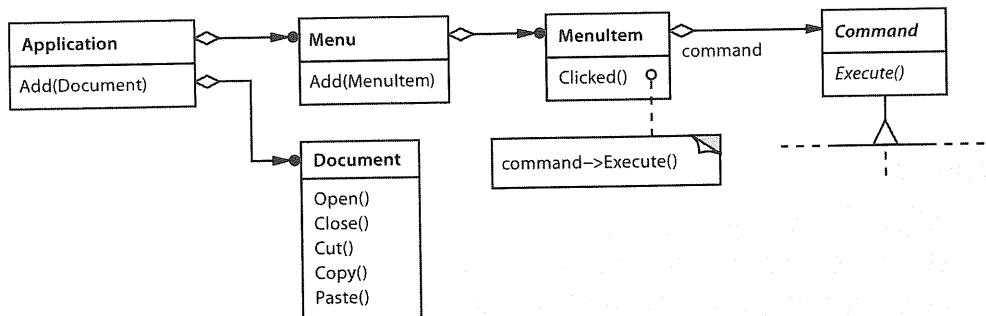
INNE NAZWY

Akcja (ang. *action*), transakcja (ang. *transaction*).

UZASADNIENIE

Czasem trzeba przesyłać żądanie do obiektów mimo braku informacji o żądanym operacjach lub odbiorcy żądania. Na przykład pakiety narzędziowe do tworzenia interfejsu użytkownika obejmują obiekty, takie jak przyciski i menu, obsługujące żądanie w odpowiedzi na działania użytkownika. Jednak w pakiecie narzędziowym nie można zaimplementować żądań bezpośrednio w przyciskach lub menu, ponieważ tylko autor aplikacji opartej na danym pakiecie wie, co powinny robić poszczególne obiekty. Projektant pakietu nie ma możliwości ustalenia odbiorcy żądania lub operacji używanych do ich obsługi.

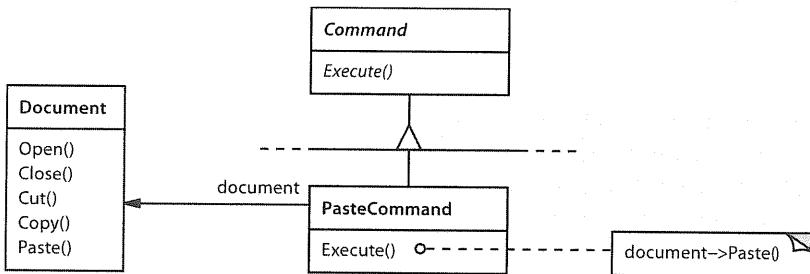
Wzorzec Polecenie umożliwia obiektom w pakietach narzędziowych przesyłanie żądań do nieokreślonych obiektów aplikacji przez przekształcenie samego żądania w obiekt. Taki obiekt można zapisać i przekazać w taki sam sposób jak inne obiekty. Kluczowym elementem tego wzorca jest klasa abstrakcyjna Command. Obejmuje ona deklarację interfejsu służącego do wykonywania operacji. W najprostszej postaci interfejs ten zawiera operację abstrakcyjną Execute. Podklasy konkretne klasy Command określają pary odbiorca-działanie przez przechowywanie odbiorcy jako zmiennej egzemplarza i udostępnianie implementacji operacji Execute służącej do wywoływania żądania. Informacje potrzebne do obsłużenia żądania ma odbiorca.



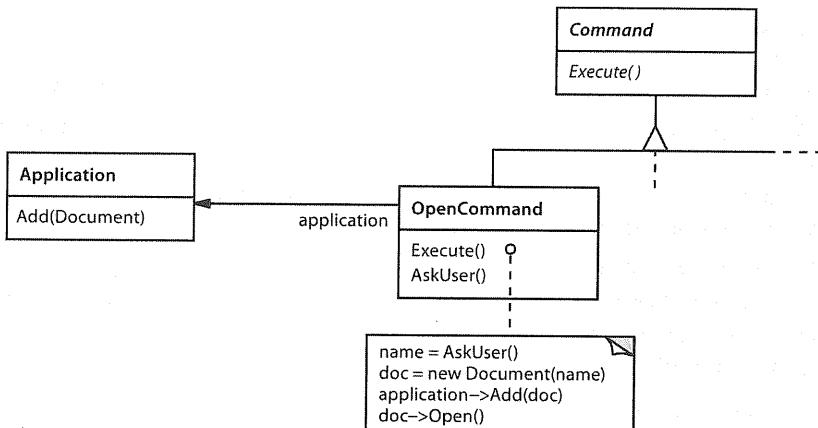
Za pomocą obiektów Command można łatwo zaimplementować menu. W klasie Menu każda opcja to egzemplarz klasy MenuItem. Klasa Application tworzy menu i opcje, a także pozostaje częścią interfejsu użytkownika. Ponadto klasa Application śledzi obiekty Document otworzone przez użytkownika.

Aplikacja konfiguruje każdy obiekt MenuItem za pomocą egzemplarza podklasy konkretnej klasy Command. Kiedy użytkownik wybierze obiekt MenuItem, obiekt ten wywoła instrukcję Execute powiązanego z nim obiektu Command, która wykona potrzebne operacje. Obiekty MenuItem nie wiedzą, z której podklasy klasy Command korzystają. Podklasy te przechowują odbiorcę żądania i wywołują przynajmniej jedną jego operację.

Na przykład operacja PasteCommand obsługuje wklejanie tekstu ze schowka do obiektu Document. Odbiorcą polecenia PasteCommand jest obiekt Document podawany w czasie tworzenia egzemplarza klasy tego polecenia. Operacja Execute wywołuje operację Paste odbiorcy (obiektu Document).

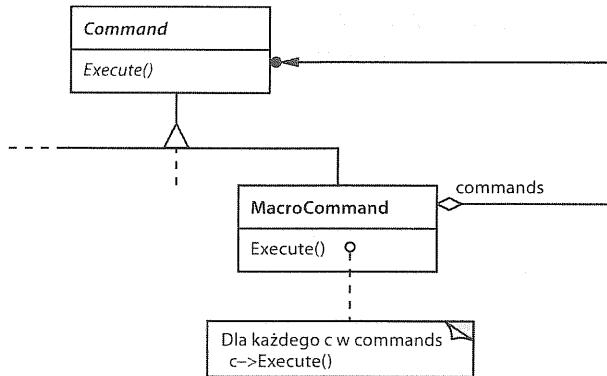


Operacja Execute podklasy OpenCommand działa inaczej. Żąda od użytkownika podania nazwy dokumentu, tworzy odpowiedni obiekt Document, dodaje go do aplikacji odbiorczej i otwiera dokument.



Czasem obiekt MenuItem musi wykonać *sekwencję* poleceń. Na przykład obiekt MenuItem przeznaczony do wyśrodkowywania tekstu oraz ustawiania normalnego rozmiaru czcionki można zbudować z obiektów CenterDocumentCommand i NormalSizeCommand. Ponieważ łączenie poleceń w takie łańcuchy to powszechnie stosowana technika, można zdefiniować klasę MacroCommand, aby umożliwić obiektom MenuItem wykonywanie dowolnej liczby poleceń.

MacroCommand to podklasa konkretna klasy **Command** przeznaczona do wykonywania sekwencji poleceń. Podklasa ta nie ma bezpośredniego odbiorcy, ponieważ polecenia w sekwencji mają zdefiniowanych własnych odbiorców.



W każdym z tych przykładów warto zauważyć, że wzorzec Polecenie oddziela obiekt wywołujący operację od obiektu, który potrafi ją wykonać. Zapewnia to dużą elastyczność przy projektowaniu interfejsu użytkownika. Aby w aplikacji umożliwić wywoływanie funkcji za pomocą opcji menu i przycisku, wystarczy sprawić, aby elementy te współużytkowały egzemplarz tej samej podklasy konkretnej klasy Command. Polecenia można zastępować dynamicznie, co jest przydatne przy implementowaniu menu zależnych od kontekstu. Można też dodać obsługę tworzenia skryptów opartych na poleceniach przez złączanie poszczególnych poleceń w większe jednostki. Wszystko to jest możliwe, ponieważ obiekt zgłaszający żądanie wie tylko, jak je przekazać, natomiast nie musi znać sposobu jego obsługi.

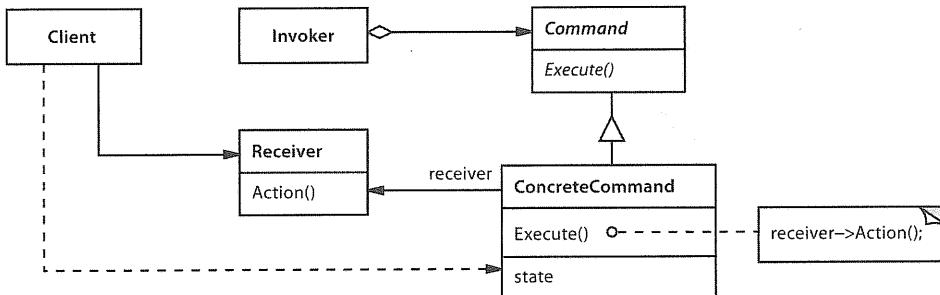
WARUNKI STOSOWANIA

Z wzorca Polecenie powinieneś korzystać do wykonywania następujących zadań:

- ▶ Do parametryzowania obiektów za pomocą wykonywanych działań (tak jak obiektów MenuItem we wcześniejszym przykładzie). W językach proceduralnych do parametryzowania można zastosować funkcje **zwrotne** rejestrowane w celu późniejszego ich wywołania. Polecenia to obiektowe odpowiedniki funkcji zwrotnych.
- ▶ Do określania, kolejkowania i wywoływania żądania w różnych miejscach programu. Czas życia obiektu Command nie musi zależeć od pierwotnego żądania. Jeśli możliwe jest przedstawienie odbiorcy żądania w sposób niezależny od przestrzeni adresowej, można przekazać obiekt polecenia związany z żądaniem do innego procesu i tam je obsługiwać.
- ▶ Do umożliwiania cofania zmian. Operacja Execute obiektu Command może w samym poleceniu przechowywać stan potrzebny do anulowania efektów jej działania. Do interfejsu klasy Command trzeba dodać operację Unexecute, która odwróci skutki wcześniejszego wywołania operacji Execute. Historia poleceń jest zapisywana na liście. Dzięki temu można cofać i powtarzać dowolną liczbę operacji przez przechodzenie na liście wstecz i naprzód oraz wywoływanie odpowiednich operacji — Unexecute i Execute.
- ▶ Do obsługi rejestrowania zmian, aby można je ponownie przeprowadzić w przypadku awarii systemu. Przez wzbogacenie interfejsu klasy Command o operacje wczytywania i zapisywania poleceń można utworzyć trwały dziennik zmian. Przywrócenie systemu po awarii polega na wczytaniu zarejestrowanych poleceń z dysku i ponownym wykonaniu ich za pomocą operacji Execute.

- ▶ Do budowania systemu na podstawie wysokopoziomowych operacji opartych na prostych operacjach. Taką strukturę powszechnie spotyka się w systemach informatycznych z obsługą transakcji. Transakcja kapsuluje zbiór zmian w danych. Wzorzec Polecenie umożliwia modelowanie transakcji. Polecenia mają wspólny interfejs, co umożliwia przeprowadzanie wszystkich transakcji w taki sam sposób. Wzorzec ten ułatwia też rozszerzanie systemu o nowe transakcje.

STRUKTURA



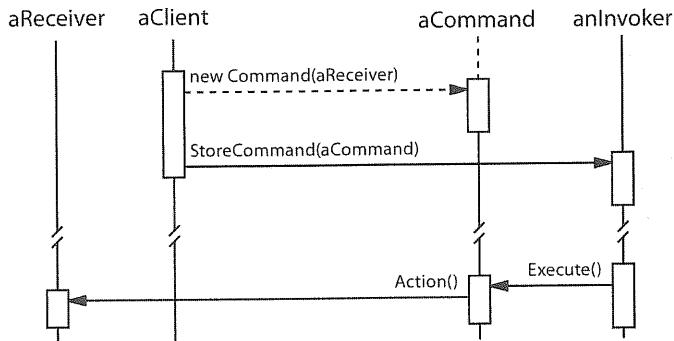
ELEMENTY

- ▶ **Command**, czyli polecenie:
 - obejmuje deklarację interfejsu przeznaczonego do wykonywania operacji.
- ▶ **ConcreteCommand** (PasteCommand, OpenCommand), czyli polecenie konkretne:
 - definiuje powiązanie między obiektem Receiver i działaniem;
 - obejmuje implementację operacji Execute w postaci wywołania odpowiednich operacji obiektu Receiver.
- ▶ **Client** (Application):
 - tworzy obiekt ConcreteCommand i określa powiązanego z nim odbiorcę.
- ▶ **Invoker** (MenuItem), czyli nadawca:
 - żąda obsłużenia żądania od polecenia.
- ▶ **Receiver** (Document, Application), czyli odbiorca:
 - potrafi wykonać operacje potrzebne do obsługi żądania. Funkcję odbiorcy może pełnić dowolna klasa.

WSPÓŁDZIAŁANIE

- ▶ Klient tworzy obiekt ConcreteCommand i określa powiązanego z nim odbiorcę.
- ▶ Obiekt Invoker przechowuje obiekt ConcreteCommand.
- ▶ Obiekt Invoker zgłasza żądanie przez wywołanie operacji Execute obiektu polecenia. Jeśli polecenia można cofać, obiekt ConcreteCommand przed wykonaniem tej operacji zapisuje stan na potrzeby anulowania zmian.
- ▶ Obiekt ConcreteCommand wywołuje operacje odbiorcy, aby obsłużyć żądanie.

Poniższy diagram przedstawia interakcje między omawianymi obiektami. Pokazaliśmy tu, w jaki sposób obiekt Command oddziela nadawcę od odbiorcy (i obsługiwanej przez niego żądania).



KONSEKWENCJE

Stosowanie wzorca Polecenie ma następujące konsekwencje:

1. Obiekt Command oddziela obiekt wywołujący operację od tego, który potrafi ją wykonać.
2. Polecenia to standardowe obiekty. Można nimi manipulować i rozszerzać je w taki sam sposób jak inne obiekty.
3. Polecenia można połączyć w polecenie złożone. Przykładem takiego rozwiązania jest opisana wcześniej klasa MacroCommand. Polecenia złożone są zwykle tworzone zgodnie z wzorcem Kompozyt (s. 170).
4. Dodawanie nowych obiektów Command jest proste, ponieważ nie wymaga modyfikowania istniejących klas.

IMPLEMENTACJA

W czasie implementowania wzorca Polecenie należy rozważyć następujące kwestie:

1. *Jak „inteligentne” powinno być polecenie?* Polecenia mogą mieć bardzo zróżnicowane możliwości. Z jednej strony mogą jedynie definiować powiązanie między odbiorcą i działaniami potrzebnymi do obsługi żądania. Z drugiej strony mogą obejmować implementację wszystkich operacji i nie delegować żadnych zadań do odbiorcy. To ostatnie skrajne rozwiązanie jest przydatne, kiedy programista chce definiować polecenia niezależne od istniejących klas, kiedy nie istnieje odpowiedni odbiorca lub kiedy polecenie ustala odbiorcę pośrednio. Na przykład polecenie tworzące nowe okno aplikacji może równie dobrze nadawać się do jego wygenerowania jak dowolny inny obiekt. Pomiędzy tymi skrajnościami możemy umieścić polecenia, które mają wystarczająco dużo informacji, aby dynamicznie określić odbiorcę.
2. *Obsługiwanie cofania i powtarzania operacji.* Polecenia mogą obsługiwać cofanie i powtarzanie operacji, jeśli udostępniają sposób na anulowanie skutków ich działania (na przykład operację Unexecute lub Undo). W klasie ConcreteCommand może to wymagać przechowywania dodatkowych informacji o stanie. Mogą one obejmować:

- ▶ obiekt Receiver wykonujący operacje w odpowiedzi na zgłoszenie żądania;
- ▶ argumenty operacji wykonywanych przez odbiorcę;
- ▶ pierwotne wartości zapisane w odbiorcy, które mogą zmienić się w procesie obsługiwanego żądania. Odbiorca musi udostępnić operacje umożliwiające przywrócenie jego wyjściowego stanu poleceniu.

Aby zapewnić obsługę cofania jednej operacji, w aplikacji wystarczy przechowywać tylko ostatnio wykonane polecenie. Do obsługi wielu poziomów anulowania i powtarzania działań potrzebna jest lista **historii poleceń**. Maksymalna długość tej listy wyznacza liczbę poziomów cofania i powtarzania operacji. Historia poleceń obejmuje sekwencję wykonanych instrukcji. Przechodzenie do tyłu listy i cofanie poleceń anuluje efekty ich uruchomienia. Poruszanie się naprzód i wykonywanie poleceń powoduje ich ponowne uruchomienie.

Polecenie umożliwiające cofnięcie czasem trzeba skopiować przed umieszczeniem w historii. Dzieje się tak, ponieważ obiekt polecenia realizujący pierwotne żądanie (na przykład od obiektu `MenuItem`) później będzie przetwarzał inne żądania. Kopiowanie jest potrzebne do odróżnienia różnych wywołań tego samego polecenia, jeśli jego stan między wywołaniami może się zmieniać.

Na przykład w obiekcie `DeleteCommand` usuwającym inne obiekty trzeba przy każdym wywołaniu zapisać inny zbiór obiektów. Dlatego kiedy obiekt `DeleteCommand` wykona swoje zadanie, trzeba go skopiować i umieścić duplikat w historii. Jeśli stan polecenia w czasie jego wykonywania się nie zmienia, kopiowanie nie jest konieczne. Wtedy w historii wystarczy umieścić referencję do polecenia. Polecenia, które trzeba skopiować przed zapisaniem w historii, działają jak prototypy (zobacz wzorzec Prototyp, s. 120).

3. *Unikanie nawarstwiania się błędów w procesie cofania poleceń*. Histereza może utrudniać utworzenie niezawodnego i zachowującego semantykę mechanizmu cofania oraz powtarzania poleceń. Błędy mogą się nawarstwiać przy wielokrotnym wykonywaniu, anulowaniu i ponownym uruchamianiu poleceń, dlatego stan aplikacji ostatecznie może różnić się od pierwotnego. Dlatego czasem konieczne jest przechowywanie w poleceniu dodatkowych informacji, aby zagwarantować, że przywrócony obiekt będzie miał wyjściowy stan. Aby zapewnić poleceniu dostęp do takich informacji bez ujawniania wewnętrznych danych innych obiektów, można zastosować wzorzec Pamiątka (s. 294).
4. *Korzystanie z szablonów języka C++*. Jeśli polecenie (1) nie umożliwia cofania i (2) nie wymaga argumentów, możemy zastosować szablon językowy C++, aby uniknąć tworzenia podklasy klasy `Command` dla działań i odbiorców każdego rodzaju. Zastosowanie tej techniki przedstawiamy w punkcie „Przykładowy kod”.

RZYKŁADOWY KOD

Przedstawiony tu kod w języku C++ to zarys implementacji klas z rodziny `Command` wymienionych w punkcie „Uzasadnienie”. Zdefiniujemy tu klasy `OpenCommand`, `PasteCommand` i `MacroCommand`. Zacznijmy od klasy abstrakcyjnej `Command`:

```
class Command {
public:
    virtual ~Command();
```

```

    virtual void Execute() = 0;
protected:
    Command();
};

```

Klasa OpenCommand otwiera dokument o nazwie podanej przez użytkownika. Do klasy OpenCommand trzeba w konstruktorze przekazać obiekt Application. AskUser to procedura pomocnicza żądająca od użytkownika podania nazwy otwieranego dokumentu.

```

class OpenCommand : public Command {
public:
    OpenCommand(Application*);

    virtual void Execute();
protected:
    virtual const char* AskUser();
private:
    Application* _application;
    char* _response;
};

OpenCommand::OpenCommand (Application* a) {
    _application = a;
}

void OpenCommand::Execute () {
    const char* name = AskUser();

    if (name != 0) {
        Document* document = new Document(name);
        _application->Add(document);
        document->Open();
    }
}

```

Do klasy PasteCommand jako odbiorcę trzeba przekazać obiekt Document. Odbiorca jest powinien być podawany jako parametr w konstruktorze klasy PasteCommand.

```

class PasteCommand : public Command {
public:
    PasteCommand(Document*);

    virtual void Execute();
private:
    Document* _document;
};

PasteCommand::PasteCommand (Document* doc) {
    _document = doc;
}

void PasteCommand::Execute () {
    _document->Paste();
}

```

Do tworzenia prostych poleceń, które nie umożliwiają cofania i nie wymagają argumentów, można użyć szablonu klasy w celu spараметryzowania odbiorcy polecenia. Na potrzeby takich poleceń zdefiniujemy podklasę szablonową SimpleCommand. Jest ona spараметryzowana za pomocą typu Receiver oraz przechowuje powiązanie między obiektem odbiorcy i działaniem zapisanym jako wskaźnik do funkcji składowej.

```
template <class Receiver>
class SimpleCommand : public Command {
public:
    typedef void (Receiver::* Action)();
    SimpleCommand(Receiver* r, Action a) :
        _receiver(r), _action(a) {}

    virtual void Execute();
private:
    Action _action;
    Receiver* _receiver;
};
```

Konstruktor zapisuje odbiorcę i działanie w odpowiednich zmiennych egzemplarza. Operacja Execute po prostu wywołuje działanie odbiorcy.

```
template <class Receiver>
void SimpleCommand<Receiver>::Execute () {
    (_receiver->*_action)();
}
```

Aby utworzyć polecenie wywołujące operację Action klasy MyClass, wystarczy umieścić w kliencie następujący kod:

```
MyClass* receiver = new MyClass;
// ...
Command* aCommand =
    new SimpleCommand<MyClass>(receiver, &MyClass::Action);
// ...
aCommand->Execute();
```

Warto pamiętać, że to rozwiązanie zadziała tylko dla prostych poleceń. Bardziej złożone polecenia, przechowujące nie tylko odbiorców, ale też argumenty i (lub) stan potrzebny przy cofaniu działań, wymagają utworzenia podklasy klasy Command.

Klasa MacroCommand zarządza sekwencją poleceń podległych i udostępnia operacje służące do dodawania oraz usuwania takich poleceń. Nie trzeba tu jawnie określać odbiorcy, ponieważ polecenia podległe same określają swoich odbiorców.

```
class MacroCommand : public Command {
public:
    MacroCommand();
    virtual ~MacroCommand();

    virtual void Add(Command* );
    virtual void Remove(Command* );
```

```

    virtual void Execute();
private:
    List<Command*>* _cmds;
};

```

W klasie MacroCommand najważniejsza jest funkcja składowa `Execute`. Przechodzi ona po wszystkich poleceniach podrzędnych i wywołuje operację `Execute` każdego z nich.

```

void MacroCommand::Execute () {
    ListIterator<Command*> i(_cmds);

    for (i.First(); !i.IsDone(); i.Next()) {
        Command* c = i.CurrentItem();
        c->Execute();
    }
}

```

Warto zauważyć, że jeśli klasa `MacroCommand` ma udostępniać operację `Unexecute`, polecenia podrzędne trzeba anulować w kolejności *odwrotnej* do tej określonej w implementacji operacji `Execute`.

W klasie `MacroCommand` trzeba też udostępnić operacje do zarządzania poleceniami podrzędnymi. Klasa ta odpowiada ponadto za usuwanie takich poleceń.

```

void MacroCommand::Add (Command* c) {
    _cmds->Append(c);
}

void MacroCommand::Remove (Command* c) {
    _cmds->Remove(c);
}

```

ZNANE ZASTOSOWANIA

Prawdopodobnie pierwszy przykład dotyczący wzorca Polecenie opisano w pracy Liebermana [Lie85]. System MacApp [App89] spowodował spopularyzowanie polecień do implementowania operacji umożliwiających cofanie. W ET++ [WGM88], InterViews [LCI-92] i Unidraw [VL90] także zdefiniowano klasy zgodne ze wzorcem Polecenie. W InterViews zdefiniowana jest klasa abstrakcyjna `Action` udostępniająca mechanizmy funkcjonowania polecień. Pakiet ten obejmuje też szablon `ActionCallback` parametryzowany za pomocą metody-akcji. Szablon ten umożliwia automatyczne tworzenie egzemplarzy podklas polecień.

W bibliotece klas THINK [Sym93b] polecenia wykorzystano do obsługi akcji umożliwiających cofanie. Polecenia w tej bibliotece to obiekty `Task`. Są one przekazywane w łańcuchu zobowiązań (s. 244), gdzie znajdują się przetworzone.

