Mariusz Trzaska

Modelowanie i implementacja systemów informatycznych

Zrozum zależności pomiędzy analizą wymagań, projektowaniem oraz implementacją oprogramowania!

Mariusz Trzaska

Modelowanie i implementacja systemów informatycznych

Warszawa 2011

Drukowaną wersję tej książki można zakupić w:

Wydawnictwo PJWSTK
ul. Koszykowa 86, 02-008 Warszawa
tel. : 0 (22) 58 44 526
e-mail: oficyna@pjwstk.edu.pl
https://ssl.pjwstk.edu.pl/sklep/

Adres autora:
Polsko-Japońska Wyższa Szkoła Technik Komputerowych
ul. Koszykowa 86,
02-008 Warszawa
mtrzaska@mtrzaska.com

Korekta
Anna Bittner

Komputerowy skład tekstu
Mariusz Trzaska

ISBN: 978-83-272-3169-7

Notka biograficzna

Dr inż. Mariusz Trzaska jest adiunktem w Polsko-Japońskiej Wyższej Szkole Technik Komputerowych, gdzie zajmuje się działalnością dydaktyczną oraz naukową. Oprócz tego bierze udział w różnych projektach naukowych i komercyjnych. Prowadzi także szkolenia oraz warsztaty. Jego zainteresowania zawodowe obejmuja inżynierię oprogramowania, bazy danych, graficzne interfejsy użytkownika, systemy rozproszone oraz technologie internetowe. Wyniki badań z wyżej wymienionych dziedzin publikuje w kraju i zagranicą.

Streszczenie

Książka poświęcona jest problematyce wytwarzania oprogramowania z wykorzystaniem podejścia obiektowego i notacji UML. Szczególny nacisk położono na przełożenie teoretycznych pojęć obiektowości na praktyczne odpowiedniki implementacyjne. Na konkretnym, biznesowym przykładzie (wypożyczalnia wideo) opisano poszczególne fazy wytwarzania oprogramowania: analiza, projekt, implementacja, testowanie ze szczególnym uwzględnieniem tych dwóch środkowych. Opis poparto implementacją biblioteki (dla języka Java) ułatwiającej stosowanie niektórych pojęć obiektowości (ekstensja, asocjacje, ograniczenia, dziedziczenie) oraz prototypem częściowo realizującym funkcjonalność wspomnianej wypożyczalni wideo (również dla języka Java).

Przy pisaniu książki wykorzystano doświadczenia autora płynące z pracy w Polsko-Japońskiej Wyższej Szkole Technik Komputerowych oraz uczestnictwa w różnych projektach komercyjnych oraz naukowo-badawczych.

Odbiorcami publikacji mogą być wszyscy zainteresowani współczesnymi obiektowymi językami programowania takimi jak Java, C# czy C++. Książka szczególnie polecana jest dla studentów nauk informatycznych chcących pogłębić swoją wiedzę dotyczącą analizy, modelowania oraz implementacji nowoczesnych systemów komputerowych.

Dla mojej Rodziny

1 Wprowadzenie

Ponad dziesięć lat temu przeczytałem książkę o programowaniu, która mnie urzekła: „Symfonia C++" napisana przez Jerzego Grębosza [Gręb96][1]. Do dzisiaj nie spotkałem lepiej napisanej książki dotyczącej języków programowania. Niektórzy mogą uważać, że pisanie o takich poważnych i skomplikowanych sprawach jak języki programowania wymaga bardzo naukowego stylu. Pan Grębosz zastosował styl „przyjacielski" – jak sam to określił: „bezpośredni, wręcz kolokwialny". Moim celem jest stworzenie książki podobnej w stylu, ale traktującej o całym procesie wytwarzania oprogramowania, ze szczególnym uwzględnieniem pojęć występujących w obiektowości i ich przełożenia na praktyczną implementację. Czy i w jakim stopniu mi się to udało oceną Czytelnicy.

Książka ta powstała na podstawie mojego doświadczenia nabytego w czasie prowadzenia wykładów, ćwiczeń oraz przy okazji prac w różnego rodzaju projektach, począwszy od badawczych, aż do typowo komercyjnych. Na co dzień pracuję w Polsko-Japońskiej Wyższej Szkole Technik Komputerowych[2] jako adiunkt, więc mam też spore doświadczenie wynikające z prowadzenia zajęć ze studentami. Dzięki temu będę w stanie omówić też typowe błędy popełniane przy tworzeniu oprogramowania.

Odbiorcami tej publikacji mogą być wszyscy zainteresowani wytwarzaniem oprogramowania, obiektowością, programowaniem czy modelowaniem pojęciowym, np. programiści, analitycy czy studenci przedmiotów związanych z programowaniem, inżynierią oprogramowania, bazami danych itp. Zakładam, że Czytelnik ma już jakąś wiedzę na temat programowania oraz modelowania, ale wszędzie, gdzie to tylko możliwe, staram się przedstawiać obszerne wyjaśnienia. Aby oszczędzić Czytelnikowi przewracania kartek oraz ułatwić zrozumienie omawianych zagadnień, w niektórych miejscach powielam wyjaśnienia, ale z uwzględnieniem trochę innego punktu widzenia (lub argumentacji).

Pomysł na książkę był prosty: pokazać cały proces wytwarzania oprogramowania, począwszy od analizy potrzeb klienta, poprzez projektowanie, implementację (programowanie), a kończąc na testowaniu. Szczególnie

[1] Na stronie 304 znajduje się bibliografia zawierająca kompletne informacje dotyczące wymienianych w tekście publikacji.

[2] Polsko-Japońska Wyższa Szkoła Technik Komputerowych, 02-008 Warszawa, ul. Koszykowa 86, http://www.pjwstk.edu.pl/.

chciałem się zająć przełożeniem efektów analizy na projektowanie oraz programowanie. W związku z tym, czynności z tej pierwszej fazy są potraktowane nieco skrótowo (nie dotyczy to diagramu klas, który jest omówiony bardzo szczegółowo). Czytelnik, który potrzebuje poszerzyć swoją wiedzę na temat tych zagadnień, powinien sięgnąć po którąś z książek traktujących o modelowaniu, UML itp. (lista proponowanych tytułów znajduje się w ostatnim rozdziale: Bibliografia). Większość przykładów w książce oparta jest na konkretnych wymaganiach biznesowych (wypożyczalnia wideo). Implementacja została wykonana dla języka programowania Java SE 6. Czasami też zamieszczam odnośniki do Microsoft C# czy C++. Na początku książki mamy jakiś biznes do skomputeryzowania (wypożyczalnia wideo), przeprowadzamy jego analizę, robimy projekt, a na koniec częściową implementację w postaci prototypu systemu komputerowego.