Obiekty polecień w platformie Unidraw są wyjątkowe, ponieważ mogą działać jak komunikaty. Polecenie w tej platformie można przesyłać w celu zinterpretowania do innego obiektu, a efekt interpretacji jest różny w zależności od odbiorcy. Ponadto odbiorca może oddelegować interpretowanie do innego obiektu (zwykle jest nim obiekt nadzorowany odbiorcy w większej strukturze,

na przykład w łańcuchu zobowiązań). Odbiorca polecenia w platformie Unidraw jest więc w większym stopniu obliczany niż przechowywany. Mechanizm interpretowania w tej platformie jest zależny od informacji o typie, dostępnych w czasie wykonywania programu.

Coplien opisuje, jak zaimplementować **funktory** (są to obiekty będące funkcjami) w języku C++ [Cop92]. Udało mu się osiągnąć pewien poziom przezroczystości w korzystaniu z nich przez przeciążenie operatora wywołania funkcji (`operator()`). Wzorzec Polecenie działa inaczej. Dotyczy przede wszystkim przechowywania powiązania pomiędzy odbiorcą i funkcją (na przykład akcją), a nie wyłącznie samej funkcji.

POWIĄZANE WZORCE

Do zaimplementowania poleceń MacroCommand można wykorzystać wzorzec Kompozyt (s. 170).

Wzorzec Pamiątka (s. 294) umożliwia zachowanie stanu potrzebnego poleceniu do cofnięcia efektów jego wykonania.

Polecenie, które trzeba skopiować przed umieszczeniem w historii, działa jak Prototyp (s. 120).

STAN (STATE)

obiektowy, operacyjny

PRZEZNACZENIE

Umożliwia obiektowi modyfikację zachowania w wyniku zmiany wewnętrznego stanu. Wygląda to tak, jakby obiekt zmienił klasę.

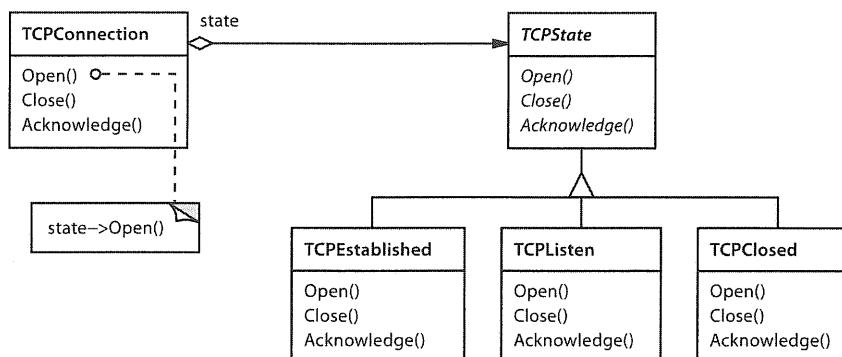
INNA NAZWA

Obiekty stanów (ang. *objects for states*).

UZASADNIENIE

Zastanówmy się nad klasą `TCPConnection` reprezentującą połaczenie sieciowe. Obiekt tej klasy przyjmuje kilka stanów — `Established` (po nawiązaniu), `Listening` (oczekiwanie), `Closed` (po zamknięciu). Kiedy obiekt `TCPConnection` odbiera żądania od innych obiektów, reaguje w różny sposób w zależności od stanu. Na przykład efekt żądania `Open` jest oparty na tym, czy połączenie znajduje się w stanie `Close` czy `Established`. Wzorzec Stan opisuje, jak obiekt `TCPConnection` może zmieniać zachowania w zależności od stanu.

Kluczowy pomysł w tym wzorcu polega na wprowadzeniu klasy abstrakcyjnej `TCPState` służącej do reprezentowania stanów połączenia sieciowego. Klasa `TCPState` obejmuje deklarację interfejsu wspólnego wszystkim klasom reprezentującym różne stany działania. W podklassach tej klasy znajduje się implementacja zachowania specyficznego dla stanu. Na przykład w klasach `TCPEstablished` i `TCPClosed` zaimplementowano działania powiązane ze stanami `Established` i `Closed` klasy `TCPConnection`.



Klasa `TCPConnection` przechowuje obiekt stanu (egzemplarz podklasy klasy `TCPState`) reprezentujący bieżący stan połączenia TCP. Klasa `TCPConnection` deleguje do tego obiektu wszystkie żądania specyficzne dla stanu. Obiekt `TCPConnection` korzysta z zapisanego w nim egzemplarza klasy `TCPState` do wykonywania operacji specyficznych dla stanu połączenia.

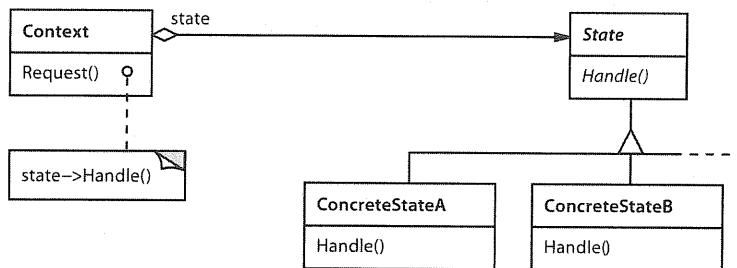
Przy każdej zmianie stanu połączenia obiekt TCPConnection zmienia używany obiekt stanu. Kiedy połączenie przejdzie ze stanu Established w stan Close, obiekt TCPConnection zastąpi egzemplarz klasy TCPEstablished egzemplarzem klasy TCPClosed.

WARUNKI STOSOWANIA

Wzorzec Stan należy stosować w następujących warunkach:

- ▶ Zachowanie obiektu zależy od jego stanu, a obiekt musi na podstawie stanu zmieniać działanie w czasie wykonywania programu.
- ▶ Operacje obejmują długie, wieloczęściowe instrukcje warunkowe zależne od stanu obiektu. Taki stan jest zwykle reprezentowany przez stałe wyliczeniowe. Często kilka operacji obejmuje tę samą strukturę warunkową. Wzorzec Stan powoduje umieszczenie każdej gałęzi takiej struktury w odrębnej klasie. Umożliwia to traktowanie stanu jak samodzielnego obiektu, który można modyfikować niezależnie od innych obiektów.

STRUKTURA



ELEMENTY

- ▶ **Context (TCPConnection):**
 - definiuje interfejs udostępniany klientom;
 - przechowuje egzemplarz podklasy klasy ConcreteState definiujący bieżący stan.
- ▶ **State (TCPState), czyli stan:**
 - definiuje interfejs do kapsułkowania zachowania związanego z określonym stanem obiektu Context.
- ▶ **Podklasy klasy ConcreteState (TCPEstablished, TCPListen, TCPClosed):**
 - każda podkلاza obejmuje implementację zachowania powiązaną ze stanem obiektu Context.

WSPÓŁDZIAŁANIE

- ▶ Obiekt Context deleguje żądania specyficzne dla stanu do bieżącego obiektu ConcreteState.
- ▶ Obiekt Context może przekazać sam siebie jako argument do obiektu State obsługującego żądanie. Umożliwia to obiekowi State uzyskanie w razie potrzeby dostępu do kontekstu.

- ▶ Obiekt Context to podstawowy interfejs dla klientów. Klienci mogą konfigurować interfejs tego obiektu za pomocą obiektów State. Po skonfigurowaniu obiektu Context klienci nie muszą bezpośrednio korzystać z obiektów State.
- ▶ Klasa Context lub podklasy klasy ConcreteState mogą określać kolejność i warunki zmiany stanów.

KONSEKWENCJE

Stosowanie wzorca Stan ma następujące konsekwencje:

1. *Pozwala umieścić w jednym miejscu zachowanie specyficzne dla stanu i rozdzielić zachowania powiązane z różnymi stanami.* Wzorzec Stan powoduje umieszczenie w jednym obiekcie całego zachowania związanego z danym stanem. Ponieważ cały kod specyficzny dla stanu znajduje się w podklasie klasy State, nowe stany i przejścia między nimi można łatwo dodać przez definiowanie nowych podkłas.

Inna możliwość to wykorzystanie wartości danych do definiowania wewnętrznych stanów i sprawdzanie ich bezpośrednio w operacjach klasy Context. Jednak wtedy w różnych miejscach implementacji klasy Context znajdują się podobne do siebie instrukcje warunkowe lub struktury case. Dodanie nowego stanu może wymagać zmodyfikowania kilku operacji, co komplikuje konserwowanie klasy.

Wzorzec Stan pozwala uniknąć tego problemu, jednak może spowodować inny, ponieważ wzorzec ten prowadzi do rozproszenia zachowania specyficznego dla różnych stanów po kilku podklasach klasy State. Rozwiążanie to zwiększa liczbę klas i jest mniej zwięzłe niż stosowanie jednej klasy. Jednak rozproszenie zachowania jest korzystne, jeśli występuje wiele stanów, które w innym podejściu wymagają tworzenia rozbudowanych instrukcji warunkowych.

Długie instrukcje warunkowe — podobnie jak długie procedury — są niepożądane. Są monolityczne i zwykle sprawiają, że kod jest mniej zrozumiałym, co z kolei utrudnia ich modyfikowanie i rozszerzanie. Wzorzec Stan to sposób na lepsze ustrukturyzowanie kodu specyficznego dla stanu. Logika określająca zmiany stanu nie znajduje się w monolitycznych instrukcjach if lub switch, ale jest podzielona na podklasy klasy State. Zakapsułkowanie wszystkich działań i przejść między stanami w klasie pozwala przekształcić stan wykonywania w pełny obiekt. Wyznacza to strukturę kodu i sprawia, że jego przeznaczenie jest bardziej zrozumiałe.

2. *Powoduje, że zmiany stanu są jawne.* Kiedy stan obiektu jest zdefiniowany całkowicie w kategoriach wewnętrznych wartości jego danych, zmiany stanu nie mają jawnej reprezentacji, ale wynikają z przypisania wartości do określonych zmiennych. Wprowadzenie odrębnych obiektów dla różnych stanów zapewnia większą jawność zmian stanu.

Ponadto obiekty State mogą chronić obiekt Context przed niespójnymi stanami wewnętrznymi, ponieważ zmiany stanu z perspektywy obiektu Context są atomowe — zachodzą w wyniku zmodyfikowania *jednej zmiennej* (przechowującej obiekt State w obiekcie Context), a nie kilku [dCLF93].

3. Stan obiektów można współużytkować. Jeśli obiekty State nie mają zmiennych egzemplarza (oznacza to, że stan reprezentowany przez te obiekty jest określany wyłącznie przez ich typ), obiekty Context mogą współużytkować obiekt State. Stan współużytkowany w ten sposób jest w istocie pyłkiem (zobacz wzorzec Pyłek, s. 201). Nie ma on wewnętrznego stanu, a jedynie udostępnia zachowanie.

IMPLEMENTACJA

Ze wzorcem Stan związane są różne zagadnienia implementacyjne:

1. *Który z elementów ma definiować zmiany stanu?* Wzorzec Stan nie określa, które elementy mają definiować kryteria zmiany stanu. Jeśli te kryteria są stałe, można je w całości zapisać w klasie Context. Jednak zwykle elastyczniejsze i bardziej odpowiednie jest umożliwienie podklasom klasy State samodzielnego określania następnego stanu oraz momentu jego zmiany. Wymaga to dodania do klasy Context interfejsu umożliwiającego obiektom State bezpośrednie ustawianie bieżącego stanu obiektu Context.

Taka decentralizacja logiki zmian stanów ułatwia jej modyfikowanie lub rozszerzanie przez definiowanie nowych podklas klasy State. Wadą tego podejścia jest to, że podklasy klasy State będą znały przynajmniej jedną inną taką podkласę, co tworzy zależności implementacyjne między nimi.

2. *Podejście oparte na tablicach.* Cargill w książce *C++ Programming Style*¹⁰ [Car92] opisuje inny sposób wyznaczania struktury kodu zależnego od stanu. Cargill wykorzystał tablice do odwzorowania danych wejściowych na zmiany stanu. Dla każdego stanu tablica odwzorowuje wszystkie możliwe dane wejściowe prowadzące do następnego stanu. Ostatecznie podejście to prowadzi do przekształcenia kodu z instrukcjami warunkowymi (i funkcjami wirtualnymi w przypadku wzorca Stan) w kod oparty na przeszukiwaniu tablicy.

Główną zaletą tablic jest ich regularność. Można zmienić kryteria przejść między stanami przez zmodyfikowanie danych zamiast kodu programu. Jednak rozwiązanie to ma też pewne wady:

- ▶ Przeszukiwanie tablicy jest często mniej wydajne niż wywołanie funkcji (wirtualnej).
- ▶ Umieszczenie logiki zmian w jednolitym formacie tabelarycznym sprawia, że kryteria przejść są mniej jednoznaczne, a dlatego także trudniejsze do zrozumienia.
- ▶ Zwykle trudno jest dodawać działania związane ze zmianą stanu. Podejście oparte na tablicy ujmuje stany i przejścia między nimi, ale trzeba je wzbogacić o możliwość wykonywania dowolnych obliczeń przy każdej zmianie stanu.

Kluczową różnicę między maszynami stanowymi opartymi na tablicach i wzorcem Stan można podsumować w następujący sposób — wzorzec Stan modeluje zachowanie specyficzne dla stanu, natomiast przy stosowaniu tablic nacisk położony jest na definiowanie zmian stanu.

3. *Tworzenie i usuwanie obiektów State.* W czasie implementowania warto zastanowić się nad typowym dilematem: (1) czy tworzyć obiekty State tylko wtedy, kiedy są potrzebne, a następnie je usuwać, czy (2) tworzyć je na początku działania programu i nigdy ich nie likwidować.

¹⁰ Wydanie polskie: *C++ . Styl programowania*, Helion, 2003 — przyp. tłum.

Pierwsze rozwiązanie jest zalecane, kiedy w czasie wykonywania programu nie wiadomo, które stany się pojawią, oraz stan obiektów Context zmienia się rzadko. To podejście pozwala uniknąć tworzenia zbędnych obiektów, co jest ważne, jeśli obiekty State przechowują dużo informacji. Drugie rozwiązanie jest lepsze, jeżeli zmiany stanu są częste. Wtedy warto uniknąć usuwania obiektów State, ponieważ wkrótce mogą być znów potrzebne. Koszty tworzenia obiektów są ponoszone raz na początku, a koszty destrukcji nie występują. Wadą tego podejścia jest to, że obiekt Context musi przechowywać referencje do wszystkich możliwych stanów.

4. *Słosowanie dziedziczenia dynamicznego.* Zachowanie związane z określonym żądaniem można zmodyfikować przez zmienienie klasy obiektu w czasie wykonywania programu, jednak w większości obiektowych języków programowania nie jest to możliwe. Wyjątki to język Self [US87] i inne języki oparte na delegacji, które udostępniają taki mechanizm i tym samym bezpośrednio obsługują wzorzec Stan. Obiekty w języku Self mogą delegować operacje do innych obiektów, co jest pewną formą dynamicznego dziedziczenia. Zmienienie docelowego delegata w czasie wykonywania programu powoduje zmodyfikowanie struktury dziedziczenia. Ten mechanizm pozwala obiektom zmieniać zachowanie, co jest równoważne ze zmianą ich klasy.

PRZYKŁADOWY KOD

Poniższe fragmenty to kod w języku C++ dla przykładu dotyczącego połączenia TCP, opisanego w punkcie „Uzasadnienie”. Ten przykład to uproszczona wersja protokołu TCP. Nie ilustruje on kompletnego protokołu ani wszystkich stanów połączeń TCP¹¹.

Najpierw zdefiniujmy klasę TCPConnection udostępniającą interfejs do przesyłania danych i obsługującą żądania zmiany stanu.

```
class TCPOctetStream;
class TCPState;

class TCPConnection {
public:
    TCPConnection();
    void ActiveOpen();
    void PassiveOpen();
    void Close();
    void Send();
    void Acknowledge();
    void Synchronize();

    void ProcessOctet(TCPOctetStream*);
private:
    friend class TCPState;
    void ChangeState(TCPState*);
private:
    TCPState* _state;
};
```

¹¹ Ten przykład oparto na protokole połączeń TCP opisany przez Lynch'a i Rose'a [LR93].

Klasa TCPConnection przechowuje egzemplarz klasy TCPState w zmiennej składowej _state. Klasa TCPState ma ten sam interfejs do zmiany stanu co klasa TCPConnection. Każda operacja klasy TCPState przyjmuje jako parametr egzemplarz klasy TCPConnection, co umożliwia obiektom TCPState dostęp do danych z obiektów TCPConnection i zmianę stanu połączenia.

```
class TCPState {
public:
    virtual void Transmit(TCPConnection*, TCPOctetStream*);
    virtual void ActiveOpen(TCPConnection*);
    virtual void PassiveOpen(TCPConnection*);
    virtual void Close(TCPConnection*);
    virtual void Synchronize(TCPConnection*);
    virtual void Acknowledge(TCPConnection*);
    virtual void Send(TCPConnection*);
protected:
    void ChangeState(TCPConnection*, TCPState*);
};
```

Klasa TCPConnection deleguje wszystkie żądania specyficzne dla stanu do egzemplarza klasy TCPState zapisanego w zmiennej _state. Klasa TCPConnection ponadto udostępnia operację do modyfikowania tej zmiennej przez zapisanie w niej nowego obiektu TCPState. Konstruktor klasy TCPConnection inicjuje obiekt TCPState za pomocą stanu TCPClosed (definiujemy go dalej).

```
TCPConnection::TCPConnection () {
    _state = TCPClosed::Instance();
}

void TCPConnection::ChangeState (TCPState* s) {
    _state = s;
}

void TCPConnection::ActiveOpen () {
    _state->ActiveOpen(this);
}

void TCPConnection::PassiveOpen () {
    _state->PassiveOpen(this);
}

void TCPConnection::Close () {
    _state->Close(this);
}

void TCPConnection::Acknowledge () {
    _state->Acknowledge(this);
}

void TCPConnection::Synchronize () {
    _state->Synchronize(this);
}
```

W klasie TCPState zaimplementowane jest domyślne zachowanie na potrzeby wszystkich delegowanych do niej żądań. Klasa ta może też zmieniać stan obiektów TCPConnection za pomocą operacji ChangeState. TCPState jest zadeklarowana jako klasa zaprzyjaźniona klasy TCPConnection, aby miała uprzywilejowany dostęp do wspomnianej operacji.

```
void TCPState::Transmit (TCPConnection*, TCPOctetStream*) { }
void TCPState::ActiveOpen (TCPConnection*) { }
void TCPState::PassiveOpen (TCPConnection*) { }
void TCPState::Close (TCPConnection*) { }
void TCPState::Synchronize (TCPConnection*) { }

void TCPState::ChangeState (TCPConnection* t, TCPState* s) {
    t->ChangeState(s);
}
```

W podkласach klasy TCPState zaimplementowane jest zachowanie specyficzne dla stanu. Połączenie TCP może znajdować się w wielu stanach — Established, Listening, Closed itd. Dla każdego z nich istnieje podklasa klasy TCPState. Szczegółowo przedstawimy tu trzy podklasy — TCPEstablished, TCPListen i TCPClosed.

```
class TCPEstablished : public TCPState {
public:
    static TCPState* Instance();

    virtual void Transmit(TCPConnection*, TCPOctetStream*);
    virtual void Close(TCPConnection*);
};

class TCPListen : public TCPState {
public:
    static TCPState* Instance();

    virtual void Send(TCPConnection*);
    // ...
};

class TCPClosed : public TCPState {
public:
    static TCPState* Instance();

    virtual void ActiveOpen(TCPConnection*);
    virtual void PassiveOpen(TCPConnection*);
    // ...
};
```

Podklasy klasy TCPState nie mają lokalnego stanu, dlatego można wspólnie użytkować obiekty tych podklas (potrzebny jest tylko jeden egzemplarz każdej z nich). Niepowtarzalny egzemplarz każdej podklasy klasy TCPState można pozyskać za pomocą statycznej operacji Instance¹².

¹² Powoduje to, że każda podklasa klasy TCPState jest singletonem (zobacz wzorzec Singleton, s. 127).

W każdej podklasie klasy TCPState zaimplementowane jest specyficzne dla stanu zachowanie powiązane z żądaniami prawidłowymi dla danego stanu:

```

void TCPClosed::ActiveOpen (TCPConnection* t) {
    // Wysyłanie komunikatów SYN, odbieranie komunikatów SYN i ACK itd.

    ChangeState(t, TCPEstablished::Instance());
}

void TCPClosed::PassiveOpen (TCPConnection* t) {
    ChangeState(t, TCPListen::Instance());
}

void TCPEstablished::Close (TCPConnection* t) {
    // Wysyłanie komunikatów FIN, odbieranie potwierdzeń dla komunikatów FIN.

    ChangeState(t, TCPListen::Instance());
}

void TCPEstablished::Transmit (
    TCPConnection* t, TCPOctetStream* o
) {
    t->ProcessOctet(o);
}

void TCPListen::Send (TCPConnection* t) {
    // Wysyłanie komunikatów SYN, odbieranie komunikatów SYN i ACK itd.

    ChangeState(t, TCPEstablished::Instance());
}

```

Po wykonaniu zadań specyficznych dla stanu operacje te wywołują operację ChangeState, aby zmienić stan obiektu TCPConnection. Sam ten obiekt nie zna protokołu połączeń TCP. To w podkласach klasy TCPState zdefiniowane są zmiany stanu i działania tego protokołu.

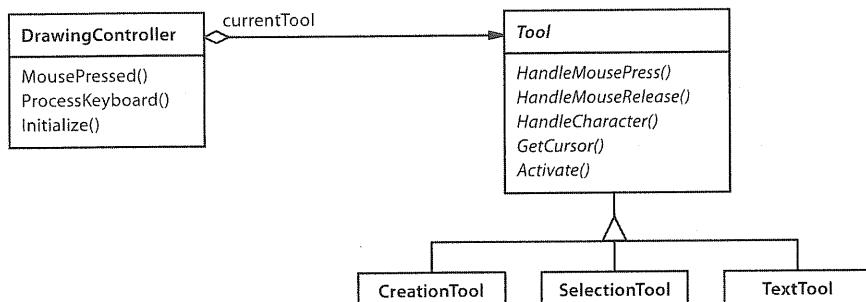
ZNANE ZASTOSOWANIA

Johnson i Zweig [JZ91] opisują wzorzec Stan oraz jego zastosowanie w protokołach połączeń TCP.

Większość popularnych interaktywnych programów graficznych udostępnia narzędzia do wykonywania operacji przez bezpośrednie manipulowanie rysunkiem. Na przykład narzędzie do rysowania linii umożliwia użytkownikowi kliknięcie i przeciągnięcie punktu w celu utworzenia nowej linii. Mechanizm zaznaczania pozwala zaznaczać kształty. Zwykle istnieje paleta dostępnych narzędzi do wyboru. Użytkownik może uważać, że wybiera narzędzie i korzysta z niego, jednak w rzeczywistości edytor zmienia zachowanie na podstawie bieżącego narzędzia. Kiedy aktywne jest narzędzie do rysowania, można tworzyć kształty, jeżeli używane jest narzędzie do zaznaczania, można zaznaczać elementy itd. Wzorzec Stan pozwala zmieniać zachowanie edytora w zależności od wybranego narzędzia.

Możemy zdefiniować klasę abstrakcyjną `Tool` i utworzyć jej podklasy z implementacją zachowania specyficznego dla narzędzia. Edytor graficzny przechowuje bieżący obiekt `Tool` i deleguje do niego żądania. Kiedy użytkownik wybierze nowe narzędzie, program zastąpi nim wspomniany obiekt, co spowoduje odpowiednią zmianę zachowania edytora.

Tę technikę zastosowano w dwóch platformach do tworzenia edytorów graficznych — HotDraw [Joh92] i Unidraw [VL90]. Umożliwiają one łatwe definiowanie w klientach nowych rodzajów narzędzi. W platformie HotDraw klasa `DrawingController` przekazuje żądania do bieżącego obiektu `Tool`. W platformie Unidraw odpowiedniki tych klas to `Viewer` i `Tool`. Poniższy diagram klas przedstawia fragment interfejsów klas `Tool` i `DrawingController`:



Ze wzorcem Stan powiązany jest idiom list-koperta opisany przez Copliena [Cop92]. List-koperta to technika do zmiany klasy obiektu w czasie wykonywania programu. Wzorzec Stan jest bardziej specyficzny i dotyczy przede wszystkim tego, jak zarządzać obiektem, którego zachowanie zależy od stanu.

POWIĄZANE WZORCE

Wzorzec Pyłek (s. 201) określa, kiedy i jak można współużytkować obiekty State.

Obiekty State często są singletonami (s. 130).



STRATEGIA (STRATEGY)

obiektowy, operacyjny

PRZEZNACZENIE

Określa rodzinę algorytmów, kapsuluje każdy z nich i umożliwia ich zamienne stosowanie. Wzorzec ten pozwala zmieniać algorytmy niezależnie od korzystających z nich klientów.

INNA NAZWA

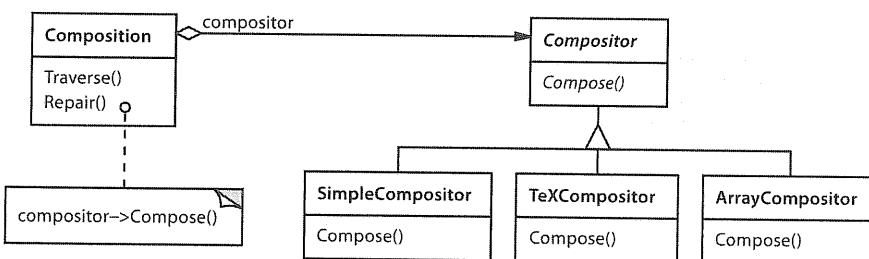
Polityka (ang. *policy*).

UZASADNIENIE

Istnieje wiele algorytmów podziału strumienia tekstu na wiersze. Trwałe wbudowanie wszystkich takich algorytmów w potrzebujące ich klasy nie jest pożądane. Wynika to z kilku powodów:

- ▶ Klienci potrzebujące podziału tekstu na wiersze są bardziej złożone, jeśli obejmują kod do wykonywania tego zadania. Prowadzi to do powiększenia klientów i utrudnia ich konserwację, zwłaszcza jeśli potrzebna jest obsługa wielu algorytmów podziału tekstu na wiersze.
- ▶ W różnych miejscach odpowiednie mogą być inne algorytmy. Nie chcemy dodawać obsługi wielu algorytmów podziału na wiersze, jeśli nie będziemy z nich wszystkich korzystać.
- ▶ Trudno jest dodawać nowe algorytmy i modyfikować istniejące, jeśli kod podziału tekstu na wiersze jest integralną częścią klienta.

Można uniknąć tych problemów przez zdefiniowanie klas kapsułkujących różne algorytmy podziału tekstu na wiersze. Algorytm zakapsułkowany w ten sposób to tak zwana strategia.



Załóżmy, że klasa **Composition** odpowiada za przechowywanie i aktualizowanie miejsc podziału na wiersze tekstu wyświetlany w czytniku. Strategie podziału na wiersze nie są zaimplementowane w klasie **Composition**. Zamiast tego zaimplementowano je osobno w podklasach klasy abstrakcyjnej **Compositor**. Podklasy klasy **Compositor** obejmują implementacje różnych strategii:

- ▶ Klasa **SimpleCompositor** reprezentuje prostą strategię określającą miejsca podziału na wiersze po jednym razie.

- ▶ Klasa **TeXCompositor** obejmuje implementację algorytmu T.X służącego do wyszukiwania miejsc podziału na wiersze. Ta strategia próbuje zoptymalizować podział na wiersze globalnie (to znaczy po jednym akapicie naraz).
- ▶ Klasa **ArrayCompositor** reprezentuje strategię wybierającą miejsca podziału w taki sposób, aby każdy wiersz miał stałą liczbę elementów. Jest to przydatne na przykład do podziału kolekcji ikon na wiersze.

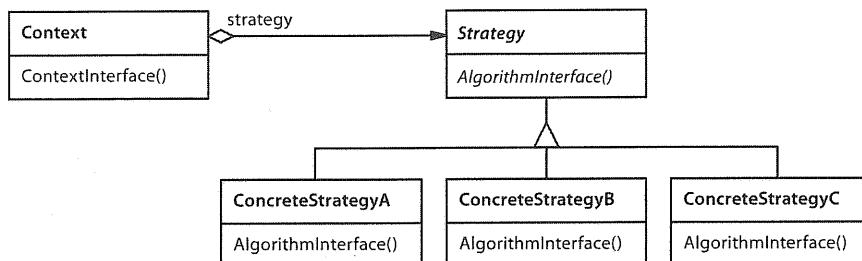
Obiekt **Composition** przechowuje referencję do obiektu **Compositor**. Kiedy obiekt **Composition** zmienia formatowanie tekstu, przekazuje to zadanie do powiązanego z nim obiektu **Compositor**. Klient obiektu **Composition** określa, który obiekt **Compositor** należy zastosować. W tym celu klient instaluje w obiekcie **Composition** pożądany obiekt **Compositor**.

WARUNKI STOSOWANIA

Wzorca Strategia należy używać w następujących warunkach:

- ▶ Kiedy wiele powiązanych klas różni się tylko zachowaniem. Strategie umożliwiają skonfigurowanie klasy za pomocą jednego z wielu zachowań.
- ▶ Jeśli potrzebne są różne wersje algorytmu. Można na przykład zdefiniować algorytmy związane z różnymi korzyściami i kosztami z zakresu pamięci oraz czasu przetwarzania. Strategię można stosować, jeśli wspomniane wersje są zaimplementowane w postaci hierarchii klas algorytmów [HO87].
- ▶ Jeżeli algorytm korzysta z danych, o których klienci nie powinny wiedzieć. Wzorzec Strategia pozwala uniknąć ujawniania złożonych, specyficznych dla algorytmu struktur danych.
- ▶ Gdy klasa definiuje wiele zachowań, a te w operacjach pojawiają się w formie złożonych instrukcji warunkowych. Zamiast tworzyć wiele takich instrukcji, należy przenieść powiązane gałęzie do odrębnych klas **Strategy**.

STRUKTURA



ELEMENTY

- ▶ **Strategy (Compositor):**
 - obejmuje deklarację wspólnego interfejsu wszystkich obsługiwanych algorytmów. Klasa **Context** może korzystać z tego interfejsu do wywoływanego algorytmów zdefiniowanych w klasach **ConcreteStrategy**.

- ▶ **ConcreteStrategy** (`SimpleCompositor`, `TeXCompositor`, `ArrayCompositor`):
 - obejmuje implementację algorytmu zgodną z interfejsem klasy `Strategy`.
- ▶ **Context** (`Composition`):
 - jest konfigurowana za pomocą obiektu `ConcreteStrategy`;
 - przechowuje referencję do obiektu `Strategy`;
 - może definiować interfejs dający obiektom `Strategy` dostęp do danych obiektu `Context`.

WSPÓŁDZIAŁANIE

- ▶ Klasy `Strategy` i `Context` współpracują w celu realizacji wybranego algorytmu. Obiekt `Context` może w momencie wywołania algorytmu przekazać do obiektu `Strategy` wszystkie dane potrzebne w tym algorytmie. Inne rozwiązanie to przekazywanie przez obiekt `Context` samego siebie jako argumentu do operacji klasy `Strategy`. Umożliwia to obiektom `Strategy` zwrotne wywołanie obiektu `Context` w razie potrzeby.
- ▶ Obiekt `Context` przekazuje żądania od jego klientów do obiektu `Strategy`. Klienci zwykle tworzą obiekt `ConcreteStrategy` i przekazują go do obiektu `Context`. Następnie klienci mogą kontaktować się tylko z obiektem `Context`. Często klienci mogą wybierać klasy z całej rodziny klas `ContextStrategy`.

KONSEKWENCJE

Oto zalety i wady związane ze wzorcem `Strategia`:

1. Powoduje powstawanie rodzin powiązanych algorytmów. Hierarchie klas `Strategy` określają rodzinę algorytmów lub zachowań przeznaczonych do wielokrotnego użytku przez obiekty `Context`. Dziedziczenie może pomóc w wyodrębnieniu wspólnych funkcji algorytmów.
2. Zapewnia alternatywę dla tworzenia podklas. Nastecną techniką umożliwiającą obsługę różnych algorytmów lub zachowań jest dziedziczenie. Można utworzyć bezpośrednią podkласę klas `Context`, aby przypisać do niej inne zachowania. Jednak powoduje to trwałe powiązanie zachowania z klasą `Context`. Implementacja algorytmu jest wtedy połączona z klasą `Context`, co utrudnia jej zrozumienie, konserwowanie i rozszerzanie. Ponadto nie można dynamicznie modyfikować algorytmu. Powstaje wiele powiązanych klas różniących się tylko algorymem lub zachowaniem. Zakapsułkowanie algorytmu w odrębnych klasach `Strategy` umożliwia zmianianie go niezależnie od klasy `Context`. Ułatwia to podmianianie algorytmów, ich zrozumienie i rozszerzanie.
3. Strategie pozwalają wyeliminować instrukcje warunkowe. Wzorzec `Strategia` można zastosować zamiast instrukcji warunkowych do wyboru pożądanego zachowania. Kiedy różne zachowania są umieszczone w jednej klasie, trudno jest uniknąć korzystania z instrukcji warunkowych do ustalenia właściwego z nich. Zakapsułkowanie zachowania w odrębnych klasach `Strategy` powoduje wyeliminowanie instrukcji warunkowych.

Bez strategii kod do podziału tekstu na wiersze mógłby wyglądać tak:

```
void Composition::Repair () {
    switch (_breakingStrategy) {
        case SimpleStrategy:
            ComposeWithSimpleCompositor();
```

```

        break;
    case TexStrategy:
        ComposeWithTeXCompositor();
        break;
    // ...
}
// Scalanie wyników z istniejącym fragmentem (jeśli to konieczne).
}

```

Wzorzec Strategia pozwala wyeliminować instrukcję `case` przez oddelegowanie zadania podziału tekstu na wiersze do obiektu `Strategy`:

```

void Composition::Repair () {
    _compositor->Compose();
    // Scalanie wyników z istniejącym fragmentem (jeśli to konieczne).
}

```

Kod obejmujący wiele instrukcji warunkowych często wskazuje na konieczność zastosowania wzorca `Strategia`.

4. *Umożliwia wybór implementacji.* Strategie mogą udostępniać różne implementacje tego samego zachowania. Klient może wybierać strategie pod kątem różnych korzyści i kosztów związanych z czasem przetwarzania oraz pamięcią.
5. *Klienci muszą znać różne strategie.* Omawiany wzorzec ma potencjalną wadę — klient musi znać różnice między obiektami `Strategy`, zanim wybierze odpowiedni z nich. Klienci mogą być przez to narażone na problemy implementacyjne. Dlatego wzorzec `Strategia` należy stosować tylko wtedy, jeśli zmiana zachowania ma znaczenie dla klientów.
6. *Powoduje koszty komunikacji między obiektami Strategy i Context.* Interfejs klasy `Strategy` jest współużytkowany przez wszystkie klasy `ConcreteStrategy` niezależnie od tego, czy zaimplementowane w nich algorytmy są proste czy złożone. Dlatego prawdopodobne jest, że niektóre obiekty `ConcreteStrategy` nie będą korzystać z wszystkich informacji przekazanych do nich poprzez wspomniany interfejs. Proste obiekty `ConcreteStrategy` mogą w ogóle nie używać takich danych! Oznacza to, że obiekt `Context` czasem tworzy i inicjuje niepotrzebne parametry. Jeśli powoduje to problemy, trzeba zwiększyć powiązanie między klasami `Strategy` i `Context`.
7. *Powoduje zwiększenie liczby obiektów.* Stosowanie strategii powoduje zwiększenie liczby obiektów w aplikacji. Czasem można zmniejszyć związane z tym koszty przez zaimplementowanie strategii jako obiektów bezstanowych, które obiekty `Context` mogą współużytkować. Stan jest wtedy przechowywany przez obiekty `Context`, które przekazują go w każdym żądaniu kierowanym do obiektu `Strategy`. Strategie współużytkowane nie powinny przechowywać stanu pomiędzy wywołaniami. Wzorzec Pyłek (s. 201) ilustruje to podejście bardziej szczegółowo.

IMPLEMENTACJA

Rozważmy następujące kwestie implementacyjne:

1. *Definiowanie interfejsów klas Strategy i Context.* Interfejsy klas `Strategy` i `Context` muszą zapewniać obiektom `ConcreteStrategy` wydajny dostęp do wszelkich potrzebnych im danych z obiektu `Context` i na odwrót.

Jedną z możliwości jest przekazywanie przez obiekt Context danych w parametrach operacji klasy *Strategy* (czyli przenoszenie danych do strategii). Pozwala to uniknąć powiązania między klasami *Strategy* i *Context*. Jednak obiekt *Context* może wtedy przekazywać dane niepotrzebne obiekowi *Strategy*.

Inna technika polega na przekazywaniu przez obiekt *Context* jako argumentu *samego siebie*, a obiekt *Strategy* jawnie żąda wtedy od niego danych. Obiekt *Strategy* może też przechowywać referencję do powiązanego z nim obiektu *Context*, co eliminuje konieczność przekazywania jakichkolwiek informacji. W obu podejściach obiekt *Strategy* może zażądać dokładnie tego, czego potrzebuje. Jednak w klasie *Context* trzeba zdefiniować bardziej rozbudowany interfejs do udostępniania danych, co zwiększa powiązanie między klasami *Strategy* i *Context*.

Wymagania związane z określonym algorytmem i jego danymi wyznaczają, która z technik będzie najlepsza.

2. *Strategie jako parametry szablonu.* W języku C++ można wykorzystać szablon do skonfigurowania klasy za pomocą strategii. Tę technikę można stosować tylko wtedy, jeśli (1) obiekt *Strategy* można wybrać w czasie komplikacji i (2) nie trzeba go zmieniać w czasie wykonywania programu. Wtedy konfigurowaną klasę (na przykład *Context*) należy zdefiniować jako klasę szablonową z parametrem w postaci klasy *Strategy*:

```
template <class AStrategy>
class Context {
    void Operation() { theStrategy.DoAlgorithm(); }
    // ...
private:
    AStrategy theStrategy;
};
```

Następnie klasę szablonową należy w momencie tworzenia jej egzemplarza skonfigurować za pomocą klasy *Strategy*:

```
class MyStrategy {
public:
    void DoAlgorithm();
};

Context<MyStrategy> aContext;
```

Przy korzystaniu z szablonów nie trzeba definiować klasy abstrakcyjnej określającej interfejs obiektów *Strategy*. Zastosowanie obiektów *Strategy* jako parametru szablonu pozwala ponadto statycznie powiązać obiekt *Strategy* i odpowiadający mu obiekt *Context*, co może zwiększyć wydajność programu.

3. *Opcjonalne stosowanie obiektów Strategy.* Klasę *Context* można uprościć, jeśli może ona działać bez obiektu *Strategy*. Obiekt *Context* powinien wtedy sprawdzać, czy obiekt *Strategy* jest dostępny, a dopiero potem z niego korzystać. Jeżeli obiekt *Strategy* istnieje, obiekt *Context* będzie stosował go w standardowy sposób. Jeżeli strategia jest niedostępna, obiekt *Context* wykorzysta standardowe zachowanie. Zaletą tego podejścia jest to, iż klienci w ogóle nie muszą stosować obiektów *Strategy*, chyba że nie odpowiada im zachowanie domyślne.

PRZYKŁADOWY KOD

Przedstawiamy tu wysokopoziomowy kod przykładu z punktu „Uzasadnienie”. Kod ten oparliśmy na implementacji klas **Composition** i **Compositor** z pakietu InterViews [LCI-92].

Klasa **Composition** przechowuje kolekcję egzemplarzy klasy **Component** reprezentujących elementy tekstowe i graficzne dokumentu. Obiekt **Composition** porządkuje obiekty **Component** w wiersze za pomocą egzemplarza podklasy klasy **Compositor** kapsułkującego strategię podziału tekstu na wiersze. Każdy obiekt **Component** ma naturalny rozmiar oraz zakres zwiększenia i zmniejszania. Zakres zwiększenia określa, w jakim stopniu można powiększyć dany komponent ponad jego naturalną wielkość. Zakres zmniejszania wyznacza, jak bardzo można pomniejszyć element. Obiekt **Composition** przekazuje te wartości do obiektu **Compositor**, który stosuje je do ustalenia najlepszych miejsc podziału tekstu na wiersze.

```
class Composition {
public:
    Composition(Compositor*);
    void Repair();
private:
    Compositor* _compositor;
    Component* _components; // Lista komponentów.
    int _componentCount; // Liczba komponentów.
    int _lineWidth; // Szerokość wiersza obiektu Composition.
    int* _lineBreaks; // Pozycje miejsc podziału wierszy
                     // w komponentach.
    int _lineCount; // Liczba wierszy.
};
```

Kiedy trzeba określić nowy układ tekstu, obiekt **Composition** żąda od obiektu **Compositor** określenia miejsc podziału wierszy. Obiekt **Composition** przekazuje do obiektu **Compositor** trzy tablice definiujące naturalne rozmiary komponentów oraz zakres ich zwiększenia i zmniejszania. Przekazuje też liczbę komponentów, szerokość wiersza i tablicę zapelnianą przez obiekt **Compositor** pozycjami każdego miejsca podziału wierszy. Projekt **Compositor** zwraca liczbę określonych miejsc podziału.

Interfejs klasy **Compositor** umożliwia obiektom **Composition** przekazanie do obiektu **Compositor** wszystkich potrzebnych informacji. Jest to przykład zastosowania podejścia „przenoszenie danych do strategii”:

```
class Compositor {
public:
    virtual int Compose(
        Coord natural[], Coord stretch[], Coord shrink[],
        int componentCount, int lineWidth, int breaks[]
    ) = 0;
protected:
    Compositor();
};
```

Warto zauważyć, że **Compositor** to klasa abstrakcyjna. Jej podklasy konkretne definiują określone strategie podziału tekstu na wiersze.

Obiekt **Composition** wywołuje powiązany z nim obiekt **Compositor** w operacji **Repair**. Operacja ta najpierw inicjuje tablice za pomocą naturalnego rozmiaru oraz zakresu zwiększenia i zmniejszenia każdego komponentu (aby zachować zwieńłość, szczegóły pomijamy). Następnie wywołuje operację każdego obiektu **Compositor**, aby ustalić miejsca podziału wierszy, i na zakończenie rozmieszcza komponenty na podstawie tych punktów (ten fragment także pomijamy):

```
void Composition::Repair () {
    Coord* natural;
    Coord* stretchability;
    Coord* shrinkability;
    int componentCount;
    int* breaks;

    // Przygotowanie tablic z pożądanym rozmiarem komponentów.
    // ...

    // Określanie miejsc podziału wierszy:
    int breakCount;
    breakCount = _compositor->Compose(
        natural, stretchability, shrinkability,
        componentCount, _lineWidth, breaks
    );
    // Rozmieszczenie komponentów zgodnie z punktami podziału wierszy.
    // ...
}
```

Przyjrzyjmy się teraz podklasom klasy **Compositor**. Podklasa **SimpleCompositor** bada komponenty po jednym wierszu naraz, aby ustalić, gdzie umieścić punkty podziału wierszy:

```
class SimpleCompositor : public Compositor {
public:
    SimpleCompositor ();
    virtual int Compose(
        Coord natural[], Coord stretch[], Coord shrink[],
        int componentCount, int lineWidth, int breaks[]
    );
    // ...
};
```

W klasie **TeXCompositor** zastosowano bardziej globalną strategię. Bada ona cały akapit i uwzględnia rozmiar komponentów oraz zakres ich zwiększenia. Próbuje też zapewnić równe „światło” akapitu przez zminimalizowanie odstępów między komponentami.

```
class TeXCompositor : public Compositor {
public:
    TeXCompositor ();
    virtual int Compose(
        Coord natural[], Coord stretch[], Coord shrink[],
        int componentCount, int lineWidth, int breaks[]
    );
    // ...
};
```

Klasa `ArrayCompositor` dzieli komponenty na wiersze w regularnych odstępach.

```
class ArrayCompositor : public Compositor {
public:
    ArrayCompositor(int interval);

    virtual int Compose(
        Coord natural[], Coord stretch[], Coord shrink[],
        int componentCount, int lineWidth, int breaks[]
    );
    // ...
};
```

Te klasy nie korzystają z wszystkich informacji przekazanych do operacji `Compose`. Klasa `SimpleCompositor` ignoruje zakres zwiększenia komponentów i uwzględnia tylko ich naturalną szerokość. Klasa `TeXCompositor` korzysta z wszystkich otrzymanych informacji, natomiast klasa `ArrayCompositor` wszystkie je pomija.

Aby utworzyć egzemplarz klasy `Composition`, należy przekazać do niego obiekt `Compositor`, którego egzemplarz ten ma używać:

```
Composition* quick = new Composition(new SimpleCompositor);
Composition* slick = new Composition(new TeXCompositor);
Composition* iconic = new Composition(new ArrayCompositor(100));
```

Interfejs klasy `Compositor` jest starannie zaprojektowany, tak aby obsługiwał wszystkie algorytmy określania układu, które można zastosować w podklasach. Niepożądane jest zmienianie tego interfejsu przy tworzeniu każdej nowej podklasty, ponieważ będzie to wymagać zmodyfikowania istniejących podklaśc. Ogólnie interfejsy klas `Strategy` i `Context` określają, jak dobrze wzorzec spełnia swoje przeznaczenie.

ZNANE ZASTOSOWANIA

W platformie ET++ [WGM88] i pakiecie InterViews strategie wykorzystano w opisany tu sposób do zakapsułkowania różnych algorytmów podziału tekstu na wiersze.

W platformie RTL System [JML92] (służy ona do optymalizacji komplikowanego kodu) strategie określają różne sposoby przydzielania rejestrów (klasy `RegisterAllocator`) i zasady szeregowania zestawów instrukcji (klasy `RISCscheduler` i `CISCscheduler`). Zapewnia to elastyczność przy dostosowywaniu optymalizatora do maszyn o różnych architekturach.

Platforma ET++SwapsManager (służy ona do tworzenia silników obliczeniowych) oblicza ceny różnych instrumentów finansowych [EG92]. Kluczowe abstrakcje w tej platformie to klasy `Instrument` i `YieldCurve`. Poszczególne instrumenty finansowe są zaimplementowane jako podklasy klasy `Instrument`. Klasa `YieldCurve` służy do obliczania współczynnika dyskonta, który określa obecną wartość przyszłych przepływów pieniężnych. Obie te klasy delegują niektóre zachowania do obiektów `Strategy`. Platforma udostępnia rodzinę klas `ConcreteStrategy` do generowania przepływów pieniężnych oraz obliczania swapów i współczynników dyskonta. Nowe silniki obliczeniowe można tworzyć przez konfigurowanie klas `Instrument` i `YieldCurve` za pomocą różnych obiektów `ConcreteStrategy`. To podejście umożliwia łączenie i dopasowywanie istniejących implementacji klas `Strategy`, a także definiowanie nowych.

W komponentach Boocha [BV90] strategie zastosowano jako argumenty szablonu. Klasa reprezentująca kolekcje obsługują trzy różne strategie przydziału pamięci: zarządzaną (przydział z puli), kontrolowaną (operacja przydziału i usuwania są chronione za pomocą blokady) i niezarządzaną (normalny mechanizm przydziału pamięci). Te strategie są przekazywane jako argumenty szablonu do klasy kolekcji przy tworzeniu jej egzemplarza. Na przykład egzemplarz klasy `UnboundedCollection` stosujący strategię niezarządzaną jest tworzony za pomocą wywołania `UnboundedCollection<MyItemType*, Unmanaged>`.

RApp to system do projektowania układów scalonych [GA89, AG90]. RApp musi określić rozmieszczenie i przebieg przewodów łączących podsystemy układu. Algorytmy wyznaczania przebiegu przewodów są w systemie RApp zdefiniowane jako podklasy klasy abstrakcyjnej `Router`. Router to odpowiednik klasy `Strategy`.

W bibliotece ObjectWindows [Bor94] firmy Borland wykorzystano strategie w oknach dialogowych, aby zagwarantować, że użytkownik wpisuje prawidłowe dane. Można na przykład zagwarantować, że liczby mieszczą się w określonym przedziale, a pole na dane liczbowe obejmuje tylko cyfry. Sprawdzenie, czy łańcuch znaków jest poprawny, może wymagać sprawdzenia zawartości tablicy.

W bibliotece ObjectWindows do kapsułkowania strategii sprawdzania poprawności służą obiekty `Validator`. Są one odpowiednikiem obiektów `Strategy`. Pola do wprowadzania danych delegują realizację strategii sprawdzania poprawności do opcjonalnego obiektu `Validator`. Klient dołącza taki obiekt do pola, jeśli sprawdzanie poprawności jest potrzebne (jest to przykład działania strategii opcjonalnej). Omawiana biblioteka klas udostępnia validatory dla standardowych przypadków, na przykład klasę `RangeValidator` do sprawdzania liczb. Nowe, specyficzne dla klienta strategie sprawdzania poprawności można łatwo zdefiniować przez utworzenie podklasy klasy `Validator`.

POWIĄZANE WZORCE

Pyłek (s. 201): obiekty `Strategy` często warto tworzyć jako pyłki.