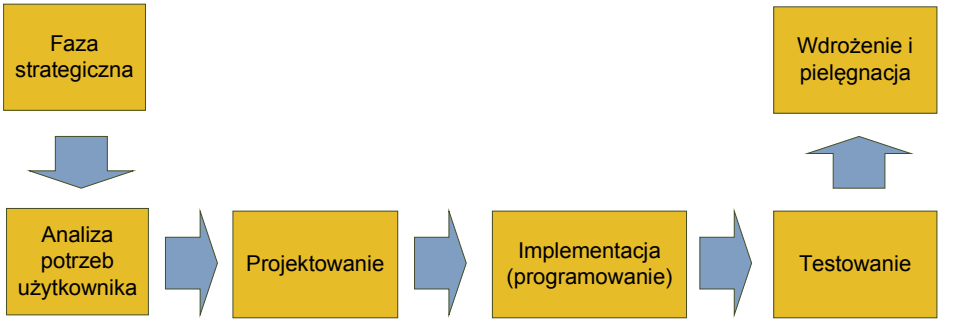
Proces analizy potrzeb użytkownika i projektowania oprogramowania oraz zagadnienia z tym związane (szeroko pojmowana obiektowość) są czasami postrzegane (szczególnie przez niektórych studentów) jako zbędny balast teoretyczny. Można spotkać się z opinią, że należy usiąść i zacząć pisać program (*programować*), a reszta jakoś się ułoży. Nieznajomość tych podstaw, nazwijmy to teoretycznych, lub ich niezrozumienie prowadzi do tego, że programy pisane w obiektowych językach programowania nie są wcale zbyt obiektowe. I tak naprawdę, przez to, że nie korzystają z tych udogodnień, wymagają więcej pracy oraz zawierają więcej błędów. Mam nadzieję, że po przeczytaniu tej książki uwierzysz, drogi Czytelniku, że jak powiedział Kurt Lewin, twórca podstaw współczesnej psychologii społecznej: „nie ma nic praktyczniejszego niż dobra teoria".

To tyle słowem wstępu – dalej już będzie bardziej konkretnie. I jeszcze jedna rzecz: bardzo proszę o przysyłanie uwag, komentarzy, pomysłów, sugestii na adres: mtrzaska@mtrzaska.com.

2 Analiza

Wytwarzanie współczesnego oprogramowania to proces bardzo skomplikowany. Bierze w nim udział cały sztab ludzi, z których każdy ma konkretne zadanie do wykonania. Aby te osoby mogły się wzajemnie porozumieć, muszą mówić wspólnym językiem. W przypadku projektowania systemu informatycznego do tego celu najczęściej używa się notacji UML (*Unified Modeling Language*). Umożliwia ona w miarę precyzyjne opisanie wszystkich elementów składających się na projekt nowoczesnego oprogramowania.

Istnieje wiele różnych metodyk definiujących proces wytwarzania oprogramowania. W większości z nich mamy do czynienia z jakąś wersją faz pokazanych na rysunku 2-1. Nawet jeżeli metodyka jest przyrostowa (iteracyjna), to i tak ma ich jakieś odpowiedniki.



2-1 Typowe fazy wytwarzania oprogramowania

Krótko rzecz biorąc, zadaniem poszczególnych faz jest:

- Faza strategiczna – podjęcie decyzji o ewentualnym rozpoczęciu projektu. Wykonawca szacuje, na podstawie wstępnej analizy, czy jest zainteresowany wykonaniem danego systemu informatycznego. Bierze pod uwagę koszty wytworzenia (w tym pracochłonność), proponowaną zapłatę i ewentualnie inne czynniki (np. prestiż).

- Analiza – ustalamy, co tak naprawdę jest do zrobienia i zapisujemy to przy pomocy różnego rodzaju dokumentów (w tym diagramów

UML). Ta faza raczej abstrahuje od aspektów technologicznych, np. języka programowania.

- Projektowanie – decydujemy, w jaki sposób zostanie zrealizowany nasz system. W oparciu o wybraną technologię (w tym język programowania) wykonujemy możliwie dokładny projekt systemu. W idealnej sytuacji tworzymy diagramy opisujące każdy aspekt systemu, każde działanie użytkownika, reakcję aplikacji itp. W praktyce, w zależności od dostępnego czasu oraz skomplikowania systemu, nie zawsze jest to tak szczegółowe.

- Implementacja poświęcona jest fizycznemu wytworzeniu aplikacji. Innymi słowy, to właśnie tutaj odbywa się programowanie w oparciu o dokładny (mniej lub bardziej) projekt systemu.

- Testowanie – jak sama nazwa wskazuje, testujemy owoce naszej pracy, mając nadzieję, że znajdziemy wszystkie błędy. Ze względu na różne czynniki zwykle to się nie udaje. Ale oczywiście dążymy do tego ideału.

- Wdrożenie i pielęgnacja. Ta faza nie zawsze występuje w pełnej postaci. Wdrożenie polega na zainstalowaniu i zintegrowaniu aplikacji z innymi systemami klienta. Z oczywistych względów nie występuje w przypadku, gdy nasz program sprzedajemy w sklepie (wtedy zwykle wykonuje ją sam kupujący). Zadaniem pielęgnacji jest tworzenie poprawek i ewentualnych zmian. Dlatego też ta faza nigdy się nie kończy. A przynajmniej trwa dopóki klient używa naszego systemu.

Z punktu widzenia tej książki najmniej interesujące są dla nas fazy 1-a (strategiczna) oraz ostatnia (wdrożenie i pielęgnacja). Z tego powodu raczej nie będziemy się nimi zajmować.

2.1 Wymagania klienta

Jak już wspomnieliśmy, książka ta będzie bazowała na wymyślonym przypadku biznesowym. Dzięki temu będziemy w stanie opisać na praktycznych przykładach sytuacje maksymalnie zbliżone do rzeczywistości.