OMÓWIENIE WZORCÓW OPERACYJNYCH

KAPSUŁKOWANIE ZMIAN

Kapsułkowanie zmian to motyw powtarzający się w wielu wzorcach operacyjnych. Jeśli pewien aspekt programu często się zmienia, wzorce z tej grupy pozwalają zdefiniować obiekt kapsułkujący dany aspekt. Następnie pozostałe części programu mogą współdziałać z tym obiektem, kiedy potrzebują określonego aspektu. Wzorce operacyjne zwykle wymagają zdefiniowania klasy abstrakcyjnej opisującej obiekty użyte do kapsułkowania, a poszczególne wzorce mają nazwy związane z takimi obiektami¹³. Na przykład:

- ▶ obiekt **Strategy** kapsułkuje algorytm (wzorzec Strategia, s. 321);
- ▶ obiekt **State** kapsułkuje zachowanie zależne od stanu (wzorzec Stan, s. 312);
- ▶ obiekt **Mediator** kapsułkuje protokół komunikacji między obiektami (wzorzec Mediator, s. 254);
- ▶ obiekt **Iterator** kapsułkuje sposób dostępu do komponentów obiektu zagregowanego i przechodzenia po nich (wzorzec Iterator, s. 230).

Wzorce operacyjne opisują aspekty programu, które prawdopodobnie będą się zmieniać. W większości wzorców występują obiekty dwóch rodzajów — nowe obiekty kapsułkujące dany aspekt i istniejące obiekty korzystające z nowych. Zwykle funkcje nowych obiektów byłyby integralną częścią istniejących obiektów, gdyby nie zastosowano danego wzorca. Na przykład kod klasy **Strategy** prawdopodobnie znalazłby się w powiązanej z nią klasie **Context**, a kod klasy **State** — bezpośrednio w klasie **Context** powiązanej ze stanem.

Jednak nie we wszystkich obiektowych wzorcach operacyjnych funkcje są podzielone w ten sposób. Na przykład wzorzec Łąncuch zobowiązań (s. 244) dotyczy dowolnej liczby obiektów (czyli łańcucha), a wszystkie one mogą już istnieć w systemie.

Łąncuch zobowiązań ilustruje też następną różnicę między poszczególnymi wzorcami operacyjnymi — nie wszystkie one określają statyczne relacje związane z komunikacją między klasami. Wzorzec Łąncuch zobowiązań opisuje komunikację między dowolną liczbą obiektów. Inne wzorce dotyczą obiektów przekazywanych jako argumenty.

OBIEKTY JAKO ARGUMENTY

W kilku wzorcach wprowadzono obiekt *zawsze* używany jako argument. Jednym z takich wzorców jest Odwiedzający (s. 280). Obiekt **Visitor** to argument polimorficznej operacji **Accept** odwiedzanych obiektów. Odwiedzający nigdy nie jest uważany za część takich obiektów, choć standardową alternatywą dla tego wzorca jest umieszczenie kodu **Visitor** w różnych klasach struktury obiektów.

¹³To podejście pojawia się też we wzorcach innego rodzaju. Wzorce Fabryka abstrakcyjna (s. 87), Budowniczy (s. 97) i Prototyp (s. 117) polegają na kapsułkowaniu sposobu tworzenia obiektów. Wzorzec Dekorator (s. 175) dotyczy kapsułkowania zadań dodawanych do obiektu. Wzorzec Most (s. 151) polega na oddzieleniu abstrakcji od implementacji, co umożliwia modyfikowanie ich niezależnie od siebie.

Inne wzorce określają obiekty działające jak „magiczne żetony”. Są one przekazywane między innymi obiektami i wywoływanie w późniejszym czasie. Do tej kategorii należą wzorce Polecenie (s. 302) i Pamiątka (s. 294). We wzorcu Polecenie żeton służy do reprezentowania ządania, a we wzorcu Pamiątka — do reprezentowania wewnętrznego stanu obiektu w określonym momencie. W obu przypadkach żeton może mieć złożoną reprezentację wewnętrzną, ale klienci jej nie znają. Jednak nawet tu występują różnice. We wzorcu Polecenie polimorfizm ma znaczenie, ponieważ uruchamianie kodu z obiektu Command to operacja polimorficzna. Z drugiej strony interfejs klasy Memento jest tak zawężony, że pamiątkę można przekazać tylko przez wartość. Dlatego prawdopodobnie nie będzie ona w ogóle udostępniać klientom operacji polimorficznych.

CZY KOMUNIKACJA POWINNA BYĆ ZAKAPSUŁKOWANA CZY ROZPROSZONA?

Mediator (s. 254) i Obserwator (s. 269) to „konkurencyjne” wzorce. Różnica między nimi polega na tym, że wzorzec Obserwator dotyczy rozdzielenia procesu komunikacji przez wprowadzenie obserwatora i podmiotu, natomiast wzorzec Mediator dotyczy kapsułkowania komunikacji między innymi obiektami.

We wzorcu Obserwator nie występuje pojedynczy obiekt kapsułkujący ograniczenie. Zamiast tego obserwator i podmiot muszą współdziałać ze sobą w celu utrzymania tego ograniczenia. Wzorce komunikacji są określone przez sposób powiązania między obserwatorami i podmiotami. Pojedynczy podmiot zwykle ma wielu obserwatorów, a obserwator jednego podmiotu jest też czasem podmiotem innego obserwatora. Wzorzec Mediator polega bardziej na centralizowaniu niż rozpraszaniu zadań. W tym wzorcu obowiązek utrzymania ograniczenia spoczywa na mediatorze.

Zauważliśmy, że wielokrotne wykorzystanie rozwiązania jest łatwiejsze przy stosowaniu obserwatorów i podmiotów niż mediatorów. Wzorzec Obserwator ułatwia podział zadań i zachowanie luźnego powiązania między obserwatorem i podmiotem, co prowadzi do tworzenia bardziej szczegółowych klas lepiej nadających się do wielokrotnego użytku.

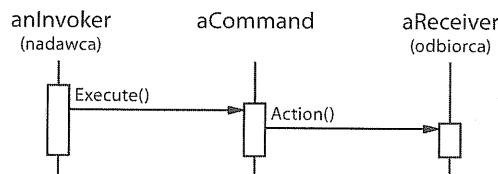
Z drugiej strony przepływ komunikacji łatwiej jest zrozumieć we wzorcu Mediator niż we wzorcu Obserwator. Obserwatory i podmioty są zwykle łączone wkrótce po ich utworzeniu, dlatego w dalszych częściach programu trudno jest ustalić rodzaj powiązania między nimi. Jeśli znasz wzorzec Obserwator, rozumiesz, że sposób połączenia między obserwatorami i podmiotami ma znaczenie, a także wiesz, na jakie powiązania zwracać uwagę. Jednak pośredniość wprowadzana przez wzorzec Obserwator utrudnia zrozumienie systemu.

W języku Smalltalk można sparametryzować obserwatory za pomocą komunikatów dostępu do stanu podmiotu. Dzięki temu możliwość wielokrotnego wykorzystania elementów tego wzorca jest jeszcze większa niż w języku C++. Dlatego programiści języka Smalltalk często stosują wzorzec Obserwator tam, gdzie programista języka C++ użyłby wzorca Mediator.

ODDZIELANIE NADAWCÓW OD ODBIORCÓW

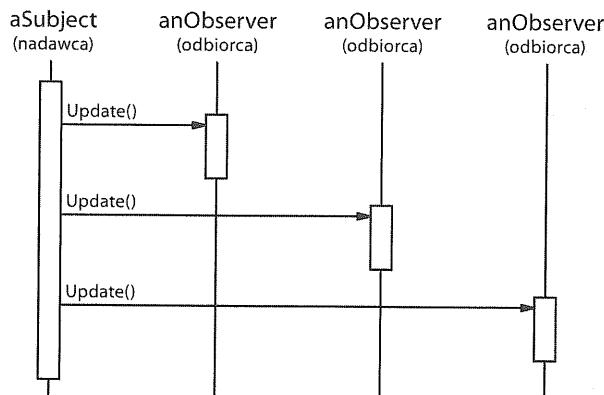
Jeśli współdziałające obiekty odwołują się bezpośrednio do siebie, stają się od siebie zależne. Może to mieć negatywny wpływ na podział na warstwy i możliwość wielokrotnego wykorzystania systemu. Wzorce Polecenie, Obserwator, Mediator i Łańcuch zobowiązają dotyczą rozdzielania nadawców od odbiorców, jednak mają różne zalety oraz wady.

We wzorcu Polecenie do rozdzielania służy obiekt polecenia. Określa on powiązanie między nadawcą i odbiorcą:



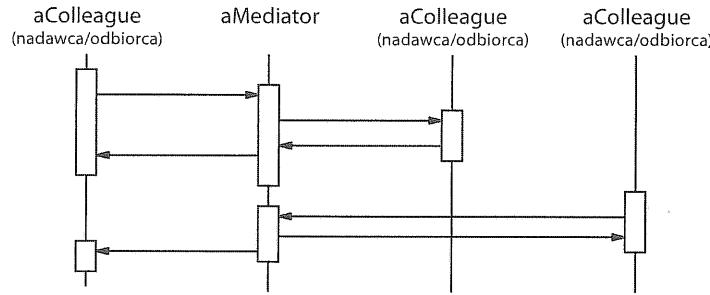
Obiekt polecenia udostępnia prosty interfejs do zgłoszenia żądania (czyli operację Execute). Zdefiniowanie połączenia między nadawcą i odbiorcą w odrębnym obiekcie umożliwia nadawcy współdziałanie z różnymi odbiorcami. Pozwala to oddzielić nadawcę od odbiorców i ułatwia jego wielokrotne wykorzystanie. Ponadto można powtórnie użyć obiektu polecenia do spараметyzowania odbiorcy za pomocą różnych nadawców. Wzorzec Polecenie teoretycznie wymaga utworzenia podklasy dla każdego połączenia między nadawcą i odbiorcą, jednak opisane są techniki implementacyjne, które pozwalają uniknąć przygotowywania podklaśs.

Wzorzec Obserwator umożliwia rozdzielenie nadawców (podmioty) od odbiorców (obserwatory) przez zdefiniowanie interfejsu do sygnalizowania zmian w podmiotach. We wzorcu Obserwator powiązanie między nadawcą i odbiorcą jest luźniejsze niż we wzorcu Polecenie, ponieważ podmiot może mieć wielu obserwatorów, a ich liczba może się zmieniać w czasie wykonywania programu.



Interfejsy podmiotu i obserwatora we wzorcu Obserwator są zaprojektowane do przekazywania informacji o zmianach. Dlatego wzorzec ten najlepiej nadaje się dla rozdzielania obiektów, jeśli występują między nimi zależności w zakresie danych.

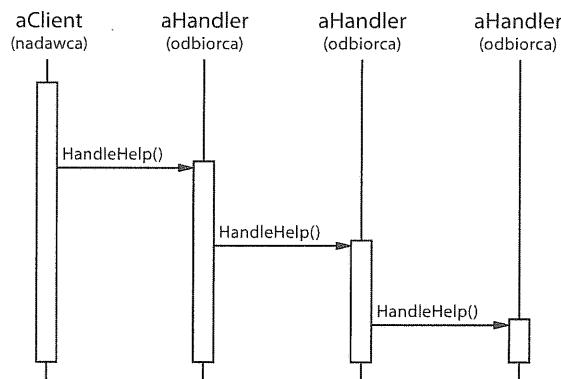
Aby rozdzielić obiekty za pomocą wzorca Mediator, należy sprawić, aby odwoływały się do siebie pośrednio — poprzez obiekt mediatora.



Obiekt aMediator przekazuje żądania między obiektami aColleague i centralizuje komunikację między nimi. Powoduje to, że obiekty aColleague mogą kontaktować się ze sobą tylko za pośrednictwem interfejsu mediatora. Ponieważ ten interfejs jest stały, w mediatorze trzeba czasem zaimplementować specjalny system rozdzielania żądań, aby zwiększyć elastyczność rozwiązania. Można zakodować żądania i zapisać argumenty w taki sposób, aby obiekty współpracujące mogły zażądać dowolnego zbioru operacji.

Wzorzec Mediator umożliwia zmniejszenie liczby podklas w systemie, ponieważ pozwala scentralizować komunikację w jednej klasie, zamiast rozmieszczać ją w wielu podklasach. Jednak przygotowane doraźnie systemy rozdzielania żądań często zmniejszają bezpieczeństwo ze względu na typ.

Wzorzec Łańcuch zobowiązań umożliwia rozdzielenie nadawcy od odbiorcy przez przekazywanie żądań wzdułż łańcucha potencjalnych odbiorców:



Ponieważ interfejs między nadawcami i odbiorcami jest stały, wzorzec Łańcuch zobowiązań również może wymagać niestandardowego systemu rozdzielania żądań. Dlatego występują tu te same wady związane z bezpieczeństwem ze względu na typ co we wzorcu Mediator. Łańcuch zobowiązań to dobry sposób na rozdzielenie nadawcy od odbiorcy, jeśli łańcuch jest już częścią struktury systemu, a do obsługi żądania może nadawać się jeden z kilku obiektów. Ponadto wzorzec ten zapewnia dodatkową elastyczność, ponieważ łańcuch można łatwo modyfikować i rozszerzać.

PODSUMOWANIE

Oprócz kilku wyjątków operacyjne wzorce projektowe uzupełniają i wspomagają się nawzajem. Na przykład klasa w łańcuchu zobowiązań będzie obejmować przynajmniej jedną metodę szablonową (s. 264). W takiej metodzie można zastosować operacje proste do określania, czy obiekt powinien obsługiwać żądanie, a także do wyboru obiektu, do którego należy je przekazać. W łańcuchu można wykorzystać wzorzec Polecenie do reprezentowania żądań jako obiektów. We wzorcu Interpreter (s. 217) można użyć wzorca Stan do zdefiniowania kontekstów analizy. Iterator może przechodzić po zawartości agregatu, a odwiedzający — uruchomić operację dla każdego elementu takiej struktury.

Wzorce operacyjne dobrze współdziałały także z innymi wzorcami. Na przykład w systemie, w którym wykorzystano wzorzec Kompozyt (s. 170), można użyć odwiedzającego do wykonywania operacji na komponentach złożenia. Można też zastosować łańcuch zobowiązań, aby zapewnić komponentom dostęp do właściwości globalnych za pośrednictwem elementów nadzędnych. Ponadto można użyć wzorca Dekorator (s. 152) do przesłonięcia właściwości w częściach złożenia. Można wykorzystać wzorzec Obserwator do powiązania jednej struktury obiektów z inną i wzorzec Stan, aby umożliwić zmianianie zachowania komponentów wraz ze zmianą stanu. Samo złożenie można utworzyć przez zastosowanie podejścia stosowanego we wzorcu Budowniczy (s. 92). Niektóre z innych części systemu mogą traktować złożenie jak prototyp (s. 120).

Dobrze zaprojektowane systemy obiektowe wyglądają właśnie tak — obejmują wiele wzorców. Jednak nie zawsze projektanci świadomie stosują te wzorce. Budowanie systemów na poziomie wzorców, a nie klas lub obiektów, pozwala łatwiej uzyskać ten sam efekt synergii.

Rozdział 6.

Podsumowanie

Niektórzy mogą twierdzić, że książka ta nie jest istotnym osiągnięciem. W końcu nie przedstawiliśmy tu żadnych algorytmów ani technik programistycznych, których nie stosowano wcześniej. Nie podaliśmy ścisłej metody projektowania systemów ani nie opracowaliśmy nowej teorii projektowania, a jedynie udokumentowaliśmy istniejące projekty. Można dojść do wniosku, że być może jest to przydatny samouczek, ale z pewnością nie oferuje zbyt wiele doświadczenemu projektantowi obiektowemu.

Mamy nadzieję, że masz inne zdanie. Katalogowanie wzorców projektowych jest ważne. Zapewnia standardowe nazwy i definicje używanych technik. Jeśli nie będziemy badać wzorców projektowych występujących w oprogramowaniu, nie będziemy mogli ich ulepszać i trudniej będzie opracować nowe wzorce.

Ta książka to dopiero początek. Obejmuje niektóre z najczęściej spotykanych wzorców projektowych stosowanych przez doświadczonych projektantów obiektowych, jednak programiści dowiadują się o nich i poznają je wyłącznie poprzez rozmowy lub analizę istniejących systemów. Pierwsze wersje tej książki zachęciły inne osoby do zapisania używanych przez nie wzorców projektowych, a w obecnej formie pozycja ta powinna skłonić do tego jeszcze większą grupę programistów. Mamy nadzieję, że będzie to oznaczać początek ruchu związanego z dokumentowaniem wiedzy eksperckiej praktyków zajmujących się rozwijaniem oprogramowania.

W tym rozdziale opisujemy wpływ, jaki naszym zdaniem wywrą wzorce projektowe, i omawiamy ich związki z innymi zadaniami w procesie projektowania. Wyjaśniamy też, jak samodzielnie zaangażować się w wyszukiwanie i katalogowanie wzorców.

6.1. CZEGO MOŻNA OCZEKIWAĆ OD WZORCÓW PROJEKTOWYCH?

Oto kilka sposobów, w jakie wzorce projektowe z tej książki mogą wpływać na to, jak projektujesz oprogramowanie obiektowe. Rozważania te opieramy na naszych codziennych doświadczeniach w korzystaniu z tych wzorców.

STANDARDOWE SŁOWNICTWO PROJEKTOWE

W badaniach nad korzystaniem ze standardowych języków programowania przez ekspertów wykazano, że wiedza i doświadczenie nie są uporządkowane według składni, ale na podstawie większych jednostek, takich jak algorytmy, struktury danych i idiomy [AS85, Cop92, Cur89, SS86] oraz plany realizacji określonych celów [SE84]. Projektanci prawdopodobnie mniej myślą o notacji używanej do zapisania projektu, a więcej o tym, jak połączyć aktualną sytuację projektową z poznanymi wcześniej planami, algorytmami, strukturami danych i idiomami.

Naukowcy zajmujący się informatyką nazywają i katalogują algorytmy oraz struktury danych, jednak często nie robią tego w stosunku do wzorców innego rodzaju. Wzorce projektowe zapewniają standardowe słownictwo, z którego projektanci mogą korzystać przy przekazywaniu, dokumentowaniu i badaniu różnych możliwości projektowych. Takie wzorce sprawiają, że system wygląda na mniej złożony, ponieważ umożliwiają omawianie go na wyższym poziomie abstrakcji niż przy opisie za pomocą notacji projektowej lub języka programowania. Wzorce projektowe podnoszą poziom, na którym możesz projektować i analizować projekty ze współpracownikami.

Kiedy przyswoisz sobie wzorce projektowe opisane w tej książce, Twoje słownictwo projektowe niemal na pewno się zmieni. Zaczniesz wyrażać się bezpośrednio w kategoriach nazw wzorców projektowych. Zauważysz, że wypowiadasz zdania w rodzaju: „Użyjmy tu obserwatora” lub „Przekształćmy te klasy w strategię”.

POMOC W DOKUMENTOWANIU I UCZENIU SIĘ

Znajomość wzorców projektowych z tej książki ułatwia zrozumienie istniejących systemów. Wzorce te występują w większości dużych systemów obiektowych. Osoby uczące się programowania obiektowego często narzekają, że w systemach, nad którymi pracują, dziedziczenie jest stosowane w skomplikowany sposób, a ponadto trudno jest śledzić przepływ sterowania. W dużej części wynika to z braku znajomości wzorców projektowych wykorzystanych w systemie. Opanowanie ich pomoże zrozumieć istniejące systemy obiektowe.

Poznanie tych wzorców może też sprawić, że stanieś się lepszym projektantem. Udostępniają one rozwiązania standardowych problemów. Jeśli wystarczająco długo będziesz pracował nad systemami obiektowymi, prawdopodobnie samodzielnie opanujesz te wzorce. Jednak przeczytanie książki pozwoli Ci poznać je dużo szybciej. Nauczenie się ich pomoże nowicjuszowi stosować techniki, z których korzystają eksperci.

Ponadto opisanie systemu w kategoriach użytych w nim wzorców projektowych znacznie ułatwia jego zrozumienie. W innym podejściu trzeba poddać projekt inżynierii wstępnej, aby odkryć zastosowane wzorce. Korzystanie ze wspólnego słownictwa oznacza, że nie trzeba opisywać całego wzorca projektowego. Wystarczy go nazwać i oczekiwany, że czytelnicy go znają. Osoba, która nie zna wzorców, najpierw będzie musiała je znaleźć, ale to jest łatwiejsze niż uciekanie się do inżynierii wstępnej.

Stosujemy opisane wzorce we własnych projektach i uważamy je za bezcenne. Jednak można stwierdzić, że korzystamy ze wzorców w naiwny sposób. Stosujemy je do wybierania nazw klas, do myślenia o dobrych projektach i nauczania ich tworzenia, a także do opisywania projektów

w kategoriach zestawów użytych wzorców projektowych [BJ94]. Łatwo można wyobrazić sobie bardziej zaawansowane sposoby stosowania wzorców, takie jak oparte na wzorcach narzędzia CASE lub dokumenty hipertekstowe. Jednak nawet bez skomplikowanych narzędzi wzorce są bardzo pomocne.

UZUPEŁNIENIE ISTNIEJĄCYCH METOD

Metody projektowania obiektowego mają ułatwiać tworzenie dobrych projektów, uczyć początkujących projektantów poprawnego projektowania i standaryzować sposób opracowywania projektów. Takie metody zwykle określają zbiór oznaczeń (zazwyczaj graficznych) do modelowania różnych aspektów projektu, a także zestaw reguł wyznaczających, jak i kiedy stosować poszczególne symbole. Metody projektowe zwykle opisują problemy występujące w czasie projektowania, a także sposoby ich rozwiązywania i oceniania projektu. Jednak nie udało się w nich uchwycić doświadczenia ekspertów z dziedziny projektowania.

Wierzymy, że opisane przez nas wzorce projektowe to ważny brakujący element metod projektowania obiektowego. Wzorce te pokazują, jak stosować proste techniki, takie jak obiekty, dziedziczenie i polimorfizm. Ilustrują parametryzowanie systemów za pomocą algorytmów, zachowania, stanu lub rodzaju tworzonych obiektów. Wzorce projektowe pozwalają odpowiedzieć na związane z projektem pytania „dlaczego?”, a nie tylko zarejestrować skutki dokonanych wyborów. Punkty „Warunki stosowania”, „Konsekwencje” i „Implementacja” w opisach wzorców projektowych pomogą Ci podjąć odpowiednie decyzje.

Wzorce projektowe są szczególnie przydatne przy przekształcaniu modeli analitycznych w modele implementacyjne. Mimo wielu zapewnień co do możliwości płynnego przechodzenia od analiz do projektów obiektowych w praktyce proces ten jest skomplikowany. Elastyczny i możliwy do wielokrotnego użytku projekt obejmuje zwykle obiekty, które nie występują w modelu analitycznym. Na projekt wpływają stosowane biblioteki klas i język programowania. Modele analityczne często trzeba zmodyfikować, aby można je wielokrotnie wykorzystać. Wiele wzorców projektowych z katalogu dotyczy właśnie tych problemów, dlatego nazywamy je wzorcami *projektowymi*.

Kompletna metoda projektowa wymaga wprowadzenia obok wzorców projektowych także innych ich rodzajów. Mogą to być wzorce analityczne, wzorce tworzenia interfejsu użytkownika lub wzorce poprawiania wydajności. Jednak wzorce projektowe to kluczowy element, którego do tej pory brakowało.

DOCELOWE ELEMENTY REFAKTORYZACJI

Jednym z problemów rozwijania oprogramowania wielokrotnego użytku jest to, że często trzeba zmieniać jego strukturę lub je **refaktoryzować** [OJ90]. Wzorce projektowe pomagają określić, jak zmienić strukturę projektu, a także mogą zmniejszyć zakres niezbędnej późniejszej refaktoryzacji.

Cykł życia oprogramowania obiektowego obejmuje kilka etapów. Brian Foote nazywa je fazami **tworzenia prototypów, rozwijania i konsolidowania** [Foo92].

Etap tworzenia prototypów to okres natężonych działań w czasie budowania oprogramowania przez błyskawiczne generowanie prototypów i przyrostowe wprowadzanie zmian do czasu spełnienia przez program wyjściowego zbioru wymagań oraz osiągnięcia dojrzałości. W tym momencie oprogramowanie zwykle składa się z hierarchii klas dokładnie odzwierciedlających jednostki z dziedziny problemowej. Podstawową techniką wielokrotnego wykorzystania rozwiązania jest tu otwarte ponowne wykorzystanie za pomocą dziedziczenia.

Kiedy oprogramowanie osiągnie dojrzałość i wejdzie do użytku, jego ewolucja zacznie zależeć od dwóch sprzecznych potrzeb. Oprogramowanie musi: (1) spełniać dodatkowe wymagania i (2) w większym stopniu umożliwiać powtórne wykorzystanie. Nowe wymagania powodują zwykle dodanie nowych klas i operacji, a często także całych hierarchii klas. Oprogramowanie przechodzi przez etap rozwijania pod kątem nowych wymogów. Nie może to jednak trwać zbyt długo. Ostatecznie oprogramowanie stanie się za mało elastyczne, aby można je dalej modyfikować. Hierarchie klas przestaną być zgodne z konkretną dziedziną problemową. Zamiast tego zaczną odzwierciedlać wiele takich dziedzin, a w klasach znajdą się liczne niepowiązane ze sobą operacje i zmienne egzemplarza.