Wśród analityków panuje przekonanie, że klient nie wie, czego chce. Zwykle chce wszystko, najlepiej za darmo i do tego na wczoraj. Po przepro-

wadzeniu fazy analizy, gdy już ustaliliśmy, czego tak naprawdę mu potrzeba, okazuje się, że to twierdzenie bardzo często jest prawdą. W związku z tym warto stosować się do kilku rad:

- Zawsze dokumentuj wszelkie informacje otrzymane od klienta. Nawet jeżeli jesteście świetnymi kumplami (i oby tak pozostało do końca projektu) i rozmowę dotycząca wymagań odbyliście późnym wieczorem przy piwie bezalkoholowym, to następnego dnia należy wysłać mail i poprosić o potwierdzenie wcześniejszych ustaleń. Dzięki temu, gdy klient będzie chciał zmienić zdanie i jedną „drobną" decyzją rozłożyć cały projekt, to mamy dowód, że wcześniej były inne ustalenia.

- Staraj się możliwie dokładnie o wszystko wypytywać. Nie wstydź się zadawać pytań i „męczyć" klienta. To jest właśnie twoja praca. Szybko się przekonasz, że z pozoru błahe pytania i wątpliwości mogą sprawić sporo problemów. A co dopiero kwestie, które już na pierwszy rzut oka są skomplikowane…

- Przy tworzeniu projektu warto rozważyć zastosowanie jakiegoś narzędzia CASE (patrz też podrozdział 4.1.3 na stronie 285). Ułatwi to znacznie wprowadzanie zmian (a te na pewno będą) oraz różne formy publikacji efektów naszej pracy.

Jak łatwo można sobie wyobrazić, proces ustalania wymagań na system nie jest zbyt prosty. Dla potrzeb tej książki załóżmy jednak, że udało nam się go przeprowadzić łatwo i bezboleśnie, a w efekcie otrzymaliśmy „historyjkę" (zamieszczoną w rozdziale 2.2) opisującą biznes naszego klienta. Celowo wybraliśmy wypożyczalnię wideo, ponieważ w zaproponowanym kształcie posiada większość elementów występujących podczas modelowania systemów komputerowych.

2.2 Wymagania dla Wypożyczalni wideo

1. System ma przechowywać informacje o wszystkich klientach. Klient może być firmą lub osobą. Każdy klient „osobowy" jest opisany przez:
    a. Imię,
    b. Nazwisko,
    c. Adres,

   d. Dane kontaktowe.
2. Dla klienta firmowego przechowujemy następujące informacje:
   a. Nazwa,
   b. Adres,
   c. NIP,
   d. Dane kontaktowe.
3. W przypadku klientów prywatnych, klientem wypożyczalni może zostać osoba, która ukończyła 16 lat.
4. System ma przechowywać informacje o wszystkich filmach, kasetach i płytach dostępnych w wypożyczalni.
5. Informacja o filmie dotyczy:
   a. Tytułu filmu,
   b. Daty produkcji,
   c. Czasu trwania,
   d. Aktorów grających główne role,
   e. Opłaty pobieranej za wypożyczenie nośnika z filmem (takiej samej dla wszystkich filmów).
6. Może istnieć wiele nośników z tym samym filmem. Każdy nośnik posiada numer identyfikacyjny.
7. Filmy podzielone są na kategorie, np. filmy fabularne, dokumentalne, przyrodnicze itd. System powinien być dostosowany do przechowywania informacji specyficznych dla poszczególnych kategorii. Zostaną one doprecyzowane w przyszłości.
8. Innym kryterium podziału filmów jest odbiorca docelowy: dziecko, młodzież, osoba dorosła, wszyscy. Dla dzieci musimy pamiętać kategorię wiekową (3, 5, 7, 9, 13 lat), a dla dorosłych przyczynę przynależności do tej kategorii wiekowej (są one z góry zdefiniowane, np. przemoc).
9. Informacja o wypożyczeniu dotyczy daty wypożyczenia oraz opłaty za wypożyczenie.
10. Do jednego wypożyczenia może być przypisane kilka nośników, jednak nie więcej niż trzy. Każdy z pobranych nośników może być oddany w innym terminie.
11. Jednocześnie można mieć wypożyczonych maks. 5 nośników.
12. Nośniki wypożycza się standardowo na jeden dzień, płatne z góry. W przypadku przetrzymania nośnika opłata za każdy dzień przetrzymania zostaje zwiększona o 10% w stosunku do opłaty standardowej.

13. Jeśli fakt przetrzymania powtórzy się trzykrotnie, klient traci na zawsze prawo do korzystania z wypożyczalni.
14. Jeśli klient oddał uszkodzony nośnik, jest zobowiązany do zwrócenia kosztów nowego.
15. Filmy przeznaczone wyłącznie dla osób dorosłych może wypożyczyć osoba, która ukończyła 18 lat.
16. Klient musi mieć możliwość samodzielnego przeglądania informacji o filmach oraz stanu swojego konta.
17. Codziennie opracowuje się raport dzienny o wydarzeniach w wypożyczalni, tzn. O:
    a. Liczbie nowych wypożyczeń,
    b. Liczbie zwrotów,
    c. Liczbie dzisiaj wypożyczonych nośników,
    d. Dziennym utargu.
18. Co jakiś czas opracowuje się raport okresowy (za zadany okres - okresy mogą się nakładać), który zawiera informacje o:
    a. Najczęściej wypożyczanym filmie,
    b. Najpopularniejszej kategorii,
    c. Najpopularniejszym aktorze.
19. Raporty są uporządkowane chronologicznie.

Jak można się zorientować, powyższe wymagania odpowiadają mniej więcej typowej wypożyczalni funkcjonującej na przeciętnym osiedlu. I jak również łatwo się zorientować, nie są całkowicie precyzyjne. Na pewno brakuje tam pewnych informacji, o czym będziesz mógł się przekonać, drogi Czytelniku, w trakcie lektury pozostałych rozdziałów tej książki. Jest to zabieg celowy: po prostu nie chciałem tworzyć osobnej książeczki poświęconej tylko opisowi wymagań na system. W rzeczywistości należy zebrać jak najwięcej informacji. A co gdy jednak o coś zapomnieliśmy zapytać? Czy robimy tak jak uważamy, że będzie dobrze? Oczywiście, w żadnym wypadku nie! Kontaktujemy się z naszym zleceniodawcą i ustalamy szczegóły (np. uwzględniając porady z rozdziału 2.1, strona 10).