Aby kontynuować ewolucję, oprogramowanie trzeba zreorganizować za pomocą procesu *refaktoryzacji*. Jest to etap, na którym często powstają platformy. Refaktoryzacja obejmuje podzielenie klas na komponenty specjalne i ogólnego użytku, przenoszenie operacji w górę lub w dół hierarchii klas i tworzenie bardziej sensownych interfejsów klas. Na etapie konsolidacji powstaje wiele nowych rodzajów obiektów. Często wynika to z podziału istniejących obiektów i zastosowania składania obiektów zamiast dziedziczenia. Dlatego otwarte powtórne wykorzystanie jest tu zastępowane przez zamknięte powtórne wykorzystanie. Ciągła konieczność zaspokajania dodatkowych wymogów w połączeniu z potrzebą zwiększenia możliwości ponownego wykorzystania prowadzi do wielokrotnego przechodzenia oprogramowania obiektowego przez etapy rozwijania i konsolidacji. Rozwijanie związane jest z realizacją nowych wymogów, a konsolidacja — ze zwiększaniem ogólności oprogramowania.



Nie można uniknąć przechodzenia przez ten cykl. Jednak dobrzy projektanci są świadomi zmian, które mogą prowadzić do refaktoryzacji. Znają też struktury klas i obiektów, które pomagają uniknąć refaktoryzacji. Ich projekty są stabilne w obliczu zmian wymagań. Staranna analiza wymogów pozwala wyróżnić te z nich, które prawdopodobnie zmienią się w cyklu życia oprogramowania, a dobry projekt będzie odporny na takie zmiany.

Omówione w książce wzorce projektowe opisują wiele struktur powstających w wyniku refaktoryzacji. Stosowanie tych wzorców na wczesnych etapach powstawania projektu zapobiega późniejszej refaktoryzacji. Jednak nawet jeśli dopiero po opracowaniu systemu dostrzeżesz, jak zastosować wzorzec, może on pomóc we wprowadzeniu zmian. Dlatego wzorce projektowe wskazują docelowe elementy refaktoryzacji.

6.2. KRÓTKA HISTORIA

Początkowo katalog wzorców był częścią rozprawy doktorskiej Ericha [Gam91, Gam92]. Znajdowała się w niej mniej więcej połowa wzorców z tej książki. Do czasu konferencji OOPSLA '91 oficjalnie powstał niezależny katalog, a Richard dołączył do Ericha w pracach nad nim. Niedługo potem współpracę nawiązał z nimi John. Przed konferencją OOPSLA '92 do grupy dołączył Ralph. Ciężko pracowaliśmy, aby katalog nadawał się do opublikowania w dokumentach z konferencji ECOOP '93, ale szybko doszliśmy do wniosku, że 90-stronicowa praca nie zostanie zaakceptowana. Dlatego streczyliśmy katalog i przesłałyśmy podsumowanie, które przyjęto. Niedługo potem zdecydowaliśmy się przekształcić katalog w książkę.

Nazwy wzorców w tym czasie nieco się zmieniły. „Nakładka” stała się „Dekoratorem”, „Klej” — „Fasadą”, „Samotnik” — „Singletonem”, a „Chodzący” — „Odwiedzającym”. Z kilku wzorców zrezygnowaliśmy, ponieważ nie wydawały się nam wystarczająco istotne. Jednak poza tym zestaw wzorców wchodzących w skład katalogu niewiele się zmienił od końca 1992 roku. Natomiast same wzorce przeszły znaczną ewolucję.

Dostrzeżenie, że coś jest wzorcem, to łatwe zadanie. Wszyscy czterej aktywnie pracujemy nad budowaniem systemów obiektowych i zauważaliśmy, że łatwo jest wykrywać wzorce na podstawie styczności z wystarczająco dużą liczbą systemów. Jednak *zajmowanie* wzorców jest znacznie łatwiejsze niż ich *opisywanie*.

Jeśli budujesz systemy, a następnie zastanawiasz się nad tym, co utworzyłeś, dostrzeżesz wzorce w swoim postępowaniu. Jednak trudno jest opisać je tak, aby osoby, które ich nie znały, zrozumiały je i zdały sobie sprawę z tego, dlaczego są ważne. Ekspertci natychmiast dostrzegli wartość katalogu po jego powstaniu. Jednak wzorce były zrozumiałe tylko dla osób, które już z nich korzystały.

Ponieważ jednym z głównych celów, jakie sobie postawiliśmy, było nauczenie projektowania obiektowego nowych projektantów, wiedzieliśmy, że musimy ulepszyć katalog. Zwiększyliśmy średnią długość opisu wzorca z poniżej 2 do ponad 10 stron przez dołączenie szczegółowego uzasadniającego przykładu i przykładowego kodu. Zaczęliśmy też analizować korzyści i koszty różnych sposobów implementacji wzorca. Sprawiło to, że uczenie się wzorców stało się łatwiejsze.

Inną ważną zmianą, która zaszła w ostatnich latach, jest zwiększenie nacisku na problem rozwiązywany przez wzorzec. Najłatwiej jest patrzyć na wzorzec jak na rozwiązanie, jak na technikę, którą można dostosować i powtórnie wykorzystać. Trudniej jest zauważać, kiedy jego stosowanie jest *właściwe* — opisać rozwiązywane problemy i kontekst, w którym wzorzec jest najlepszym rozwiązaniem. Ogólnie łatwiej jest zobaczyć, co ktoś robi, niż ustalić, dlaczego tak postępuje. Ważna jest też wiedza o celu korzystania ze wzorca, ponieważ pomaga wybierać stosowane rozwiązania, a także zrozumieć projekt istniejących systemów. Autor wzorca powinien określić i scharakteryzować problem rozwiązywany przez wzorzec, nawet jeśli musi to zrobić po odkryciu rozwiązania.

6.3. SPOŁECZNOŚĆ ZWIĄZANA ZE WZORCAMI

Nie tylko my zajmujemy się pisaniem książek katalogujących wzorce stosowane przez ekspertów. Jesteśmy częścią większej społeczności zainteresowanej ogólnie wzorcami, a w szczególności wzorcami związanymi z oprogramowaniem. Christopher Alexander to architekt, który jako pierwszy zbadał wzorce występujące w budynkach i środowiskach oraz opracował „język wzorców” do ich generowania. Jego dokonania są dla nas nieustającą inspiracją. Dlatego stosowne i wartościowe będzie porównanie jego pracy z naszą. Następnie przyjrzymy się dokonaniom innych osób w dziedzinie wzorców związanych z oprogramowaniem.

JĘZYKI WZORCÓW ALEXANDRA

Nasza praca pod wieloma względami przypomina dokonania Alexandra. I on, i my opieramy się na obserwowaniu istniejących systemów oraz wyszukiwaniu w nich wzorców. Stosujemy też szablony do opisu wzorców (choć szablony te znaczco różnią się od siebie), w omówieniu wzorców korzystamy z języka naturalnego i wielu przykładów, a nie z języków formalnych, a także uzasadniamy używanie każdego wzorca.

Jednak pod równie wieloma względami podejścia nasze i Alexandra się różnią:

1. Ludzie tworzą budynki od tysięcy lat i istnieje wiele klasycznych przykładów, na których można się opierać. Systemy oprogramowania są rozwijane od stosunkowo niedługiego czasu i bardzo niewielkie z nich są uważane za klasyczne.
2. Alexander podaje kolejność stosowania wzorców, natomiast my tego nie robimy.
3. We wzorcach Alexandra nacisk położony jest na rozwiązywanie problemy, natomiast we wzorcach projektowych opisujemy szczegółowo same rozwiązania.
4. Alexander twierdzi, że jego wzorce pozwalają tworzyć kompletne budynki. My nie uważamy, że wzorce projektowe pozwolą zbudować kompletne programy.

Kiedy Alexander stwierdza, że można zaprojektować dom po prostu przez zastosowanie jego wzorców jeden po drugim, ma podobne cele, co niektórzy metodolođy z obszaru projektowania obiektowego, którzy przedstawiają reguły projektowania krok po kroku. Alexander nie zaprzecza temu, że pomysłowość jest potrzebna. Niektóre z jego wzorców wymagają zrozumienia nawyków osób, które będą korzystać z budynku. Ponadto wiara Alexandra w „poezję” projektu wskazuje na konieczność posiadania wiedzy wykraczającej poza sam język wzorców¹. Jednak opis generowania projektów na podstawie wzorców sugeruje, że język wzorców sprawia, iż proces projektowania jest deterministyczny i powtarzalny.

Perspektywa Alexandra pomogła nam skoncentrować się na korzyściach i kosztach projektów — różnych „siłach”, które je kształtują. Jego wpływ zachęcił nas do ciejszej pracy w celu zrozumienia warunków stosowania wzorców i konsekwencji ich użycia. Ponadto nie martwiliśmy się o definiowanie formalnej reprezentacji wzorców. Choć taka reprezentacja może umożliwić automatyzację wzorców, na obecnym etapie ważniejsze jest zbadanie przestrzeni wzorców obiektowych niż jej sformalizowanie.

¹ Zobacz *The poetry of the language* [AIS-77].

Z punktu widzenia Alexandra wzorce opisane w tej książce nie tworzą języka wzorców. Z uwagi na różnorodność systemów oprogramowania rozwijanych przez ludzi trudno jest wyobrazić sobie możliwość udostępniania „kompletnego” zestawu wzorców, który przedstawia instrukcje projektowania aplikacji krok po kroku. Możemy to zrobić dla określonych klas aplikacji, na przykład programów do tworzenia raportów lub systemów wypełniania formularzy. Jednak nasz katalog to jedynie kolekcja powiązanych ze sobą wzorców. Nie możemy udawać, że jest to język wzorców.

Uważamy, że mało prawdopodobne jest, aby *kiedykolwiek* powstał kompletny język wzorców z dziedziny oprogramowania. Jednak z pewnością możliwe jest zbudowanie *bardziej kompletnego* katalogu. Dodatki musiałyby obejmować opis platform i ich stosowania [Joh92], wzorce projektowania interfejsów użytkownika [BJ94], wzorce analityczne [Coa92] i wszystkie inne aspekty rozwijania oprogramowania. Wzorce projektowe to tylko część większego języka wzorców dla oprogramowania.

WZORCE W OPROGRAMOWANIU

Nasze pierwsze wspólne doświadczenie w badaniu architektury oprogramowania to warsztaty OOPSLA '91 prowadzone przez Bruce'a Andersona. Były one poświęcone tworzeniu podręcznika dla architektów oprogramowania (oceniając na podstawie tej książki, uważamy, że lepszą nazwą byłaby „encyklopedia” niż „podręcznik” architektury). Te pierwsze warsztaty doprowadziły do serii spotkań, a ostatnie z nich miało miejsce przy okazji pierwszej konferencji Pattern Languages of Programs. Odbyła się ona w sierpniu 1994 roku. W ten sposób powstała społeczność osób zainteresowanych dokumentowaniem wiedzy eksperckiej z obszaru oprogramowania.

Oczywiście podobny cel przyświecał też innym. Książka *The Art of Computer Programming*² Donalda Knutha [Knu73] była jedną z pierwszych prób skatalogowania wiedzy na temat oprogramowania, choć autor skoncentrował się na opisie algorytmów. Mimo to zadanie okazało się zbyt rozbudowane, aby udało się je ukończyć. Seria *Graphics Gems* [Gla90, Arv91, Kir92] to następny katalog wiedzy na temat oprogramowania, jednak również te książki dotyczą głównie algorytmów. Program Domain Specific Software Architecture sponsorowany przez Departament Obrony Stanów Zjednoczonych [GM92] ma służyć zbieraniu informacji o architekturach. Społeczność związana z inżynierią oprogramowania na podstawie wiedzy stara się przedstawić ogólną wiedzę związaną z oprogramowaniem. Istnieje wiele grup, których cele są przynajmniej trochę zbliżone do naszych.

Książka *Advanced C++: Programming Styles and Idioms*³ Jamesa Copliena [Cop92] wywarła wpływ także na nas. Wzorce z tej książki są nieco bardziej specyficzne dla języka C++ niż wzorce opisane przez nas, a książka Copliena obejmuje także wiele wzorców niższego poziomu. Występują jednak pewne podobieństwa, na co wskazujemy w opisach wzorców. Jim aktywnie działa w społeczności osób zainteresowanych wzorcami. Obecnie pracuje nad wzorcami opisującymi role osób w firmach zajmujących się rozwijaniem oprogramowania.

² Wydanie polskie: *Sztuka programowania*, WNT, 2003 — przyp. tłum.

³ Wydanie polskie: *C++. Styl i technika zaawansowanego programowania*, Helion, 2004 — przyp. tłum.

Jest też wiele innych miejsc, w których można znaleźć opisy wzorców. Kent Beck był jedną z pierwszych osób w społeczności zajmującej się oprogramowaniem, które zwróciły uwagę na pracę Christophera Alexandra. Kent w 1993 roku rozpoczął prowadzenie rubryki poświęconej wzorom języka Smalltalk w magazynie *The Smalltalk Report*. Także Peter Coad od pewnego czasu zbiera wzorce. Jego tekst dotyczy głównie wzorców analitycznych [Coa92]. Nie znamy najnowszych wzorców Petera, choć wiemy, że wciąż nad nimi pracuje. Słyszeliśmy o tym, że powstaje kilka książek poświęconych wzorom, ale nie widzieliśmy żadnej z nich. Możemy jedynie poinformować, że się pojawią. Jedna z tych książek będzie pochodziła z konferencji Pattern Languages of Programs.

6.4. ZAPROSZENIE

Co możesz zrobić, jeśli interesują Cię wzorce? Po pierwsze, korzystaj z nich i zwracaj uwagę na inne wzorce pasujące do sposobu, w jaki projektujesz. W najbliższych latach pojawi się wiele książek i artykułów na temat wzorców, dlatego dostępnych będzie wiele ich źródeł. Rozwijaj słownictwo związane ze wzorcami i korzystaj z niego. Stosuj je, kiedy rozmawiasz z innymi o projektach. Używaj go, kiedy o nich myślisz i piszesz.

Po drugie, bądź krytycznym konsumentem. Katalog wzorców projektowych to wynik ciężkiej pracy — nie tylko naszej, ale też wielu recenzentów, którzy przekazali nam informacje zwrotne. Jeśli dostrzeżesz problem lub uważasz, że potrzebne są dodatkowe wyjaśnienia, skontaktuj się z nami. To samo dotyczy każdego innego katalogu wzorców projektowych — przekaż informacje zwrotne autorom! Jedną ze wspaniałych cech wzorców jest to, że pozwalają podejmować decyzje projektowe nie tylko na podstawie niejasnej intuicji. Wzorce umożliwiają autorom jawne określenie kompromisów, na które się decydują. Ułatwia to dostrzeżenie wad wzorców i dyskusję z ich autorami. Wykorzystaj to.

Po trzecie, zwracaj uwagę na stosowane wzorce i zapisuj je. Umieszczaj je w dokumentacji. Pokaż innym osobom. Nie musisz pracować w laboratorium badawczym, aby wyszukiwać wzorce. Tak naprawdę znajdowanie adekwatnych wzorców jest prawie niemożliwe bez praktycznego doświadczenia. Swobodnie rozwijaj własny katalog wzorców, ale koniecznie skorzystaj z pomocy innej osoby przy dopracowywaniu ich ostatecznego kształtu!

6.5. SŁOWO NA ZAKOŃCZENIE

W najlepszych projektach stosowanych jest wiele wzorców projektowych, które zazębają się i przeplatają, tworząc większą całość. Jak pisze Christopher Alexander:

Można tworzyć budynki przez łączenie wzorców w luźny sposób. Budynek powstały w ten sposób jest zestawem wzorców. Nie jest spoisty. Brakuje w nim glebi. Jednak można też połączyć wzorce w taki sposób, aby wiele z nich pokrywało się w tej samej przestrzeni fizycznej. Taki budynek jest bardzo spoisty. Na małej przestrzeni umieszczono w nim wiele znaczeń. I dzięki tej spoistości zyskuje na glebi.

A Pattern Language [AIS-77, strona XLI]

DODATEK A

Słowniczek

Delegacja. Mechanizm implementacyjny polegający na przekazywaniu lub *delegowaniu* przez obiekt żądań do innych obiektów. Delegat obsługuje żądanie na rzecz pierwotnego obiektu.

Destruktor. W języku C++ jest to operacja wywoływana automatycznie dla obiektu przeznaczonego do usunięcia.

Diagram interakcji. Diagram ilustrujący przepływ żądań między obiektemi.

Diagram klas. Schemat ilustrujący klasy, ich wewnętrzną strukturę i operacje, a także statyczne relacje między klasami.

Diagram obiektów. Diagram ilustrujący konkretną strukturę obiektów w czasie wykonywania programu.

Dziedziczenie prywatne. W języku C++ jest to dziedziczenie po klasie wyłącznie jej implementacji.

Dziedziczenie. Relacja pozwalająca zdefiniować jedną jednostkę w kategoriach innej. **Dziedziczenie klas** polega na definiowaniu nowej klasy na podstawie jednej lub kilku klas nadrzędnych. Nowa klasa dziedziczy po klasach nadrzędnych interfejs oraz implementację i jest nazywana **podklassą** lub (w języku C++) **klasą pochodną**. Dziedziczenie klas obejmuje **dziedziczenie interfejsu** i **dziedziczenie implementacji**. Dziedziczenie interfejsu polega na definiowaniu nowego interfejsu w kategoriach jednego lub kilku istniejących. Przy dziedziczeniu implementacji nowa implementacja jest definiowana w kategoriach jednej lub kilku istniejących.

Interfejs. Zestaw wszystkich sygnatur zdefiniowanych w operacjach obiektu. Interfejs opisuje zbiór żądań obsługiwanych przez obiekt.

Kapsułkowanie. Wynik ukrycia reprezentacji i implementacji w obiekcie. Reprezentacja nie jest widoczna, dlatego nie można bezpośrednio uzyskać do niej dostępu spoza obiektu. Jedyny sposób na dostęp do reprezentacji obiektu i zmodyfikowanie jej to skorzystanie z operacji.

Klasa abstrakcyjna. Klasa przeznaczona przede wszystkim do definiowania interfejsu. Implementacja części lub wszystkich elementów klasy abstrakcyjnej jest umieszczana w jej podklasach. Nie można tworzyć egzemplarzy klas abstrakcyjnych.

Klasa konkretna. Klasa bez operacji abstrakcyjnych. Takie klasy umożliwiają tworzenie egzemplarzy.

Klasa mieszana (ang. *mixin class*). Klasa zaprojektowana tak, aby łączyć ją z innymi klasami za pomocą dziedziczenia. Klasy mieszane są zwykle abstrakcyjne.

Klasa nadrzędna. Klasa, po której dziedziczy inna klasa. Synonimy tej nazwy to **nadklasa** (język Smalltalk), **klasa bazowa** (język C++) i **klasa macierzysta**.

Klasa zaprzyjaźniona. W języku C++ jest to klasa mająca te same prawa dostępu do operacji i danych innej klasy co owa klasa.

Klasa. Klasa definiuje interfejs i implementację obiektu. Określa wewnętrzną reprezentację obiektu oraz operacje, które może on wykonywać.

Konstruktor. W języku C++ jest to operacja automatycznie wywoływana w celu zainicjowania nowych egzemplarzy.

Metaklasa. W języku Smalltalk klasy są obiektami. Metaklasa to klasa obiektu reprezentującego klasy.

Nadtyp. Typ nadrzędny, po którym dziedziczy inny typ.

Obiekt zagregowany. Obiekt składający się z obiektów podrzędnych, nazywanych zwykle **elementami** lub **częściami**. Obiekt zagregowany odpowiada za obiekty podrzędne.

Obiekt. Jednostka działająca w czasie wykonywania programu. Obejmuje dane i procedury manipulujące tymi danymi.

Odbiorca. Obiekt docelowy dla żądania.

Operacja abstrakcyjna. Operacja z zadeklarowaną sygnaturą, ale bez implementacji. W języku C++ odpowiednikiem operacji abstrakcyjnych są **czysto wirtualne funkcje składowe**.

Operacja. Danymi w obiekcie można manipulować tylko za pomocą jego operacji. Obiekt wykonyuje operacje po otrzymaniu żądania. W języku C++ operacje są nazywane **funkcjami składowymi**, a w języku Smalltalk — **metodami**.

Operacje statyczne. Operacje działające na klasie, a nie na poszczególnych obiektach. W języku C++ operacje statyczne są nazywane **statycznymi funkcjami składowymi**.

Otwarte powtórne wykorzystanie (ang. *white-box reuse*). Sposób powtórnego wykorzystania kodu oparty na dziedziczeniu klas. Podklasa ponownie wykorzystuje interfejs i implementację klasy nadrzędnej, a ponadto może mieć dostęp do zwykle prywatnych elementów klasy nadrzędnej.

Pakiet narzędziowy. Kolekcja klas udostępniająca przydatne funkcje, jednak nieokreślająca projektu aplikacji.

Platforma (ang. *framework*). Zestaw współdziałających klas składających się na przeznaczony do wielokrotnego użytku projekt oprogramowania określonego rodzaju. Platforma zapewnia wskazówki co do architektury, ponieważ dzieli projekt na klasy abstrakcyjne i definiuje ich zadania oraz sposoby współdziałania. Programista dostosowuje platformę do konkretnej aplikacji przez tworzenie podklas i klas zagregowanych na podstawie egzemplarzy klas platformy.

Podklasa. Klasa dziedzicząca po innej. W języku C++ podklasy są nazywane **klassami pochodnymi**.

Podsystem. Niezależna grupa klas współdziałających przy wykonywaniu zbioru zadań.

Podtyp. Jeśli jeden typ obejmuje interfejs innego typu, jest podtypem tego drugiego.

Polimorfizm. Możliwość zastępowania w czasie wykonywania programu obiektów innymi obiektami o pasującym interfejsie.

Powiązanie abstrakcyjne (ang. *abstract coupling*). Jeśli klasa A obejmuje referencję do klasy abstrakcyjnej B, mówimy, że klasa A jest *abstrakcyjnie powiązana* z klasą B. Nazwa techniki wynika z tego, że klasa A zawiera referencję do *typu*, a nie do konkretnego obiektu.

Powiązanie. Poziom zależności między komponentami oprogramowania.

Protokół. Jest to interfejs rozwinięty o listę akceptowanych sekwencji żądań.

Przesłanianie. Ponowne definiowanie w podklasie operacji odziedziczonej po klasie nadzędnej.

Referencja do obiektu. Wartość identyfikująca inny obiekt.

Relacja agregacji. Jest to relacja między obiektem zagregowanym i jego elementami. W klasach ta relacja definiowana jest dla ich egzemplarzy (czyli obiektów zagregowanych).

Relacja znajomości (ang. *acquaintance relationship*). Klasa odwołująca się do innej klasy pozostała z nią w relacji *asocjacji*.

Składanie obiektów. Składanie lub *komponowanie* obiektów w celu uzyskania obiektu o bardziej złożonych zachowaniach.

Sygnatura. Sygnatura operacji określa jej nazwę, parametry i zwracaną wartość.

Typ sparametryzowany. Typ, w którym część typów składowych pozostaje nieokreślona.

Te niesprecyzowane typy są podawane za pomocą parametrów w miejscu zastosowania danego typu. W języku C++ typy sparametryzowane są nazywane **szablonami**.

Typ. Nazwa określonego interfejsu.

Wiązanie dynamiczne. Łączenie w czasie wykonywania programu żądania do obiektu z jedną z jego operacji. W języku C++ dynamicznie wiązane są tylko funkcje wirtualne.

Wzorzec projektowy. Wzorzec projektowy pozwala w systematyczny sposób nazwać, uzasadnić i wyjaśniać ogólny projekt rozwiązujejący problem projektowy powtarzający się w systemach obiektowych. Wzorzec opisuje problem, a także rozwiązanie, warunki jego stosowania i skutki jego wykorzystania. Obejmuje też wskazówki dotyczące implementacji i przykłady. Rozwiązanie to ogólny układ obiektów i klas pozwalający rozwiklać problem. Aby poradzić sobie z problemem w konkretnym kontekście, należy dostosować rozwiązanie do warunków i zaimplementować je.

Zamknięte ponowne wykorzystanie (ang. *black-box reuse*). Sposób powtórnego wykorzystania kodu oparty na składaniu obiektów. Składane obiekty nie ujawniają wewnętrznych mechanizmów innym obiektom, dlatego przypominają „zamknięte „czarne skrzynki”.

Zmienna egzemplarza. Fragment danych definiujący część reprezentacji obiektu. W języku C++ używana jest nazwa **zmienna składowa**.