2.3 Przypadki użycia

Diagramy przypadków użycia są bardzo ważnym elementem analizy. Ich głównym zadaniem jest zdefiniowanie funkcjonalności tworzonego systemu. Pojęcie funkcjonalności określa, co system ma robić (jakie funkcje może realizować). Ich odzwierciedleniem będą odpowiednie komendy w menu, wyświetlające dedykowane elementy GUI (*Graphical User Inter-*

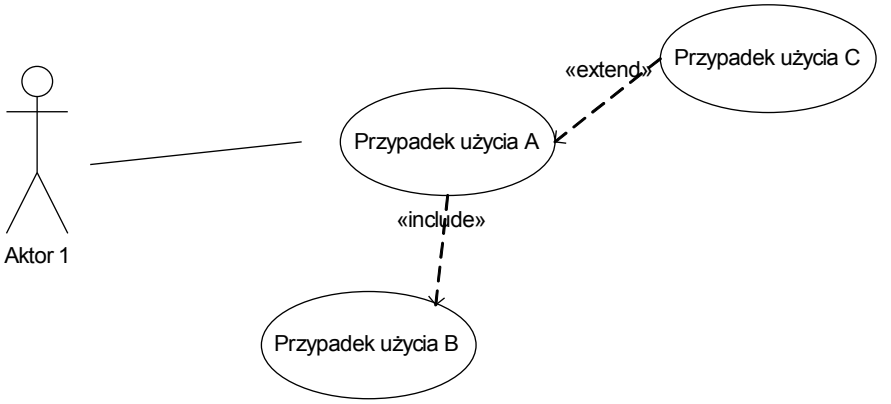
face – graficznego interfejsu użytkownika). Ewentualnie, w przypadku tzw. Aplikacji konsolowych (pozbawionych graficznego interfejsu użytkownika) będą miały przełożenie na odpowiednie parametry tekstowe, np. „/remove" lub „-add". Jednakże na tym etapie tworzenia oprogramowania nie interesuje nas dokładne mapowanie przypadków użycia (funkcji) na konkretne pozycje menu czy przyciski. Nie zajmujemy się też określaniem, jak te funkcje mają być dokładnie wykonywane – to jest przedmiotem fazy projektowania.

Przeprowadzenie dokładnej analizy jest o tyle istotne, że rzutuje na dalsze prace projektowe. Wyobraźmy sobie sytuację, że na tym etapie zapomniano o uwzględnieniu jakiejś funkcji. Oznacza to, że nie zostanie ona uwzględniona przy projektowaniu, implementacji oraz testowaniu. Po prostu jej nie będzie w tworzonej aplikacji! Naturalnie, zawsze można ją dodać na późniejszym etapie (bo nie będziemy mieli innego wyjścia – klient nie zechce systemu, który nie ma wymaganych funkcji). Należy jednak pamiętać, że tworzony system komputerowy jest trochę jak naczynia połączone. Zmiana elementu w jednym miejscu skutkuje zmianami w innym. W efekcie coś, co dobrze działało, może przestać działać lub, co gorsza, zacząć działać źle (złe działanie jest gorsze od braku działania, ponieważ trudniej jest to zaobserwować; gdy coś nie działa, to widzimy to od razu). Generalnie, w inżynierii oprogramowania uważa się, że im błąd wcześniej popełniony, tym ma gorsze skutki dla całego projektu. Jest to też zgodne ze zdrowym rozsądkiem: niezauważenie konieczności dodania funkcji na etapie analizy jest dużo poważniejsze niż jakiś błąd typowo programistyczny, który, gdy występuje w sposób deterministyczny (wiemy, co trzeba zrobić, aby wystąpił), możemy stosunkowo łatwo naprawić.

W większości przypadków, zamiast rysować diagram przypadków użycia (*use case diagram*) możemy wypisać poszczególne funkcje w punktach. Pod względem zawartości te dwa podejścia są prawie równoważne. Wydaje się, że główne zalety diagramu to:

- możliwość zobaczenia wszystkich funkcji „z lotu ptaka",
- łatwość zaobserwowania powiązań pomiędzy przypadkami użycia (czyli funkcjami systemu),
- szansa na wychwycenie funkcji, które są wspólne (powtarzają się) i możemy je zrealizować w bardziej ogólny sposób.

Diagramy przypadków użycia są stosunkowo proste. Ich najważniejsze elementy to (zobacz rysunek 2-2):



2-2 Przykładowy diagram przypadków użycia służący jako ilustracja notacji

- Aktor – uosabia użytkownika danego przypadku użycia. Może nim być konkretna osoba lub instytucja. Nazwa aktora jest raczej nazwą roli (pracownik), a nie konkretnego użytkownika (np. Kowalski). W przypadku analizy jest to ktoś (lub coś), kto fizycznie obsługuje system. Przykładowo, dla przypadku użycia opisującego sprzedaż w supermarkecie będzie to kasjerka, a nie klient robiący zakupy. Natomiast w tym samym supermarkecie klient może być aktorem dla przypadku użycia sprawdzania ceny w skanerze (bo to rzeczywiście klient obsługuje system).

- Przypadek użycia - jest po prostu funkcją systemu (np. dodanie towaru). Nazwy powinny być tworzone z punktu widzenia systemu (np. sprzedaż towaru, a nie zakup towaru – bo to jest punkt widzenia klienta).

Przypadki użycia mogą być powiązane na dwa sposoby:

- <<include>> oznacza, że przypadek użycia a zawsze korzysta (wywołuje, uruchamia) przypadek użycia B,

- <<extend>> formalnie oznacza, że przypadek użycia A jest rozszerzany przez przypadek użycia C. Mówiąc po polsku: A czasami używa C. Nie mamy informacji jak często. Jeżeli dodatkowo wyspecyfi-

kowaliśmy punkt rozszerzalności (*extension point*), to wiemy, kiedy to ewentualne użycie zachodzi. Warto zwrócić uwagę na zwrot strzałek: mówiąc, że A czasami używa C, pokazujemy z C do A (wzięło się to z tego, że C „rozszerza" A).

2.3.1 Ogólny diagram przypadków użycia dla Wypożyczalni wideo

Diagramy przypadków użycia można rysować na różnym poziomie szczegółowości. Zwykle robi się jeden (lub więcej dla bardzo rozbudowanych systemów) ogólny diagram i uszczegóławia się go na oddzielnych diagramach. Nasuwa się pytanie: jak bardzo należy wchodzić w szczegóły? Nie ma jednej precyzyjnej odpowiedzi. Przeważnie robimy to tak, aby dało się z niego odczytać dość ogólne informacje. Do przedstawiania bardziej szczegółowych i precyzyjnych danych służą inne diagramy (np. aktywności – podrozdział 2.5 na stronie 94, stanów – podrozdział 2.6 na stronie 95; warto także zajrzeć do książki [Płod05]).

Rysunek 2-3 zawiera przykładowy, ogólny (zawierający wszystkie funkcje) diagram dla naszej wypożyczalni. Słowo przykładowy nie jest przypadkowe. Inny analityk (np. ty, Czytelniku) mógłby narysować (trochę) inny diagram. Jego kształt oraz zakres pokazanych funkcji zależy od dwóch głównych czynników:

- Uwarunkowania biznesowe, czyli co ten system ma robić (jakie ma mieć funkcje). Część z nich staje się jasna dopiero po zadaniu wielu pytań naszemu klientowi. Niektórzy analitycy je zadadzą, inni nie. Może to sugerować po prostu błędny diagram. Owszem może, ale nie musi – patrz następny punkt (ktoś mógł uznać, że pewne informacje są zbyt szczegółowe).


Wyszukiwanie informacji o aktorach
<<extend>>
<<extend>>
Wyszukanie informacji o filmach
Klient
<<extend>>
Wyszukanie informacji o nośnikach
Właściciel
Wypożyczenie nośnika
Generowanie raportów
Pracownik
Przyjęcie nośnika
Zarządzanie informacjami o klientach
Zarządzanie informacjami o nośnikach
Zarządzanie informacjami o aktorach
Zarządzanie informacjami o filmach

2-3 Diagram przypadków użycia dla wypożyczalni wideo

- Sposobu modelowania (postrzegania pewnych zjawisk) konkretnego analityka. Nawet mając te same dane wejściowe (sytuację biznesową), różni analitycy mogą stworzyć różne diagramy. Naturalnie część z nich może być błędna (mniej lub bardziej), ale też może istnieć wiele poprawnych (też mniej lub bardziej) i zarazem różnych rozwiązań. Te różnice są szczególnie widoczne w przypadku bardziej skomplikowanych diagramów (np. klas – patrz rozdział 2.4). Można w takim razie zapytać; jakie są kryteria poprawności – który z nich jest lepszy, a który gorszy? i znowu nie ma jednoznacznej odpowiedzi – po prostu różni ludzie widzą te same sprawy w różny sposób (i jak pewnie nieraz mogłeś się przekonać, drogi Czytelniku, nie dotyczy to tylko modelowania). Na pewno błędnym diagramem będzie taki, który nie przykrywa pewnej funkcjonalności, tworzy nową, o której zleceniodawca nie wspominał (tak – i w tym przypadku nadgorliwość jest wadą) albo po prostu utrudnia jej zrealizowanie.

Poniżej w punktach omówimy niektóre decyzje projektowe podjęte przy okazji tworzenia tego diagramu:

- Aktorzy. Jak łatwo się zorientować, ich liczba oraz zakres „obsługiwanych" funkcji nie wynika wprost z wymagań na system (patrz podrozdział 2.2, strona 11). Przykładowo, funkcje systemu związane z dodawaniem informacji o nowych tytułach oraz nośnikach mógłby pełnić dedykowany aktor. W prawdziwym systemie tego typu decyzje są uzależnione od czynników biznesowych (np. wielkości firmy). Na potrzeby tej książki chcieliśmy ograniczyć ich liczbę. Załóżmy, że w toku długich rozmów z klientem doszliśmy do wniosku, że właśnie tacy aktorzy będą obsługiwać nasz system.

- Przypadki „Wyszukanie informacji o aktorach" oraz „Wyszukanie informacji o filmach" są wzajemnie powiązane przy pomocy <<extend>>. Sens tego jest taki, że szukając informacji o filmach (a właściwie konkretnym filmie), możemy chcieć zobaczyć aktorów, którzy tam grają. Możliwa jest też sytuacja odwrotna: mając konkretnego aktora, możemy chcieć zobaczyć „jego" filmy. Ponieważ możemy to zrobić, ale nie zawsze będziemy korzystali z tej funkcji, zasto-

sowaliśmy <<extend>>, który jak wcześniej powiedzieliśmy, oznacza opcjonalne wykorzystanie.

- Przypadek użycia „Wyszukiwanie informacji o nośnikach" rozszerza „Wyszukiwanie informacji o filmach". Oznacza to, że czasami, wyświetlając informacje o filmie, chcemy zobaczyć listę nośników, na których się znajduje.

- „Wypożyczenie nośnika" oraz „Przyjęcie nośnika" są przypisane do aktora Pracownik. Jest tak dlatego, że to właśnie pracownik obsługuje system w momencie wypożyczania/zwracania. Gdybyśmy chcieli zrobić system dla wypożyczalni samoobsługowej, to właściwym aktorem byłby klient.

- Grupa przypadków użycia „Zarządzanie informacjami o …". Bierzemy tu pod uwagę dodawanie, usuwanie i edycję. Oczywiście można dyskutować, czy nie powinniśmy do tego włączyć też wyszukiwania: raczej nie, ponieważ wyszukiwanie jest dostępne dla klienta, a dodawanie/usuwanie pewnie nie powinno.