Żądanie. Obiekt wykonuje operację, kiedy otrzyma od innego obiektu odpowiadające jej żądanie. Często stosowanym synonimem określenia „żądanie” jest **komunikat**.

DODATEK B

Przewodnik po notacji

W książce stosujemy diagramy do ilustrowania ważnych kwestii. Niektóre diagramy, na przykład zrzut przedstawiający okno dialogowe lub schematyczne drzewo obiektów, są nieformalne. Jednak do samych wzorców projektowych stosujemy bardziej formalne notacje, aby określić relacje i interakcje między klasami oraz obiektami. W tym dodatku szczegółowo opisujemy te notacje.

Stosujemy trzy różne typy diagramów:

1. **Diagramy klas** ilustrują klasy, ich strukturę oraz statyczne relacje między nimi.
2. **Diagramy obiektów** przedstawiają konkretną strukturę obiektów w czasie wykonywania programu.
3. **Diagram interakcji** określa przepływ żądań między obiektami.

Opis każdego wzorca projektowego obejmuje przynajmniej jeden diagram klas. W razie potrzeby jako uzupełnienie analiz stosowane są inne diagramy. Diagramy klas i obiektów oparte są na notacji OMT (ang. *Object Modeling Technique*) [RBP-91, Rum94]¹. Diagramy interakcji pochodzą z notacji Objectory [JCJO92] i metody Boochka [Boo94]. Notacje te przedstawiamy w skrócie na początku i końcu książki.

B.1. DIAGRAM KLAS

Rysunek B.1a przedstawia notację OMT dla klas abstrakcyjnych i konkretnych. Klasę oznacza prostokąt z nazwą klasy zapisaną pogrubioną czcionką w górnej części pola. Pod nazwą klasy wymienione są jej kluczowe operacje. Pod operacjami znajduje się lista wszystkich zmiennych egzemplarza. Informacje o typie są opcjonalne. Stosujemy konwencje z języka C++. Zgodnie z nimi nazwa typu znajduje się przed nazwą operacji (w celu podkreślenia zwracanego typu), zmiennej egzemplarza lub parametru. Czcionka pochyła oznacza, że klasa lub operacja jest abstrakcyjna.

¹ W notacji OMT nazwa „diagram obiektów” oznacza diagram klas. W książce stosujemy nazwę „diagram obiektów” wyłącznie do określania struktur obiektów.

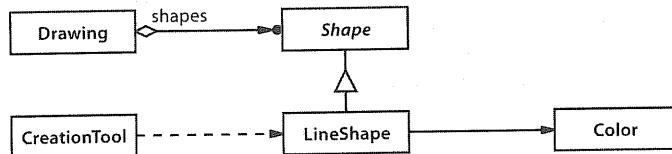
<i>AbstractClassName</i>
<i>AbstractOperation1()</i> Type <i>AbstractOperation2()</i>

<i>ConcreteClassName</i>
<i>Operation1()</i> Type <i>Operation2()</i>
<i>instanceVariable1</i> Type <i>instanceVariable2</i>

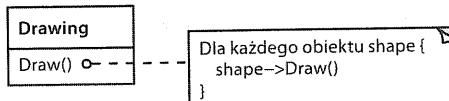
(a) Klasa abstrakcyjne i konkretne



(b) Klasa Client jako element wzorca (po lewej) i niejawna klasa Client (po prawej)



(c) Relacje między klasami



(d) Uwagi w pseudokodzie

W niektórych wzorcach projektowych warto wiedzieć, które klasy klienckie odwołują się do klas reprezentujących elementy wzorca. Jeśli jednym z tych elementów jest klasa Client (oznacza to, że klient pełni we wzorcu określone zadanie), pojawia się ona jako zwykła klasa. Ma to miejsce na przykład we wzorcu Pyłek (s. 201). Jeżeli wzorzec nie obejmuje elementu w postaci klasy Client (czyli klienci nie wykonują żadnych zadań), jednak dodanie jej pozwala wyjaśnić, które elementy współdziałają z klientami, klasa Client jest przedstawiona w kolorze szarym, jak ilustruje to rysunek B.1b. Tak działa na przykład wzorzec Pełnomocnik (s. 191). Szary kolor klasy Client informuje też, że nieprzypadkowo pominęliśmy ją w omówieniu elementów wzorca.

Rysunek B.1c przedstawia różne relacje między klasami. W notacji OMT dziedziczenie klas jest przedstawione za pomocą trójkąta łączącego podklasę (LineShape na rysunku) z klasą nadzczną (Shape). Referencja do obiektu reprezentująca relację bycia częścią lub agregacji jest oznaczona strzałką z rombem na początku. Grot wskazuje na klasę aggregatu (na przykład Shape). Strzałka bez rombu oznacza relację znajomości (na przykład klasa LineShape przechowuje

referencję do obiektu `Color`, który może być współużytkowany przez inne kształty). Nazwa takiej referencji może pojawić się obok początku strzałki, co pozwala odróżnić ją od pozostałych referencji².

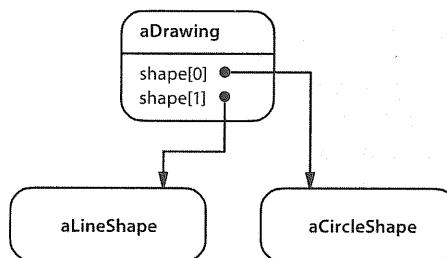
Warto też pokazać, które klasy tworzą egzemplarze innych klas. Stosujemy do tego przerywane strzałki, ponieważ notacja OMT nie opisuje takiej relacji (nazywamy ją relacją „tworzenia”). Grot wskazuje klasę, której egzemplarz powstaje. Na rysunku B.1c klasa `CreationTool` tworzy obiekty `LineShape`.

W notacji OMT wypełniony okrąg oznacza „więcej niż jeden”. Kiedy taki symbol pojawia się na końcu linii przedstawiającej referencję, oznacza to, że agregowanych lub wskazywanych za pomocą tej referencji jest wiele obiektów. Rysunek B.1c pokazuje, że w klasie `Drawing` zagregowano wiele obiektów typu `Shape`.

Ponadto wzbogaciliśmy notację OMT o komentarze w postaci pseudokodu, które pozwalają naszkicować implementację operacji. Rysunek B.1d przedstawia takie uwagi dotyczące operacji `Draw` klasy `Drawing`.

B.2. DIAGRAM OBIEKTÓW

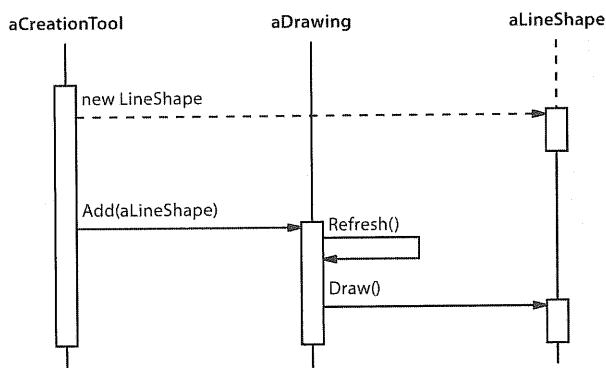
Diagram obiektów ilustruje wyłącznie egzemplarze klas. Przedstawia obraz obiektów opisanych we wzorcu projektowym. Obiekty noszą nazwy *aSomething*, gdzie *Something* to klasa obiektu. Symbolem reprezentującym obiekt (zmodyfikowanym nieco w porównaniu ze standardową notacją OMT) jest prostokąt z zaokrąglonymi wierzchołkami i linią oddzielającą nazwę obiektu od referencji do obiektów. Strzałki wskazują obiekty, do których prowadzą referencje. Przykład znajduje się na rysunku B.2.



² W notacji OMT zdefiniowane są też **asocjacje** między klasami reprezentowane jako zwykłe linie między prostokątami przedstawiającymi klasy. Asocjacje są dwukierunkowe. Choć ich stosowanie jest właściwe na etapie analiz, uważałyśmy, że jest to relacja ze zbyt wysokiego poziomu, aby przedstawiać ją we wzorcach projektowych (ponieważ asocjacje w czasie projektowania trzeba odwzorować na elementy z niższego poziomu — referencje do obiektów lub wskaźniki). Referencje do obiektów są z natury jednokierunkowe, dlatego lepiej nadają się do przedstawiania interesujących nas relacji. Na przykład obiekt `Drawing` wie o istnieniu obiektów `Shape`, ale obiekty `Shape` nie znają obiektu `Drawing`, w którym się znajdują. Nie można przedstawić tej relacji za pomocą samych asocjacji.

B.3. DIAGRAM INTERAKCJI

Diagram interakcji przedstawia kolejność wykonywania żądań przekazywanych między obiektami. Na rysunku B.3 widać diagram interakcji ilustrujący dodawanie figury do rysunku.



Czas na diagramach interakcji biegnie od góry do dołu. Ciągła pionowa linia określa czas życia danego obiektu. Konwencje nazewnictwa dla obiektów są tu takie same jak na diagramach obiektów — stosujemy nazwę klasy z przedrostkiem w postaci litery „a” (na przykład aShape). Jeśli obiekt powstaje dopiero w czasie przedstawionym na diagramie, do punktu jego utworzenia jest reprezentowany za pomocą linii przerywanej.

Pionowy prostokąt informuje, że obiekt jest aktywny (oznacza to, że obsługuje żądanie). Operacja może wysyłać żądania do innych obiektów. Jest to oznaczone poziomą strzałką prowadzącą do obiektu odbiorcy. Nad strzałką zapisana jest nazwa żądania. Żądania utworzenia obiektu mają postać przerywanej strzałki. Żądanie skierowane do nadawcy ilustruje strzałka prowadzącą z powrotem do niego.

Rysunek B.3 pokazuje, że pierwsze żądanie jest zgłoszane przez obiekt aCreationTool i dotyczy utworzenia obiektu aLineShape. Następnie obiekt aLineShape jest dodawany (żądanie Add) do obiektu aDrawing, co powoduje wysłanie przez obiekt aDrawing żądania Refresh do samego siebie. Warto zauważyć, że obiekt aDrawing w ramach operacji Refresh wysyła żądanie Draw do obiektu aLineShape.

Diagram interakcji jest narzędziem pomocnym w analizie i projektowaniu systemów. Wykorzystywany jest do przedstawiania i analizowania kolejności wykonywania żądań przekazywanych między obiektami. Współczesne edytory UML oferują funkcję generowania kodu źródłowego na podstawie diagramów interakcji, co ułatwia ich zastosowanie w praktyce programistycznej. Warto zauważyć, że diagramy interakcji są jednym z wielu narzędzi do modelowania systemów, a ich skuteczność zależy od dobrej współpracy z innymi metodami i technikami projektowania.

DODATEK C

Klasy podstawowe

Ten dodatek obejmuje dokumentację klas podstawowych stosowanych w przykładowym kodzie w języku C++ w kilku wzorcach projektowych. Klasy te celowo są proste i jak najmniejsze. Opisujemy tu następujące klasy:

- ▶ **List** (uporządkowana lista obiektów);
- ▶ **Iterator** (interfejs zapewniający sekwencyjny dostęp do obiektów agregatu);
- ▶ **ListIterator** (iterator do przechodzenia po zawartości obiektów **List**);
- ▶ **Point** (dwuwymiarowy punkt);
- ▶ **Rect** (prostokąt równoległy do układu współrzędnych).

Niektóre z nowszych typów standardowych języka C++ mogą nie być dostępne w części kompilatorów. W szczególności jeśli używasz kompilatora bez definicji typu **bool**, możesz zdefiniować go ręcznie w następujący sposób:

```
typedef int bool;
const int true = 1;
const int false = 0;
```

C.1. LIST

Szablon klasy **List** to podstawowy kontener do przechowywania uporządkowanych list obiektów. W obiektach **List** zapisane są wartości elementów, co oznacza, że klasa ta działa zarówno dla typów wbudowanych, jak i dla egzemplarzy klas. Na przykład instrukcja **List<int>** to deklaracja listy elementów **int**. Jednak w większości wzorców klasa **List** służy do przechowywania wskaźników do obiektów, tak jak w deklaracji **List<Glyph*>**. W ten sposób klasę **List** można wykorzystać do przechowywania różnorodnych obiektów.

Dla wygody w klasie **List** udostępniono synonimy do wykonywania operacji na stosie. Powoduje to, że kod, w którym klasę **List** zastosowano jako stos, jest zrozumiały, a przy tym nie trzeba definiować nowej klasy.

```

template <class Item>
class List {
public:
    List(long size = DEFAULT_LIST_CAPACITY);
    List(List&);
    ~List();
    List& operator=(const List&);

    long Count() const;
    Item& Get(long index) const;
    Item& First() const;
    Item& Last() const;
    bool Includes(const Item&) const;

    void Append(const Item&);
    void Prepend(const Item&);

    void Remove(const Item&);
    void RemoveLast();
    void RemoveFirst();
    void RemoveAll();

    Item& Top() const;
    void Push(const Item&);
    Item& Pop();
};


```

W następnych punktach szczegółowo opisujemy te operacje.

KONSTRUOWANIE, USUWANIE, INICJOWANIE I PRZYPISYWANIE

List(long size)

Inicjuje listę. Parametr `size` określa początkową liczbę elementów.

List(List&)

Przesłania domyślny konstruktor kopiący, aby prawidłowo zainicjować dane składowe.

~List()

Zwalnia zasoby wewnętrznych struktur danych listy, ale *nie* elementów listy. Omawiana klasa nie jest zaprojektowana pod kątem tworzenia jej podklas, dlatego destruktör nie jest wirtualny.

List& operator=(const List&)

Obejmuje implementację operacji przypisywania (umożliwia prawidłowe przypisywanie wartości do danych składowych).

DOSTĘP DO ELEMENTÓW

Operacje z tej grupy zapewniają podstawowy dostęp do elementów listy.

long Count() const

Zwraca liczbę obiektów przechowywanych na liście.

Item& Get(long index) const

Zwraca obiekt zapisany pod danym indeksem.

Item& First() const

Zwraca pierwszy obiekt z listy.

Item& Last() const

Zwraca ostatni obiekt z listy.

DODAWANIE ELEMENTÓW

void Append(const Item&)

Dodaje argument do listy jako jej ostatni element.

void Prepend(const Item&)

Dodaje argument do listy jako jej pierwszy element.

USUWANIE ELEMENTÓW

void Remove(const Item&)

Usuwa podany element z listy. Ta operacja wymaga, aby typ elementów listy obsługiwał operator porównywania (==).

void RemoveFirst()

Usuwa pierwszy element z listy.

void RemoveLast()

Usuwa ostatni element z listy.

void RemoveAll()

Usuwa wszystkie elementy z listy.

INTERFEJS STOSU

Item& Top() const

Zwraca wierzchołek stosu (jeśli klasa List jest stosowana jak stos).

void Push(const Item&)

Umieszcza element na stosie.

Item& Pop()

Zdejmuje wierzchołek ze stosu.

C.2. ITERATOR

Iterator to klasa abstrakcyjna definiująca interfejs do przechodzenia po elementach agregatów.

```
template <class Item>
class Iterator {
public:
    virtual void First() = 0;
    virtual void Next() = 0;
    virtual bool IsDone() const = 0;
    virtual Item CurrentItem() const = 0;
protected:
    Iterator();
};
```

Operacje z tej klasy wykonują następujące zadania:

`virtual void First()`

Umieszcza iterator na pierwszym obiekcie agregatu.

`virtual void Next()`

Przenosi iterator do następnego obiektu w ciągu.

`virtual bool IsDone() const`

Zwraca wartość true, jeśli w ciągu nie ma już dalszych obiektów.

`virtual Item CurrentItem() const`

Zwraca obiekt znajdujący się na bieżącej pozycji w ciągu.

C.3. LISTITERATOR

Klasa ListIterator to implementacja interfejsu Iterator służąca do przechodzenia po elementach obiektów List. Konstruktor tej klasy przyjmuje jako argument listę, po której ma się poruszać.

```
template <class Item>
class ListIterator : public Iterators<Item> {
public:
    ListIterator(const List<Item>* aList);

    virtual void First();
    virtual void Next();
    virtual bool IsDone() const;
    virtual Item CurrentItem() const;
};
```

C.4. POINT

Klasa Point reprezentuje punkt w dwuwymiarowym kartezjańskim układzie współrzędnych. Klasa to obsługuje podstawową arytmetykę wektorową. Współrzędne obiektu Point są zdefiniowane w następujący sposób:

```
typedef float Coord;

Operacji klasy Point nie trzeba wyjaśniać.

class Point {
public:
    static const Point Zero;

    Point(Coord x = 0.0, Coord y = 0.0);

    Coord X() const; void X(Coord x);
    Coord Y() const; void Y(Coord y);

    friend Point operator+(const Point&, const Point&);
    friend Point operator-(const Point&, const Point&);
    friend Point operator*(const Point&, const Point&);
    friend Point operator/(const Point&, const Point&);

    Point& operator+=(const Point&);
    Point& operator-=(const Point&);
    Point& operator*=(const Point&);
    Point& operator/^(const Point&);

    Point operator-();

    friend bool operator==(const Point&, const Point&);
    friend bool operator!=(const Point&, const Point&);

    friend ostream& operator<<(ostream&, const Point&);
    friend istream& operator>>(istream&, Point&);
};
```

Składowa statyczna Zero reprezentuje obiekt Point(0, 0).

C.5. RECT

Klasa Rect reprezentuje prostokąt równoległy do osi układu współrzędnych. Obiekt Rect jest określany przez punkt początkowy oraz wymiary (szerokość i wysokość). Operacje klasy Rect nie wymagają wyjaśnień.

```
class Rect {
public:
    static const Rect Zero;

    Rect(Coord x, Coord y, Coord w, Coord h);
    Rect(const Point& origin, const Point& extent);
```

```
Coord Width() const; void Width(Coord);
Coord Height() const; void Height(Coord);
Coord Left() const; void Left(Coord);
Coord Bottom() const; void Bottom(Coord);

Point& Origin() const; void Origin(const Point&);
Point& Extent() const; void Extent(const Point&);

void MoveTo(const Point&);
void MoveBy(const Point&);

bool IsEmpty() const;
bool Contains(const Point&) const;
};
```

Składowa statyczna Zero odpowiada następującemu prostokątowi:

```
Rect(Point(0, 0), Point(0, 0));
```

BIBLIOGRAFIA

- [Add94] *NEXTSTEP General Reference: Release 3, Volumes 1 and 2*, Addison-Wesley, Reading, MA, 1994.
- [AG90] D.B. Anderson, S. Gossain, „Hierarchy evolution and the software lifecycle”, w: *TOOLS '90 Conference Proceedings*, strony 41 – 50, Prentice Hall, Paryż, lipiec 1990.
- [AIS-77] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, S. Angel, *A Pattern Language*, Oxford University Press, Nowy Jork, 1977.
- [App89] Apple Computer, Inc., *Macintosh Programmers Workshop Pascal 3.0 Reference*, Cupertino, CA, 1989.
- [App92] Apple Computer, Inc., *Dylan. An object-oriented dynamic language*, Cupertino, CA, 1992.
- [Arv91] J. Arvo, *Graphics Gems II*, Academic Press, Boston, MA, 1991.
- [AS85] B. Adelson, E. Soloway, „The role of domain experience in software design”, *IEEE Transactions on Software Engineering*, 11(11): 1351 – 1360, 1985.
- [BE93] A. Birrer, T. Eggenschwiler, „Frameworks in the financial engineering domain: An experience report”, w: *European Conference on Object-Oriented Programming*, strony 21 – 35, Springer-Verlag, Kaiserslautern, Niemcy, lipiec 1993.
- [BJ94] K. Beck, R. Johnson, „Patterns generate architectures”, w: *European Conference on Object-Oriented Programming*, strony 139 – 149, Springer-Verlag, Bolonia, Włochy, lipiec 1994.
- [Boo94] G. Booch, *Object-Oriented Analysis and Design with Applications, Second Edition*, Benjamin/Cummings, Redwood City, CA, 1994.
- [Bor81] A. Borning, „The programming language aspects of ThingLab—a constraint-oriented simulation laboratory”, *ACM Transactions on Programming Languages and Systems*, 3(4): 343 – 387, październik 1981.
- [Bor94] Borland International, Inc., *A Technical Comparison of Borland ObjectWindows 2.0 and Microsoft MFC 2.5*, Scotts Valley, CA, 1994.

- [BV90] G. Booch, M. Vilot, „The design of the C++ Booch components”, w: *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, strony 1 – 11, ACM Press, Ottawa, Kanada, październik 1990.
- [Cal93] P.R. Calder, *Building User Interfaces with Lightweight Objects*, rozprawa doktorska, Stanford University, 1993.
- [Car89] J. Carolan, „Constructing bullet-proof classes”, w: *Proceedings C++ at Work '89*, SIGS Publications, 1989.
- [Car92] T. Cargill, *C++ Programming Style*, Addison-Wesley, Reading, MA, 1992.
- [CIRM93] R.H. Campbell, N. Islam, D. Raila, P. Madeany, „Designing and Implementing Choices: An object-oriented system in C++”, *Communications of the ACM*, 36(9): 117 – 126, wrzesień 1993.
- [CL90] P.R. Calder, M.A. Linton, „Glyphs: Flyweight objects for user interfaces”, w: *ACM User Interface Software Technologies Conference*, strony 92 – 101, Snowbird, UT, październik 1990.
- [CL92] P.R. Calder, M.A. Linton, „The object-oriented implementation of a document editor”, w: *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, strony 154 – 165, ACM Press, Vancouver, British Columbia, Kanada, październik 1992.
- [Coa92] P. Coad, „Object-oriented patterns”, w: *Communications of the ACM*, 35(9): 152 – 159, wrzesień 1992.
- [Coo92] W.R. Cook, „Interfaces and specifications for the Smalltalk-80 collection classes”, w: *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, strony 1 – 15, ACM Press, Vancouver, British Columbia, Kanada, październik 1992.
- [Cop92] J.O. Coplien, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, Reading, MA, 1992.
- [Cur89] B. Curtis, „Cognitive issues in reusing software artifacts”, w: T.J. Biggerstaff, A.J. Perlis (red.), *Software Reusability, Volume II: Applications and Experience*, strony 269 – 287, Addison-Wesley, Reading, MA, 1989.
- [dCLF93] D. de Champeaux, D. Lea, P. Faure, *Object-Oriented System Development*, Addison-Wesley, Reading, MA, 1993.
- [Deu89] L.P. Deutsch, „Design reuse and frameworks in the Smalltalk-80 system”, w: T.J. Biggerstaff, A.J. Perlis (red.), *Software Reusability, Volume II: Applications and Experience*, strony 57 – 71, Addison-Wesley, Reading, MA, 1989.
- [Ede92] D.R. Edelson, „Smart pointers: They're smart, but they're not pointers”, w: *Proceedings of the 1992 USENIX C++ Conference*, USENIX Association, strony 1 – 19, Portland, OR, sierpień 1992.
- [EG92] T. Eggenschwiler, E. Gamma, „The ET++SwapsManager: Using object technology in the financial engineering domain”, w: *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, ACM Press, strony 166 – 178, Vancouver, British Columbia, Kanada, październik 1992.

- [ES90] M.A. Ellis, B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, Reading, MA, 1990.
- [Foo92] B. Foote, „A fractal model of the lifecycles of reusable objects”, *OOPSLA '92 Workshop on Reuse*, Vancouver, British Columbia, Kanada, październik 1992.
- [GA89] S. Gossain, D.B. Anderson, „Designing a class hierarchy for domain representation and reusability”, w: *TOOLS '89 Conference Proceedings*, strony 201 – 210, Prentice Hall, CNIT Paris – La Defense, Francja, listopad 1989.
- [Gam91] E. Gamma, *Object-Oriented Software Development based on ET++: Design Patterns, Class Library, Tools* (po niemiecku), rozprawa doktorska, University of Zurich, Institut für Informatik, 1991.
- [Gam92] E. Gamma, *Object-Oriented Software Development based on ET++: Design Patterns, Class Library, Tools* (po niemiecku), Springer-Verlag, Berlin, 1992.
- [Gla90] A. Glassner, *Graphics Gems*, Academic Press, Boston, MA, 1990.
- [GM92] M. Graham, E. Mettala, „The Domain-Specific Software Architecture Program”, w: *Proceedings of DARPA Software Technology Conference*, 1992, strony 204 – 210, kwiecień 1992. Opublikowane też w: *CrossTalk, The Journal of Defense Software Engineering*, strony 19 – 21, 32, październik 1992.
- [GR83] A.J. Goldberg, D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, MA, 1983.
- [HHMV92] R. Helm, T. Huynh, K. Marriott, J. Vlissides, „An object oriented architecture for constraint-based graphical editing”, w: *Proceedings of the Third Eurographics Workshop on Object-Oriented Graphics*, strony 1 – 22, Champéry, Szwajcaria, październik 1992. Dostępne także jako: IBM Research Division Technical Report RC 18524 (79392).
- [HO87] D.C. Halbert, P.D. O'Brien, „Object-oriented development”, *IEEE Software*, 4(5): 71 – 79, wrzesień 1987.
- [ION94] IONA Technologies, Ltd., *Programmer's Guide for Orbix, Version 1.2*, Dublin, Irlandia, 1994.
- [JCJO92] I. Jacobson, M. Christerson, P. Jonsson, G. Overgaard, *Object-Oriented Software Engineering —A Use Case Driven Approach*, Addison-Wesley, Wokingham, Anglia, 1992.
- [JF88] R.E. Johnson, B. Foote, „Designing reusable classes”, *Journal of Object-Oriented Programming*, 1(2): 22 – 35, czerwiec/lipiec 1988.
- [JML92] R.E. Johnson, C. McConnell, J.M. Lake, „The RTL system: A framework for code optimization”, w: R. Giegerich, S.L. Graham (red.), *Code Generation — Concepts, Tools, Techniques. Proceedings of the International Workshop on Code Generation*, strony 255 – 274, Springer-Verlag, Dagstuhl, Niemcy, 1992.
- [Joh92] R. Johnson, „Documenting frameworks using patterns”, w: *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, strony 63 – 76, ACM Press, Vancouver, British Columbia, Kanada, październik 1992.