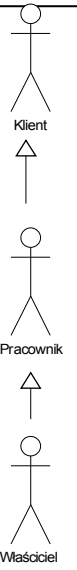
- No i ostatni przypadek użycia, czyli „Generowanie raportów". Jest on przypisany dla Właściciela, a nie pracownika. Zakładamy, że w trakcie wywiadu nasz zleceniodawca postanowił ograniczyć wgląd do raportów dla zwykłych pracowników. Drugą kwestią dyskusyjną jest to, czy należy wypisywać (pokazywać na diagramie) te wszystkie rodzaje raportów. Ja uznałem, że nie (bo na tym etapie szczegółowości jest ważny dla nas sam fakt tworzenia raportów; i tak nie będziemy w stanie pokazać tutaj konkretnych informacji, które mają się tam znaleźć), ale inny analityk (np. Ty, drogi Czytelniku) mógłby chcieć stworzyć dwa różne przypadki użycia (po jednym dla każdego rodzaju raportu) lub rozszerzyć (<<extend>>) ten, który ja dodałem.

- Warto zwrócić też uwagę na nazewnictwo przypadków użycia. Nazwy powinny odpowiadać nazwom funkcjonalności, np. *dodanie klienta* lub, ewentualnie, *dodaj klienta*. Raczej unikamy nazw typu *dodawanie klienta* (to jest dobra nazwa dla stanu na diagramie stanów). A na pewno całkowicie złą nazwą jest: *klient* (bo co niby miałoby to oznaczać? Niestety takie nazwy też widywałem).

- Uważny czytelnik (mam nadzieję, że to jesteś właśnie Ty), mógłby się zdziwić: chwileczkę, ale jak to – szef nie ma dostępu do funkcji zwykłego pracownika czy wręcz klienta (analogicznie z funkcjami pracownika oraz klienta)? Faktycznie trochę dziwne. Zastanówmy się, jakie mamy możliwości, aby to zmienić:

    - Pierwsza to połączenie wszystkich przypadków użycia klienta z aktorem pracownik. Analogicznie łączymy wszystkie przypadki pracownika z szefem. W rezultacie mamy straszną gmatwaninę linii na diagramie (a diagram z definicji powinien być czytelny).

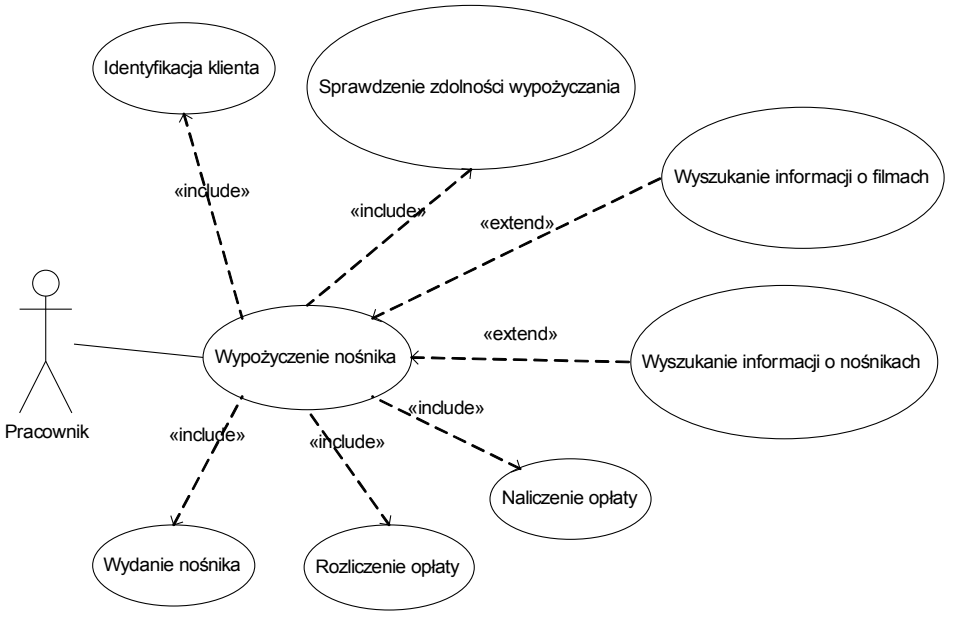
    - Narysować oddzielny diagram pokazujący zależności pomiędzy aktorami (lub nanieść je na istniejący diagram – tak też można). I taki właśnie diagram jest na rysunku 2-4. Specjalny znaczek (strzałka z trójkątnym grotem) oznacza dziedziczenie. Czyli wszystkie przypadki użycia nad-aktora są dostępne dla pod-aktora, odpowiednio: klient i pracownik oraz pracownik i właściciel. Rozwiązanie z dziedziczeniem jest czytelniejsze oraz umożliwia łatwe wprowadzanie zmian (modyfikujemy tylko hierarchię dziedziczenia aktorów, co oznacza zmiany dostępnych przypadków użycia dla wszystkich zaangażowanych aktorów).



2-4 Diagram dziedziczenia dla aktorów

2.3.2 Szczegółowy diagram przypadków użycia

Jak już wcześniej ustaliliśmy, nie ma jednoznacznych reguł, czy i jak organizować różne stopnie szczegółowości dla diagramów przypadków użycia. Każdy z analityków może mieć własny pogląd na tę sprawę. Generalnie zakłada się, że umieszczanie zbyt wielu szczegółów zmniejsza czytelność tych diagramów i dlatego należy tego unikać.



2-5 Przykładowy, uszczegółowiony diagram przypadków użycia

Rysunek 2-5 zawiera uszczegółowioną wersję jednego z przypadków użycia („Wypożyczenie nośnika") przedstawionych ogólnie na rysunku 2-3 (strona 18). Poniżej zamieszczono jego skrótową analizę:

- „Identyfikacja klienta" – zanim będziemy mogli wypożyczyć nośnik, musimy wiedzieć, którego klienta ta operacja dotyczy. Warto zwrócić uwagę, że nazwa tego przypadku użycia jest dość ogólna i nie precyzuje sposobu identyfikacji. Naturalnie jest to celowy zabieg. Z punktu widzenia wypożyczania nie interesuje nas, jak to się będzie odbywało – czy wykorzystamy jakiś rodzaj podawania danych (np. numer klienta), czy coś bardziej zaawansowanego jak czytnik kart magnetycznych (zakładając, że nasza wypożyczalnia jest aż tak nowoczesna). Można zwiększyć stopień szczegółowości tego diagramu poprzez dodanie przypadków użycia rozszerzających (zastosujemy <<extend>>, ponieważ prawdopodobnie będziemy korzystali tylko z jednego naraz) „Identyfikacja klienta" o np. „Identyfikacja kartą" oraz „Identyfikacja numerem".

- Następnie musimy być pewni, że dany klient (uprzednio zidentyfikowany) może coś wypożyczać. W wymaganiach na system, które z takim trudem ustaliliśmy, są podane pewne warunki (patrz m.in. punkt 2.2.2.2 i dalsze, strona 12), które muszą być spełnione.

- Opisując powyższe przypadki użycia użyliśmy, sformułowań takich jak „następnie", „uprzednio". Jest to naturalne, ponieważ chcemy zachować pewien porządek, który zwiększy nasze szanse na niepominięcie niczego istotnego. W związku z tym należy się zastanowić, w jakiej kolejności umieszczamy przypadki użycia na diagramie? Od prawej do lewej, z góry na dół? Pytanie jest podchwytliwe – otóż diagramy przypadków użycia nie uwzględniają kolejności. Dlatego kolejność dołączania poszczególnych elementów („pod-przypadków" użycia) jest dowolna. Zwykle robimy to zgodnie z kierunkiem ruchu wskazówek zegara, ale jest to tylko zdroworozsądkowy zwyczaj.

- Na pewno zwróciłeś uwagę na to, że część elementów jest dołączona korzystając z <<include>>, a inne przy użyciu <<extend>>. Jak mówiliśmy wcześniej, użycie <<include>> oznacza, że coś jest zawsze wykonywane, a <<extend>>, że tylko czasami. Tutaj jest tak samo. Zastanowienie może budzić tylko kwestia, dlaczego, w świetle powyższego rozumowania, „Wyszukanie informacji o filmach" oraz „Wyszukanie informacji o nośnikach" są wykonywane czasami. Przecież witalną częścią wypożyczania jest wyszukiwanie powyższych informacji. Oczywiście, że tak, ale może wystąpić kilka scenariuszy:

    - Klient przychodzi i mówi, że chce film „Terminator". W takiej sytuacji musimy odnaleźć nośnik, który go zawiera.

    - Klient przychodzi i mówi, że chce po prostu fajny film sensacyjny. System przedstawia kilka propozycji (ponieważ każdy film należy do jakieś kategorii – patrz punkt 2.2.2.2, strona 12). Ale klientowi żaden z nich nie odpowiadał, więc nic nie wypożyczył – czyli nie musieliśmy szukać nośnika.

    - Wreszcie, może być tak, że klient chce wypożyczyć nośnik o numerze 6341. Wtedy nie musimy wyszukiwać filmu.

Jak widać, możliwości jest dość dużo. I dlatego wybraliśmy najbardziej uniwersalne rozwiązanie: obydwa przypadku użycia są wykorzystywane opcjonalnie. Rysując diagram przypadków użycia, należy założyć pewien scenariusz, który określa, co się stanie, jakie decyzje podejmie klient itp. W przeciwnym wypadku większość naszych przypadków użycia wyglądałaby podobnie: najpierw identyfikacja, którą przeprowadzamy zawsze (<<inculde>>), a później same połączenia z <<extend>>'ami (ponieważ zawsze może nie dojść do wykonania przypadku użycia).

- „Naliczenie opłaty" polega po prostu na wyliczeniu należności za wypożyczane filmy. Ponieważ wykorzystaliśmy <<include>, więc łatwo się domyślić, że zakładamy, iż klient wybrał coś do wypożyczenia.

- „Rozliczenie opłaty" jest nazwą dość ogólną. Analogicznie jak przy „Identyfikacji klienta", specjalnie nie chcemy wnikać w szczegóły, np. rozliczenie może być za pomocą karty, gotówki czy po prostu dopisania do rachunku opłacanego przelewem raz w miesiącu.

- „Wydanie nośnika" oznacza, że cały proces wypożyczania został szczęśliwie zakończony i powinien zostać zapamiętany w systemie. Celowo nie piszemy czegoś w rodzaju „zatwierdzenie transakcji w bazie danych", ponieważ przypadki użycia z definicji unikają wszelkiego „technicyzowania".

2.4 Diagram klas

Diagram klas jest uznawany chyba za najważniejszy z diagramów wykorzystywanych w czasie tworzenia oprogramowania. I nie jest tak bez przyczyny: informacje, które się na nim znajdują, są bezpośrednio wykorzystywane przy implementacji (naturalnie, jeżeli korzystamy z obiektowego języka programowania takiego jak Java, C# czy C++). Definiują sposób przechowywania informacji w naszym systemie, a poprzez umieszczenie metod pełnią rolę swoistego szkieletu umożliwiającego zdefiniowanie zachowania się tworzonej aplikacji. Tak naprawdę możemy mówić przynajmniej o dwóch rodzajach diagramów klas:

- Tworzonym na etapie analizy (i tym zajmiemy się w tym rozdziale). W tym przypadku możemy korzystać z dowolnych konstrukcji występujących w obiektowości oraz wykorzystywanej notacji (w naszym przypadku jest to notacja UML).

- Tworzonym na etapie projektowania (ten będzie przedmiotem naszego zainteresowania w rozdziale 3, strona 97), a wykorzystywanym na etapie programowania. Projektując system, musimy wziąć pod uwagę możliwości środowiska, w którym będzie odbywała się implementacja. Z tego powodu wszystkie konstrukcje umieszczone na etapie analizy, a niewystępujące w danym języku (np. w Java), należy odpowiednio zmodyfikować (zastąpić je właściwymi odpowiednikami).

Czytając opis powyższego podziału, można się zastanawiać, po co na etapie analizy stosować pewne konstrukcje, skoro później trzeba będzie je zastąpić czymś innym? Tradycyjnie już nie będę w stanie podać odpowiedzi, która wszystkich zadowoli, ale spróbujmy przedstawić pewne argumenty przemawiające za takim sposobem modelowania:

- Teoretycznie na etapie analizy nie znamy naszego środowiska implementacyjnego, a co za tym idzie, nie wiemy, które konstrukcje są dozwolone, a które nie. Słowo „teoretycznie" jest tutaj w pełni uzasadnione, ponieważ w praktyce w większości przypadków nasz docelowy język programowania będziemy wybierali spomiędzy kilku najbardziej popularnych: Java, C#, C++. Niezależnie od różnic pomiędzy nimi wszystkie mają podobne (takie same?) ograniczenia dotyczące (nie)występowania pewnych konstrukcji z dziedziny obiektowości. Czyli, dla większości sytuacji, ten argument nie wydaje się bardzo zasadny.