- [JZ91] R.E. Johnson, J. Zweig, „Delegation in C++”, *Journal of Object-Oriented Programming*, 4(11): 22 – 35, listopad 1991.
- [Kir92] D. Kirk, *Graphics Gems III*, Harcourt, Brace, Jovanovich, Boston, MA, 1992.
- [Knu73] D.E. Knuth, *The Art of Computer Programming, Volumes 1, 2, and 3*, Addison-Wesley, Reading, MA, 1973.
- [Knu84] D.E. Knuth. *The T. Xbook*, Addison-Wesley, Reading, MA, 1984.
- [Kof93] T. Kofler, „Robust iterators in ET++”, *Structured Programming*, 14: 62 – 85, marzec 1993.
- [KP88] G.E. Krasner, S.T. Pope, „A cookbook for using the model-view controller user interface paradigm in Smalltalk-80”, *Journal of Object-Oriented Programming*, 1(3): 26 – 49, sierpień/wrzesień 1988.
- [LaL94] W. LaLonde, *Discovering Smalltalk*, Benjamin/Cummings, Redwood City, CA, 1994.
- [LCI-92] M. Linton, P. Calder, J. Interrante, S. Tang, J. Vlissides, *Interviews Reference Manual, 3.1 edition*, CSL, Stanford University, 1992.
- [Lea88] D. Lea, „libg++ , the GNU C++ library”, w: *Proceedings of the 1988 USENIX C++ Conference*, USENIX Association, strony 243 – 256, Denver, CO, październik 1988.
- [LG86] B. Liskov, J. Guttag, *Abstraction and Specification in Program Development*, McGraw-Hill, Nowy Jork, 1986.
- [Lie85] H. Lieberman, „There's more to menu systems than meets the screen”, w: *SIGGRAPH Computer Graphics*, strony 181 – 189, San Francisco, CA, lipiec 1985.
- [Lie86] H. Lieberman, „Using prototypical objects to implement shared behavior in object-oriented systems”, w: *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, strony 214 – 223, Portland, OR, listopad 1986.
- [Lin92] M.A. Linton, „Encapsulating a C++ library”, w: *Proceedings of the 1992 USENIX C++ Conference*, ACM Press, strony 57 – 66, Portland, OR, sierpień 1992.
- [LP93] M. Linton, C. Price, „Building distributed user interfaces with Fresco”, w: *Proceedings of the 7th X Technical Conference*, strony 77 – 87, Boston, MA, styczeń 1993.
- [LR93] D.C. Lynch, M.T. Rose, *Internet System Handbook*, Addison-Wesley, Reading, MA, 1993.
- [LVC89] M.A. Linton, J.M. Vlissides, P.R. Calder, „Composing user interfaces with Interviews”, *Computer*, 22(2): 8 – 22, luty 1989.
- [Mar91] B. Martin, „The separation of interface and implementation in C++”, w: *Proceedings of the 1991 USLNIX C++ Conference*, USENIX Association, strony 51 – 63, Washington D.C., kwiecień 1991.
- [McC87] P. McCullough, „Transparent forwarding: First steps”, w: *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, ACM Press, strony 331 – 341, Orlando, FL, październik 1987.

- [Mey88] B. Meyer, *Object-Oriented Software Construction*, Series in Computer Science, Prentice Hall, Englewood Cliffs, NJ, 1988.
- [Mur93] R.B. Murray, *C++ Strategies and Tactics*, Addison-Wesley, Reading, MA, 1993.
- [OJ90] W.F. Opdyke, R.E. Johnson, „Refactoring: An aid in designing application frameworks and evolving object-oriented systems”, w: *SOOPPA Conference Proceedings*, ACM Press, strony 145 – 161, Marist College, Poughkeepsie, NY, wrzesień 1990.
- [OJ93] W. F. Opdyke, R. E. Johnson, „Creating abstract superclasses by refactoring”, w: *Proceedings of the 21st Annual Computer Science Conference (ACM CSC '93)*, strony 66 – 73, Indianapolis, IN, luty 1993.
- [P-88] A.J. Palay et al., „The Andrew Toolkit: An overview”, w: *Proceedings of the 1988 Winter USENIX Technical Conference*, USENIX Association, strony 9 – 21, Dallas, TX, luty 1988.
- [Par90] ParcPlace Systems, *ObjectWorks|Smalltalk Release 4 Users Guide*, Mountain View, CA, 1990.
- [Pas86] G.A. Pascoe, „Encapsulators: A new software paradigm in Smalltalk-80”, w: *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, ACM Press, strony 341 – 346, Portland, OR, październik 1986.
- [Pug90] W. Pugh, „Skiplists: A probabilistic alternative to balanced trees”, *Communications of the ACM*, 33(6): 668 – 676, czerwiec 1990.
- [RBP-91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorenson, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.
- [Rum94] J. Rumbaugh, „The life of an object model: How the object model changes during development”, *Journal of Object-Oriented Programming*, 7(1): 24 – 32, marzec/kwiecień 1994.
- [SE84] E. Soloway, K. Ehrlich, „Empirical studies of programming knowledge”, *IEEE Transactions on Software Engineering*, 10(5): 595 – 609, wrzesień 1984.
- [Sha90] Yen-Ping Shan, „MoDE: A UIMS for Smalltalk”, w: *ACM OOPSLA/ECOOP '90 Conference Proceedings*, ACM Press, strony 258 – 268, Ottawa, Ontario, Kanada, październik 1990.
- Sny86] A. Snyder, „Encapsulation and inheritance in object-oriented languages”, w: *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, ACM Press, strony 38 – 45, Portland, OR, listopad 1986.
- SS86] J.C. Spohrer, E. Soloway, „Novice mistakes: Are the folk wisdoms correct?”, *Communications of the ACM*, 29(7): 624 – 632, lipiec 1986.
- SS94] D.C. Schmidt, T. Suda, „The Service Configurator Framework: An extensible architecture for dynamically configuring concurrent, multi-service network daemons”, w: *Proceeding of the Second International Workshop on Configurable Distributed Systems*, IEEE Computer Society, strony 190 – 201, Pittsburgh, PA, marzec 1994.
- Str91] B. Stroustrup, *The C++ Programming Language, Second Edition*, Addison-Wesley, Reading, MA, 1991.

- [Str93] P.S. Strauss, „IRIS Inventor, a 3D graphics toolkit”, w: *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, strony 192 – 200, ACM Press, Washington D.C., wrzesień 1993.
- [Str94] B. Stroustrup, *The Design and Evolution of C++*, Addison-Wesley, Reading, MA, 1994.
- [Sut63] I.E. Sutherland, *Sketchpad: A Man-Machine Graphical Communication System*, rozprawa doktorska, MIT, 1963.
- [Swe85] R.E. Sweet, „The Mesa programming environment”, *SIGPLAN Notices*, 20(7): 216 – 229, lipiec 1985.
- [Sym93a] Symantec Corporation, *Bedrock Developer's Architecture Kit*, Cupertino, CA, 1993.
- [Sym93b] Symantec Corporation, *THINK Class Library Guide*, Cupertino, CA, 1993.
- [Sza92] D. Szafron, „SPECTalk: An object-oriented data specification language”, w: *Technology of Object-Oriented Languages and Systems (TOOLS 8)*, strony 123 – 138, Prentice Hall, Santa Barbara, CA, sierpień 1992.
- [US87] D. Ungar, R.B. Smith, „Self: The power of simplicity”, w: *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, strony 227 – 242, ACM Press, Orlando, FL, październik 1987.
- [VL88] J.M. Vlissides, M.A. Linton, „Applying object-oriented design to structured graphics”, w: *Proceedings of the 1988 USENIX C++ Conference*, strony 81 – 94, USENIX Association, Denver, CO, październik 1988.
- [VL90] J.M. Vlissides, M.A. Linton, „Unidraw: A framework for building domain-specific graphical editors”, *ACM Transactions on Information Systems*, 8(3): 237 – 268, lipiec 1990.
- [WBJ90] R. Wirfs-Brock, R.E. Johnson, „A survey of current research in object-oriented design”, *Communications of the ACM*, 33(9): 104 – 124, 1990.
- [WBWW90] R. Wirfs-Brock, B. Wilkerson, L. Wiener, „*Designing Object-Oriented Software*”, Prentice Hall, Englewood Cliffs, NJ, 1990.
- [WGM88] A. Weinand, E. Gamma, R. Marty, „ET++ — An object oriented application framework in C++”, w: *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, strony 46 – 57, ACM Press, San Diego, CA, wrzesień 1988.

SKOROWIDZ

_instance, 132, 135

A

Abstract Factory, 101
AbstractClass, 266
AbstractExpression, 219
AbstractFactory, 103, 109
Abstraction, 183
AbstractList, 231
AbstractProduct, 103
abstrakcje, 18, 26
abstrakcyjne ujęcie procesu tworzenia obiektów, 59
action, 302
ActionCallback, 310
Ada, 35
Adaptee, 143
Adapter, 22, 141, 143, 213
 Adaptee, 143
 Adapter, 143
 Client, 143
elementy, 143
implementacja, 145, 147
konsekwencje stosowania, 144
powiązane wzorce, 151
przykładowy kod, 147
struktura, 143
Target, 143
uzasadnienie, 141
warunki stosowania, 142
współdziałanie, 143
zastosowanie, 150

adapter dołączalny, 144, 146
adapter dwukierunkowy, 145
adapter klasowy, 139, 142, 144
adapter obiektowy, 143, 144, 149
adapter parametryzowany, 147
Adaptive Communications Environment, 100
AddressTranslation, 168
Aggregate, 232, 233
agregacja, 36
akcja, 302
Alexander, 340
Alexander Christopher, 16
algorytmy, 38
 algorytm formatowania, 53
 algorytm przechodzenia, 233
AlternationExpression, 219, 223
Anderson Bruce, 341
API, 37
aplikacje, 39
aplikacje graficzne, 170
AppKit, 190
Application, 110, 264
ApplicationWindow, 65
architektura MVC, 18, 118, 179
 kontroler, 20
 model, 19
reagowanie widoku na działania użytkownika, 20
widok, 19
zagnieżdżanie widoków, 19
ArrayCompositor, 322
ArrayIterator, 77
ASCII, 92

ASCII7Stream, 160
 ASCIIConverter, 92
 automatyczne przekazywanie, 249

B

Beck Kent, 342
 Bedrock, 156
 BinaryExpression, 180
 black-box reuse, 32
 BNF, 221
 BooleanExp, 225
 Border, 56
 BorderDecorator, 153
 BorderedComposition, 56
 brak elastyczności, 90
 Bridge, 181
 BTee, 208
 Budowniczy, 22, 92, 138
 Builder, 93
 ConcreteBuilder, 93
 Director, 93, 94
 elementy, 93
 implementacja, 95
 konsekwencje stosowania, 94
 powiązane wzorce, 100
 Product, 94
 przykładowy kod, 96
 relacje, 93
 Smalltalk-80, 100
 uzasadnienie, 92
 warunki stosowania, 93
 współdziałanie, 94
 zastosowanie, 100
 Builder, 92, 93
 Bytecode, 164
 BytecodeStream, 100, 161, 164

C

C++, 18
 dziedziczenie, 30
 kontrola dostępu, 267
 przeciążanie operatora dostępu do składowych, 195
 szablony, 35
 this, 33
 Caretaker, 296
 CASE, 337
 centralizacja sterowania, 258
 Chain of Responsibility, 244
 ChangeManager, 262, 274, 275
 Choices, 168
 CISCScheduler, 328
 ClassBuilder, 100
 Client, 103, 122, 143, 172, 205, 220, 246, 305
 Clone, 124
 CLOS, 18
 cofanie działań, 294, 304, 306
 Colleague, 257
 Command, 71, 72, 253, 302, 305
 Component, 154, 156, 172, 174, 179, 253
 ComponentView, 253
 Composite, 170, 172, 179
 CompositeView, 19
 Composition, 53, 326, 327
 Compositor, 53, 326
 CompressingStream, 160
 ConcreteAggregate, 232
 ConcreteBuilder, 93
 ConcreteClass, 266
 ConcreteCommand, 305
 ConcreteComponent, 154
 ConcreteCreator, 111, 115
 ConcreteDecorator, 154, 155
 ConcreteElement, 283, 284, 288
 ConcreteFactory, 103
 ConcreteFlyweight, 204
 ConcreteHandler, 246
 ConcreteImplementor, 183
 ConcreteIterator, 232
 ConcreteMediator, 257
 ConcreteObserver, 271
 ConcreteProduct, 103, 111
 ConcretePrototype, 122
 ConcreteState, 313
 ConcreteStrategy, 323
 ConcreteSubject, 271
 ConcreteVisitor, 283, 288
 ConstraintSolver, 294, 298
 ConstraintSolverMemento, 298
 Context, 220, 313, 323
 Controller, 18, 20
 CreateIterator, 231
 CreateMaze(), 117
 Creator, 111
 CSolver, 300
 cursor, 230
 cykl życia oprogramowania obiektowego, 337
 czytnik dokumentów RTF, 92

D

DebuggerAdaptor, 128
 DebuggingGlyph, 159
 decentralizacja logiki zmian stanów, 315
 Decorator, 152, 154
 defaultController(), 118
 defaultControllerClass(), 118
 definiowanie rozszerzalnych fabryk, 105
 dekorator, 152
 Dekorator, 22, 58, 152, 213
 Component, 154
 ConcreteComponent, 154
 ConcreteDecorator, 154
 Decorator, 154
 elementy, 154
 implementacja, 155
 konsekwencje stosowania, 155
 powiązane wzorce, 160
 przykładowy kod, 157
 uzasadnienie, 152
 warunki stosowania, 154
 współdziałanie, 154
 zastosowanie, 159
 delegat, 33, 146
 delegowanie, 33, 34
 dependents, 269
 DialogDirector, 259
 DialogKit, 109
 DialogWindow, 65
 Director, 93, 94
 Display, 268
 dobre projekty, 15
 Document, 110, 264
 dodawanie operacji, 284
 dodawanie produktów w czasie wykonywania programu, 122
 doesNotUnderstand, 196, 199, 249
 dokumentowanie, 336
 dokumentowanie platformy, 41
 dokumenty RTF, 92
 Domain, 168
 dostęp do rozproszonych informacji, 75
 double dispatch, 287
 DrawingApplication, 110
 DrawingController, 320
 DrawingDocument, 110
 drzewo składni abstrakcyjnej, 221, 280
 dwukrotna dyspozycja, 287

Dylan, 300
 dynamiczna konfiguracja aplikacji, 123
 dyspozycja
 dwukrotna, 287
 jednokrotna, 287
 wielokrotna, 287
 dziedziczenie, 27, 29, 215
 C++, 30
 Eiffel, 30
 interfejsy, 30
 klasy, 30, 31
 klasy abstrakcyjne, 29
 klasy nadzędne, 29
 podklasy, 29
 przesłanianie metod, 29
 Smalltalk, 30
 stosowanie, 31
 typy spараметryzowane, 35
 dziedziczenie dynamiczne, 316

E

edytor dokumentów, 45
 ApplicationWindow, 65
 Border, 56
 BorderedComposition, 56
 cofanie operacji, 72
 Command, 71, 72
 Composition, 53
 Compositor, 53
 Dekorator, 58
 DialogWindow, 65
 dostęp do rozproszonych informacji, 75
 działania użytkowników, 69
 Fabryka abstrakcyjna, 62
 fabryki, 60
 formatowanie, 46, 52
 glify-widgety, 60
 Glyph, 50, 59, 76
 GUIFactory, 60
 historia poleceń, 73
 IconWindow, 65
 interfejs użytkownika, 46
 Iterator, 77
 kapsułkowanie algorytmu formatowania, 52
 kapsułkowanie analiz, 81
 kapsułkowanie dostępu do danych, 75
 kapsułkowanie zależności implementacyjnych, 64
 kapsułkowanie żądania, 70

- edytor dokumentów
 klasy produktów, 60
 Kompozyt, 52
 konfiguracja obiektów WINDOW, 68
 MenuItem, 70
 MonoGlyph, 56
 MotifFactory, 60
 MotifScrollBar, 59, 60
 obsługa wielu standardów wyglądu i działania, 59
 obsługa wielu systemów okienkowych, 63
 operacje, 69
 ozdabianie interfejsu użytkownika, 46, 55
 PMScrollBar, 59
 podział słów, 74
 Polecenie, 74
 powtarzanie operacji, 72
 problemy projektowe, 45
 produkty, 61
 przechodzenie po elementach, 80
 Rectangle, 51
 ScrollBar, 59
 Scroller, 57
 SpellingChecker, 81, 82, 83
 sprawdzanie pisowni, 74
 Strategia, 55
 struktura dokumentu, 45, 47
 Visitor, 84
 Window, 51, 64, 66
 WindowImp, 66
 edytor graficzny, 141
 egzemplarz klasy, 28
 Eiffel, 35
 elastyczność projektu, 90
 Element, 283
 Encapsulator, 200
 Equipment, 289
 ET++, 118, 253
 ET++SwapsManager, 328
 Etgdb, 128
 Execute, 302, 303
- F**
- Fabryka abstrakcyjna, 22, 62, 63, 101, 137
 AbstractFactory, 103
 AbstractProduct, 103
 Client, 103
 ConcreteFactory, 103
 ConcreteProduct, 103
- elementy, 103
 implementacja, 104
 konsekwencje stosowania, 103
 powiązane wzorce, 109
 przykładowy kod, 106
 uzasadnienie, 101
 warunki stosowania, 102
 współdziałanie, 103
 zastosowanie, 109
- Facade, 161, 163, 167
 Factory Method, 110
 Fasada, 22, 161
 elementy, 163
 Facade, 163
 implementacja, 164
 klasy podsystemu, 163
 konsekwencje stosowania, 163
 powiązane wzorce, 169
 przykładowy kod, 164
 uzasadnienie, 161
 warunki stosowania, 162
 współdziałanie, 163
 zastosowania, 167
- FileStream, 159
 FileSystemInterface, 168
 Flyweight, 201, 204
 FlyweightFactory, 205, 206, 210
 FontDialogDirector, 255
 Foote Brian, 337
 format danych
 ASCII, 92
 RTF, 92
 framework, 40
 funkcje zwrotne, 304
 funktry, 311
- G**
- GenerateCode, 287
 Glyph, 50, 76, 206
 GlyphContext, 208
 głębokie kopiowanie, 124
 gra, 88
 gramatyka, 217, 220
 Graphic, 120, 197
 GraphicTool, 120, 137
 GUIFactory, 60

H

Handle, 189
 handle/body, 181
 HandleHelp, 245
 Handler, 246, 250, 252
 HelpHandler, 245, 250
 historia poleceń, 73, 307
 HotDraw, 320

I

IconWindow, 65, 182
 ImageProxy, 192, 198
 implementacja gramatyki, 220
 implementacja obiektu, 28
 Implementor, 183, 184
 inicjowanie klonów, 125
 Initialize, 125
 InspectClass, 167
 InspectObject, 167
 instrukcje warunkowe, 323
 Instrument, 328
 inteligentne referencje, 193, 194
 inteligentne wskaźniki, 193
 interfejs użytkownika, 101
 interfejsy, 27

 dziedziczenie, 30

Interpret, 221

Interpreter, 22, 217

 AbstractExpression, 219

 Client, 220

 Context, 220

 elementy, 219

 implementacja, 221

 konsekwencje stosowania, 220

 NonterminalExpression, 220

 powiązane wzorce, 229

 przykładowy kod, 222

 TerminalExpression, 220

 uzasadnienie, 217

 warunki stosowania, 219

 współdziałanie, 220

 zastosowanie, 228

InterViews, 109, 136, 150, 159, 278

InvalidateRect, 253

InventoryVisitor, 291

Invoker, 305

IRIS Inventor, 292

iteracja polimorficzna, 231
 IterationState, 300
 Iterator, 22, 77, 80, 230, 234, 330
 Aggregate, 232, 233
 ConcreteAggregate, 232
 ConcreteIterator, 232
 elementy, 232
 implementacja, 233
 Iterator, 232
 konsekwencje stosowania, 233
 powiązane wzorce, 243
 przykładowy kod, 236
 uzasadnienie, 230
 warunki stosowania, 232
 współdziałanie, 232
 zastosowanie, 243
 iteratory, 77, 230
 aktywny, 233
 pasywny, 233
 polimorficzny, 234
 pusty, 235
 wewnętrzny, 233, 240
 zewnętrzny, 233
 izolowanie klas konkretnych, 103

J

jawne referencje do elementu nadzawanego, 173
 język programowania, 18
 C++, 18
 Dylan, 300
 Smalltalk-80, 18

K

kapsułkowanie, 26
 algorytm formatowania, 52
 dostęp do danych, 75
 komunikacja między obiektami, 331
 zależności implementacyjne, 64
 złożona semantyka aktualizacji, 274
 zmiany, 330
 żądania, 70
 katalog wzorców projektowych, 22
 kategoria wzorca, 20
 KernelProxy, 192
 kit, 101
 klasowe wzorce operacyjne, 215
 klasowe wzorce strukturalne, 139

klasy, 28
 dziedziczenie, 29, 30
 klasy abstrakcyjne, 29, 50
 klasy konkretne, 29
 klasy mieszane, 29
 klasy nadrzędne, 29

klient, 26
 klonowanie obiektów, 120, 124
 Knuth Donald, 341
 kod wielokrotnego użytku, 15
 kolejkowanie żądania, 304
 kompilator, 228, 280
 komponenty, 56
 Kompozyt, 22, 52, 170, 213

Client, 172
 Component, 172
 Composite, 172
 elementy, 172
 implementacja, 173
 konsekwencje stosowania, 173
 Leaf, 172
 powiązane wzorce, 180
 przykładowy kod, 177
 uzasadnienie, 170
 warunki stosowania, 171
 współdziałanie, 172
 zastosowanie, 179

kompresja danych ze strumienia, 159
 komunikacja, 331
 komunikaty, 26
 konsekwencje stosowania wzorca projektowego, 17
 konserwowanie złożonych gramatyk, 221
 konsolidowanie, 337
 Konstruktor wirtualny, 110
 kontrola dostępu do jedynego egzemplarza, 131
 kontrola dostępu do obiektu, 191
 kontroler, 18, 20
 konwencje nazewnicze, 116, 267
 konwersja dokumentów RTF, 100
 kopiowanie przy zapisie, 194
 kurSOR, 230, 233

L

labirynt, 88, 106
 LALR, 100
 Layout, 211, 212
 Leaf, 172
 leniwe inicjowanie, 115

libg++, 189
 limit liczby egzemplarzy, 131
 List, 35, 236
 lista rozwijana, 254
 ListIterator, 77, 230, 240
 LiteralExpression, 219
 Look, 212
 luźne powiązanie, 38

Ł

Łańcuch, 333
 łańcuch następników, 247
 Łańcuch zobowiązań, 22, 244
 Client, 246
 ConcreteHandler, 246
 elementy, 246
 Handler, 246
 implementacja, 247
 konsekwencje stosowania, 247
 powiązane wzorce, 253
 przykładowy kod, 250
 uzasadnienie, 244
 warunki stosowania, 246
 współdziałanie, 247
 zastosowanie, 252
 łączenie następników, 248

M

MacApp, 116, 118, 156
 MacroCommand, 306, 307, 309
 MapSite, 127
 MazeFactory, 106
 mechanizm powtórnego wykorzystania rozwiązania, 32
 Mediator, 23, 254, 330, 333
 ConcreteMediator, 257
 elementy, 257
 implementacja, 258
 konsekwencje stosowania, 258
 Mediator, 257
 powiązane wzorce, 263
 przykładowy kod, 259
 uzasadnienie, 254
 warunki stosowania, 256
 współdziałanie, 257
 zastosowanie, 261

Memento, 294, 296
 MemoryObject, 168
 MemoryObjectCache, 168
 MemoryStream, 159
 menedżer prototypów, 124
 MenubarLayout, 211
 MenuItem, 70
 metaklasy, 136
 Metoda szablonowa, 23, 264

 AbstractClass, 266
 ConcreteClass, 266
 elementy, 266
 implementacja, 267
 konsekwencje stosowania, 266
 powiązane wzorce, 268
 przykładowy kod, 267
 uzasadnienie, 264
 warunki stosowania, 265
 współdziałanie, 266
 zastosowanie, 268

Metoda wytwórcza, 20, 23, 110, 137

 ConcreteCreator, 111
 ConcreteProduct, 111
 Creator, 111
 elementy, 111
 implementacja, 113
 konsekwencje stosowania, 112
 powiązane wzorce, 119
 Product, 111
 przykładowy kod, 117
 uzasadnienie, 110
 warunki stosowania, 111
 współdziałanie, 112
 zastosowanie, 118

metodologie projektowania obiektowego, 26

metody, 26

metody projektowania obiektowego, 337

mixin class, 29

Mode Composer, 129

model, 18

Model, 18, 278

model publikuj-subskrybuj, 270

model wyciągania, 274

model wypychania, 274

Model/View/Controller, 18

MonoGlyph, 56

Most, 23, 69, 181, 213

 Abstraction, 183

 ConcreteImplementor, 183

elementy, 183
 implementacja, 184
 Implementor, 183
 konsekwencje stosowania, 184
 powiązane wzorce, 190
 przykładowy kod, 185
 RefinedAbstraction, 183
 uzasadnienie, 181
 warunki stosowania, 182
 współdziałanie, 184
 zastosowanie, 189
 Motif, 60, 101
 MotifFactory, 60
 MotifScrollBar, 59, 60
 MotifWidgetFactory, 102
 MVC, 18, 278

N

nadtypy, 27
 Nakładka, 141, 152
 nazwa wzorca, 17, 20
 nazwy klas abstrakcyjnych, 29
 niejawny odbiorca żądań, 245
 nieoczekiwane aktualizacje, 272
 niepowtarzalność egzemplarza, 131
 niewidoczna otoczka, 56
 niezawodny iterator, 234
 NodeVisitor, 281
 NonterminalExpression, 220
 notacja BNF, 221
 notacja OMT, 21, 28
 NullIterator, 78, 235
 NXImage, 190
 NXImagRep, 190
 NXProxy, 192, 200

O

obiekt fasadowy, 161
 obiektowe wzorce operacyjne, 215
 obiektowy język programowania, 18
 obiekty, 26
 interfejsy, 27
 poziom szczegółowości, 27
 składanie, 33
 obiekty jako argumenty, 330
 obiekty stanów, 312
 obiekty zależne, 269

obiekty złożone, 19
 objects for states, 312
 ObjectStructure, 283
 ObjectWindows, 243, 329
 Observer, 269, 271, 272
 Obserwator, 23, 269, 331, 332
 aspekty obiektów Subject, 274
 ConcreteObserver, 271
 ConcreteSubject, 271
 elementy, 270
 implementacja, 272
 konsekwencje stosowania, 272
 Observer, 271
 obserwator, 270
 podmiot, 270
 powiązane wzorce, 279
 przykładowy kod, 276
 publikuj-subskrybij, 270
 Subject, 270
 uzasadnienie, 269
 warunki stosowania, 270
 współdziałanie, 271
 zastosowanie, 278
 obsługa cofania działań, 294
 obsługa rozsyłania grupowego komunikatów, 272
 obsługa wielu standardów wyglądu i działania, 59
 obsługa wielu systemów okienkowych, 63
 oddzielanie interfejsu od implementacji, 184
 oddzielanie nadawców od odbiorców, 332
 odporność na zmiany, 37
 Odwiedzający, 23, 85, 280
 ConcreteElement, 283, 284
 ConcreteVisitor, 283
 Element, 283
 elementy, 283
 implementacja, 285
 konsekwencje stosowania, 284
 ObjectStructure, 283
 powiązane wzorce, 293
 przykładowy kod, 288
 uzasadnienie, 280
 Visitor, 283
 warunki stosowania, 282
 współdziałanie, 283
 zastosowanie, 292
 ograniczanie tworzenia podklas, 258
 określanie implementacji obiektów, 28
 określanie nowych obiektów
 przez modyfikowanie struktury, 123
 przez zmianę wartości, 122

określanie poziomu szczegółowości obiektu, 27
 OMT, 21, 28
 OpenCommand, 307
 OpenDocument, 265
 operacje, 26
 abstrakcyjne operacje, 29
 operator dostępu do składowych, 195
 opis problemu, 17
 opis wzorców projektowych, 20
 oprogramowanie obiektowe, 335
 Orbix ORB, 119
 Originator, 296
 otoczka, 56
 otwarte powtórne wykorzystanie, 32
 ozdabianie interfejsu użytkownika, 55

P

pakiety narzędziowe, 39
 Pamiątka, 23, 294
 Caretaker, 296
 elementy, 296
 implementacja, 297
 konsekwencje stosowania, 297
 Memento, 296
 Originator, 296
 powiązane wzorce, 301
 przykładowy kod, 298
 uzasadnienie, 294
 warunki stosowania, 295
 współdziałanie, 296
 zastosowanie, 300
 źródło, 295
 pamięć podrzczna, 176
 Pane, 262
 parametry, 35
 parametryzacja obiektów, 304
 parametryzacja systemu, 137
 Parser, 100, 161, 164
 parserClass(), 118
 PassivityWrapper, 159
 PasteCommand, 307
 Pełnomocnik, 23, 191, 214
 elementy, 193
 implementacja, 195
 konsekwencje stosowania, 194
 powiązane wzorce, 200
 Proxy, 193
 przykładowy kod, 197

- RealSubject, 194
- Subject, 194
- uzasadnienie, 191
- warunki stosowania, 192
- współdziałanie, 194
- zastosowanie, 200
- pełnomocnik wirtualny, 192, 194, 200
- pełnomocnik zabezpieczający, 194
- platforma, 40
- pluggable adapter, 144
- PluggableAdapter, 150
- płytka kopia, 124
- PMIconWindow, 181
- PMScrollBar, 59
- PMWindow, 181
- podklasy, 29, 38, 123
- podklasy klasy Singleton, 133
- podsystem kompilujący, 161
- podtypy, 27
- podział słów, 74
- podział strumienia tekstu na wiersze, 321
- Polecenie, 23, 74, 302, 332
 - Client, 305
 - Command, 305
 - ConcreteCommand, 305
 - elementy, 305
 - implementacja, 306
 - Invoker, 305
 - konsekwencje stosowania, 306
 - powiązane wzorce, 311
 - przykładowy kod, 307
- Receiver, 305
- uzasadnienie, 302
- warunki stosowania, 304
- współdziałanie, 305
- zastosowania, 310
- policy, 321
- polimorfizm, 27
- polityka, 321
- połączenie równoległych hierarchii klas, 112
- połączenie sieciowe, 312
 - TCP, 316
- pomijanie klasy abstrakcyjnej, 155
- porządkowanie wzorców, 25
- PostorderIterator, 77
- pośrednik
 - wirtualny, 194, 197
 - zabezpieczający, 192
 - zdalny, 194
- powtarzanie operacji, 306
- powtórne wykorzystanie, 32, 37
 - kod, 39, 266
 - projekt, 40
- poziom szczegółowości obiektu, 27
- PreorderIterator, 77
- Presentation Manager, 101, 181
- PricingVisitor, 290
- problem, 17
- problemy specyficzne dla języka, 115
- Product, 94, 111
- produkty, 61
- ProgrammingEnvironment, 167
- ProgramNode, 161
- ProgramNodeBuilder, 161
- ProgramNodeEnumerator, 292
- programowanie pod kątem interfejsu, 31
- programy obiektowe, 26
- projekt obiektowy, 15
- projekt platformy, 41
- projektowanie
 - aplikacje, 40
 - edytor dokumentów, 45
 - obiektowe, 26, 31, 33
 - oprogramowanie obiektowe, 15
 - pod kątem zmian, 37
- prototyp, 120
- Prototyp, 23, 104, 105, 120, 138
 - Client, 122
 - ConcretePrototype, 122
 - elementy, 122
 - implementacja, 124
 - konsekwencje zastosowania, 122
 - powiązane wzorce, 129
 - Prototype, 122
 - przykładowy kod, 125
 - uzasadnienie, 120
 - warunki stosowania, 121
 - zastosowanie, 128
- Prototype, 120, 122
- Proxy, 191, 193
- przeciążanie operatora dostępu do składowych, 195
- przesłanianie metod, 29
- przestrzeń nazw, 131
- przestrzeń wzorców projektowych, 24
- przyrostowe zmiany, 298
- publikuj-subskrybij, 269, 270
- publish-subscribe, 269
- punkty zaczepienia, 266
 - podklasy, 112

Pyłek, 23, 201
 Client, 205
 ConcreteFlyweight, 204
 elementy, 204
 Flyweight, 204
 FlyweightFactory, 205
 implementacja, 206
 konsekwencje stosowania, 205
 powiązane wzorce, 212
 przykładowy kod, 206
 UnsharedConcreteFlyweight, 205
 uzasadnienie, 201
 warunki stosowania, 203
 współdziałanie, 205
 zastosowanie, 211

Q

QOCA, 145, 228
 Queue, 243

R

RApp, 329
 ReadStream, 243
 RealSubject, 194
 Receiver, 305
 Rectangle, 51
 refaktoryzacja, 337
 RefinedAbstraction, 183
 RegisterAllocator, 328
 RegisterTransfer, 180
 RegularExpression, 217, 224
 rejestr singletonów, 133
 rejestrowanie zmian, 304
 rekurencyjne składanie, 48
 relacje między wzorcami projektowymi, 25
 RepetitionExpression, 219, 223
 reprezentowanie żądań, 248
 Request, 248
 Rich Text Format, 92
 RISCscheduler, 328
 rozmiar obiektu, 191
 rozsyłanie grupowe komunikatów, 272
 rozszerzanie gramatyki, 220
 rozwiązywanie, 17
 rozwijanie, 337
 RTF, 92
 RTFReader, 92

RTL, 180
 RTL System, 328
 RTLExpression, 180

S

Scanner, 161, 164
 ScrollBar, 59
 ScrollbarLayout, 211
 ScrollDecorator, 153
 Scroller, 57
 self, 33
 Self, 124, 316
 SequenceExpression, 223
 Service Configurator, 100
 Session, 136
 SimpleCompositor, 321
 singleton, 130
 Singleton, 23, 91, 104, 131
 elementy, 131
 implementacja, 131
 konsekwencje stosowania, 131
 powiązane wzorce, 136
 przykładowy kod, 135
 uzasadnienie, 130
 warunki stosowania, 130
 zastosowanie, 136
 Sketchpad, 128
 składanie obiektów, 32, 33, 56, 137, 149, 215
 składanie rekurencyjne, 48
 słownictwo projektowe, 336
 Smalltalk-80, 18
 automatyczne przekazywanie, 249
 Budowniczy, 100
 dziedziczenie, 30
 self, 33
 sparametryzowane metody wytwórcze, 113
 SPECTalk, 228
 SpellingChecker, 81, 82, 83
 społeczność związana ze wzorcami, 340
 spójność między produktami, 104
 SSA, 180
 Stan, 23, 312
 ConcreteState, 313
 Context, 313
 elementy, 313
 implementacja, 315
 konsekwencje stosowania, 314
 powiązane wzorce, 320

przykładowy kod, 316
 State, 313
 uzasadnienie, 312
 warunki stosowania, 313
 współdziałanie, 313
 zastosowania, 319
 stan wewnętrzny, 201
 stan zewnętrzny, 201
 State, 312, 313, 330
 statyczne dziedziczenie, 155
 stosowanie wzorca projektowego, 43
Strategia, 20, 23, 55, 321
 ConcreteStrategy, 323
 Context, 323
 elementy, 322
 implementacja, 324
 konsekwencje stosowania, 323
 powiązane wzorce, 329
 przykładowy kod, 326
 Strategy, 322
 uzasadnienie, 321
 warunki stosowania, 322
 współdziałanie, 323
 zastosowanie, 328
Strategy, 321, 322, 330
StreamDecorator, 160
 strukturalne wzorce obiektowe, 139
 struktury projektowe, 18
 strumienie, 159
 Subject, 194, 270, 272
 Substytut, 191
 SunDbxAdaptor, 128
 SunWindowPort, 189
 surrogate, 191
 sygnatura, 27
 system odporny na zmiany, 37
 system pomocy w graficznym interfejsie użytkownika, 244
 szablony, 35, 116, 307

§

ścisłe powiązanie, 38
 środowiska okienkowe, 63

T

TableAdaptor, 151
 tablice, 315
 Target, 143

TCP, 316
 TCPConnection, 312, 316
 Template method, 264
 TerminalExpression, 220
 TeXCompositor, 322
 TextConverter, 92
 ThingLab, 128
 THINK, 310
 this, 33
 token, 294
 Tool, 320
 transaction, 302
 transakcja, 302
 TransientWindow, 182
 transparent enclosure, 56
 TreeDisplay, 146
 tworzenie
 egzemplarze klasy, 28
 podtypy, 30
 produkty, 104
 prototypy, 337
 TypeCheck, 281
 typy danych, 27
 typy generyczne, 35
 typy sparametryzowane, 35

U

Uchwyty/ciało, 181
 ukrywanie szczegółów implementacji przed klientami, 184
 Unidraw, 114, 145, 300, 310, 320
 UnsharedConcreteFlyweight, 205
 uporządkowanie elementów podległych, 176
 usuwanie produktów w czasie wykonywania programu, 122
 usuwanie stanu zewnętrznego, 206

V

Validator, 329
 ValueModel, 150
 View, 18, 179
 ViewManager, 262
 virtual constructor, 110
 Visitor, 84, 280, 283, 330
 VisualComponent, 153
 VObject, 179

W

- wewnętrzna reprezentacja produktu, 94
 white-box reuse, 32
 wiązanie dynamiczne, 27
 WidgetFactory, 101, 102
 WidgetKit, 109, 136
 widok, 18
 wielodziedziczenie, 143, 148, 185
 wielokrotny użytk, 15
 Window, 51, 64, 66, 181, 182, 185
 WindowImp, 66, 182, 185
 WindowPort, 189
 wiszące referencje do usuniętych podmiotów, 273
 wrapper, 141, 152
 współużytkowanie komponentów, 173
 współużytkowanie symboli końcowych, 221
 wyrażenia regularne, 217, 222
 WYSIWYG, 45
 wyszukiwanie liniowe, 90
 wyszukiwanie wzorców, 42
 wywoływanie żądania, 304
 wzorce konstrukcyjne, 24, 87
 - Abstract Factory, 101
 - Budowniczy, 92
 - Builder, 92
 - Fabryka abstrakcyjna, 101
 - Metoda twórcza, 110
 - Prototyp, 120
 - Prototype, 120
 - Singleton, 91, 130
 wzorce operacyjne, 24, 215
 - Chain of Responsibility, 244
 - Command, 302
 - Interpreter, 217
 - Iterator, 230
 - kapsułkowanie zmian, 330
 - Łańcuch zobowiązań, 244
 - Mediator, 254
 - Memento, 294
 - Metoda szablonowa, 264
 - obiekty jako argumenty, 330
 - Observer, 269
 - Obserwator, 269
 - oddzielanie nadawców od odbiorców, 332
 - Odwiedzający, 280
 - Pamiątka, 294
 - Polecenie, 302
 - Stan, 312
 State, 312
 Strategia, 321
 Strategy, 321
 Template method, 264
 Visitor, 280
 wzorce klasowe, 215
 wzorce obiektowe, 215
 wzorce projektowe, 16, 17
 - abstrakcje, 26
 - Adapter, 22, 141
 - architektura MVC, 18
 - Budowniczy, 22, 92
 - Dekorator, 22, 58, 152
 - elementy, 17
 - Fabryka abstrakcyjna, 22, 62, 63, 101
 - Fasada, 22, 161
 - graficzna reprezentacja klas, 21
 - implementacja, 21
 - Interpreter, 22, 217
 - Iterator, 22, 80, 230
 - katalog, 22
 - kategoria, 20
 - Kompozyt, 22, 52, 170
 - konsekwencje, 17, 21
 - Łańcuch zobowiązań, 22, 244
 - Mediator, 23, 254
 - Metoda szablonowa, 23, 264
 - Metoda twórcza, 23, 110
 - Most, 23, 69, 181
 - nazwa, 17, 20
 - Obserwator, 23, 269
 - Odwiedzający, 23, 85, 280
 - opis, 17, 20
 - Pamiątka, 23, 294
 - Pełnomocnik, 23, 191
 - Polecenie, 23, 74, 302
 - Prototyp, 23, 104, 120
 - przeznaczenie, 20
 - Pyłek, 23, 201
 - relacje między wzorcami, 25
 - rozwiązywanie, 17
 - Singleton, 23, 130
 - Stan, 23, 312
 - stosowanie, 43
 - Strategia, 23, 55, 321
 - struktura, 21
 - warunki stosowania, 21
 - współdziałanie, 21
 - wybór, 42
 - zasięg, 24

wzorce strukturalne, 24, 139
 Adapter, 141
 Bridge, 181
 Composite, 170
 Decorator, 152
 Dekorator, 152
 Facade, 161
 Fasada, 161
 Flyweight, 201
 klasowe wzorce, 139
 Kompozyt, 170
 Most, 181
 obiektowe wzorce, 139
 Pełnomocnik, 191
 Proxy, 191
 Pyłek, 201

X

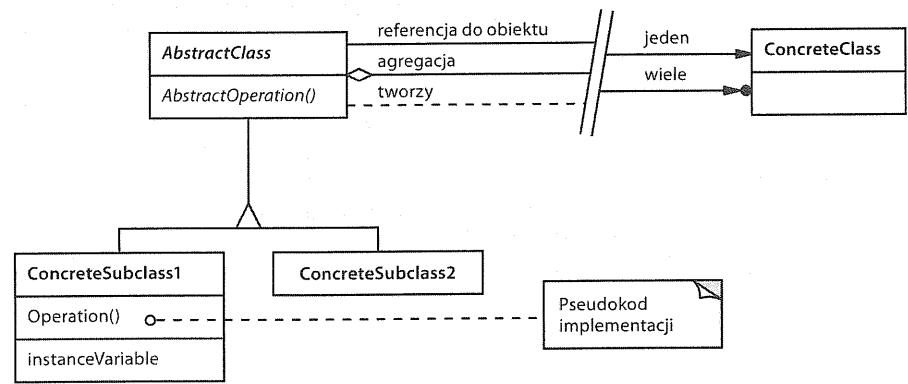
X Window, 181
 XIconWindow, 181
 XWindow, 181
 XWindowPort, 189

Y

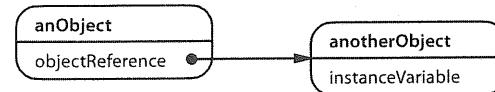
YieldCurve, 328

Z, Ź, Ż

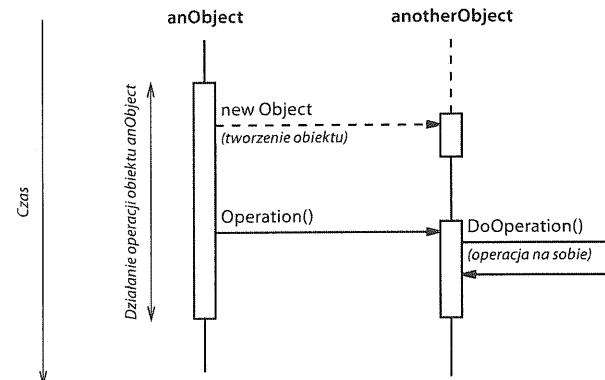
zagłędzanie widoków, 19
 zależność od algorytmów, 38
 zależność od platformy sprzętowej lub programowej, 37
 zależność od reprezentacji lub implementacji obiektu, 38
 zależność od specyficznych operacji, 37
 zamknięte powtórne wykorzystanie, 32
 zapisywane przyrostowych zmian, 298
 zarządzanie pamięcią wirtualną, 168
 zarządzanie współużytkowanymi obiekty, 206
 zasięg wzorca, 24
 zdalny pełnomocnik, 192
 Zestaw, 101
 zliczanie referencji, 193
 zmiany, 37, 304, 330
 zmiany stanu, 314
 zmienne egzemplarza, 28
 zmniejszanie złożoności systemu, 161
 znacznik, 294
 znajomość obiektów, 36
 związki między strukturami czasu wykonywania programu i strukturami czasu komplikacji, 36
 źródło pamiętki, 295
 żądanie, 26, 248



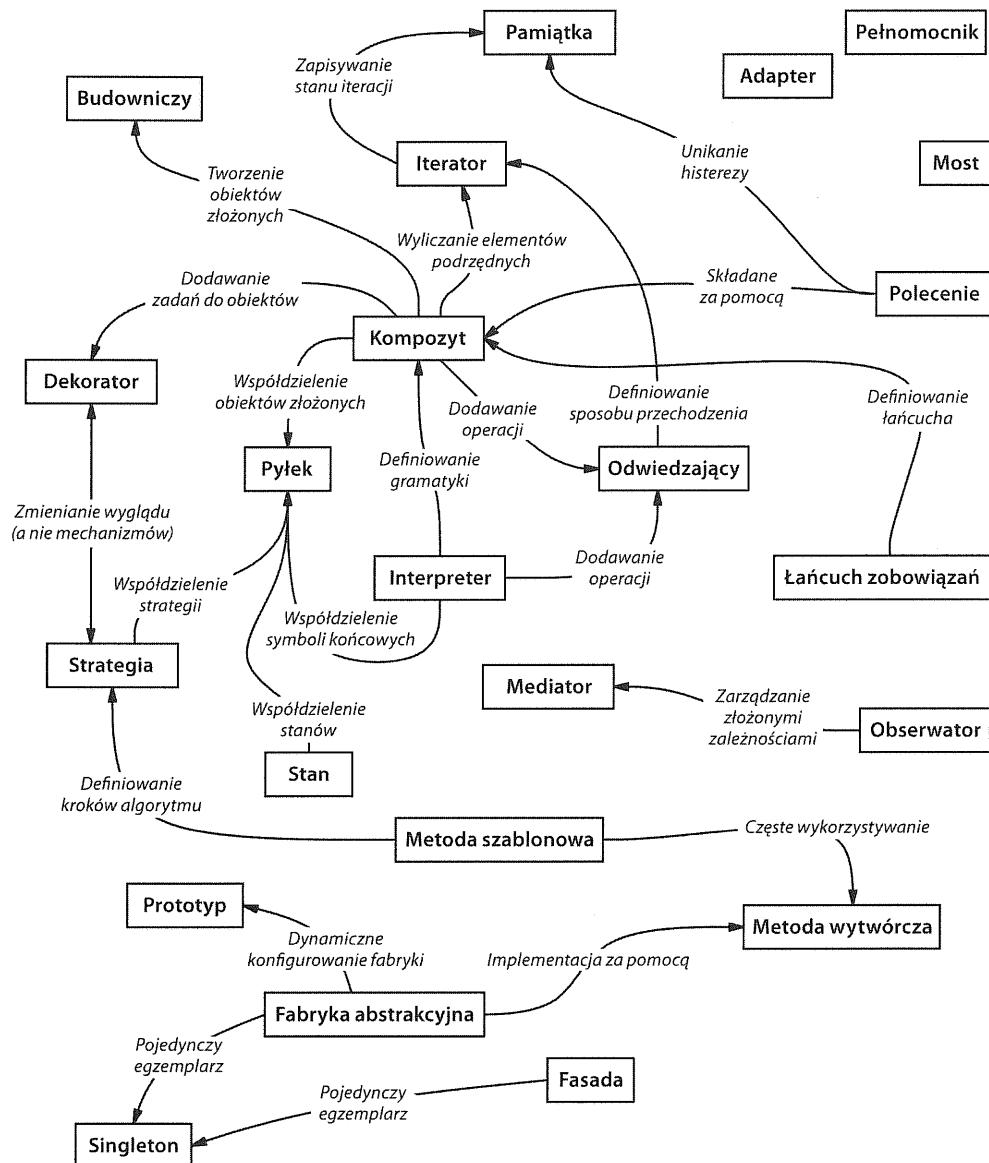
NOTACJA DIAGRAMÓW KLAS



NOTACJA DIAGRAMÓW OBIEKTÓW



NOTACJA DIAGRAMÓW INTERAKCJI



RELACJE MIEDZYZ WZORCAMI PROJEKTOWYMI