- Mocniejszym argumentem jest to, że te „szczególne" (niewystępujące w popularnych językach programowania) konstrukcje są bardzo użyteczne z punktu widzenia analizy, oraz co ważniejsze, istnieją proste techniki ich obejścia. W związku z tym, z kwestią zastąpienia jednych konstrukcji innymi nie powinno być większego problemu.

Spotyka się też podejście, polegające na niewykorzystywaniu tych „trudnych" konstrukcji. Pewną korzyścią będzie łatwiejsza implementacja, ale bardzo istotną wadą takiego postępowania może być nadmierne skomplikowanie fazy analizy (m.in. poprzez trudniejsze „odczytywanie" diagramu). Czyli, innymi słowy, to co przeanalizowaliśmy, łatwiej zaprojektujemy, ale

zwiększamy prawdopodobieństwo popełnienia błędów we wcześniejszej fazie (analizy). A jak pamiętamy z poprzednich rozdziałów, im błąd wcześniej popełniony, tym większe problemy może sprawić. Dlatego, ja osobiście przestrzegam przed unikaniem tych konstrukcji na etapie analizy.

Myślę, że teraz jest właściwy moment, abyśmy przeanalizowali poszczególne elementy występujące na diagramie klas. Jedna drobna uwaga: jak pisaliśmy wcześniej, książka ta jest poświęcona głównie przejściu od fazy analizy, poprzez projektowanie do implementacji. Nie będziemy się koncentrowali na bardzo dokładnym i szczegółowym omówieniu pojęć związanych z obiektowością jako taką. Więcej informacji na ten temat można znaleźć w [Płod05], [Fowl04] czy [Wryc05].

2.4.1 Obiekt

Każda z osób, które zetknęły się z programowaniem w jednym ze współczesnych języków programowania (np. Java, C# czy C++), ma jakieś własne, intuicyjne wyobrażenie obiektu. Spróbujmy je doprecyzować, na razie zapominając o językach programowania (bez obawy - wrócimy do nich w rozdziale 3 „Projektowanie”, strona 97). Nasza definicja będzie oparta na tej pochodzącej z [Płod05] (z niewielkimi zmianami):

Obiekt — byt, który posiada dobrze określone granice i własności oraz jest wyróżnialny w analizowanym fragmencie dziedziny problemowej.

Z pojęciem obiektu związane jest zagadnienie jego tożsamości. Zakłada się, że obiekt jest rozpoznawany na podstawie samego faktu istnienia, a nie korzystając z jakiejś kombinacji jego cech. Warto zauważyć, że jest to definicja niebiorąca pod uwagę specyfiki systemów komputerowych. Wrócimy do tej kwestii w rozdziale 3 (strona 98). Przykładami obiektów mogą być: krzesło, budynek, faktura, student, osoba itd.

2.4.2 Klasa

Jak sama nazwa wskazuje, podstawowym elementem znajdującym się na diagramie klas są klasy. Cóż to takiego jest klasa? Naukowcy zajmujący się problemami obiektowości nie są do końca zgodni, ale myślę, że jedną z najlepszych definicji może być definicja pochodząca z [Płod05]:

Klasa	nazwany opis grupy obiektów o podobnych własnościach.

Zgodnie z tą definicją, klasa nie jest zbiorem obiektów (do tego jest oddzielne pojęcie). Jest to bardzo ważne stwierdzenie, ponieważ często jest to mylone. Innymi słowy, cytując z pamięci [Gręb96]: klasa jest jakby przepisem na obiekt. Opisuje jego cechy (atrybuty), zachowanie (metody) oraz budowę (w kontekście języków programowania).

Nazwa klasy powinna być rzeczownikiem w liczbie pojedynczej (np. pracownik). Można użyć liczby mnogiej, jeżeli pojedyncza instancja klasy (obiekt) opisuje wiele elementów (np. nazwa Pracownicy w przypadku gdy przechowujemy informacje o ich grupach). Dozwolone jest też używanie kilku słów, jeżeli ma to biznesowe uzasadnienie.

Ważnym pojęciem związanym z klasą jest jej ekstensja. Spójrzmy na definicję:

Ekstensja klasy	Zbiór aktualnie istniejących wszystkich wystąpień (innymi słowy: instancji lub jak kto woli: obiektów) danej klasy.

Dla pewności potwierdźmy: tak - w działającym systemie komputerowym będzie wiele ekstensji klas, w przybliżeniu (patrz dalej uwagę o dziedziczeniu) po jednej dla każdej istniejącej klasy biznesowej, np. Film, Aktor, Kaseta itp.

Tak naprawdę powyższa definicja nie jest do końca precyzyjna (ktoś wie dlaczego?), ponieważ nie określa zależności w stosunku do obiektów z podklas oraz nadklas (wrócimy do tego przy okazji dziedziczenia – patrz podrozdział 2.4.4.2 na stronie 49).

Klasy możemy podzielić na:

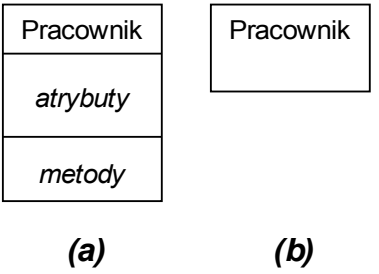
- Konkretne,
- Abstrakcyjne.

Aby dobrze zrozumieć ten podział, konieczne jest wprowadzenie pojęcia dziedziczenia. Dlatego wrócimy do tego tematu w podrozdziale 2.4.4 na stronie 47.

Rysunek 2-6 zawiera różne warianty notacji służącej do przedstawiania klasy na diagramie:

- Część (a) pokazuje kompletną klasę (razem z atrybutami oraz metodami),

- Część (b) przedstawia klasę bez pokazanych metod oraz atrybutów. Notacja zezwala na ukrywanie niepotrzebnych (brakujących) elementów.



2-6 Notacja dla klas

2.4.2.1 Atrybuty

Atrybuty służą do opisu własności obiektów należących do pewnej klasy. Ze względu na różne kryteria można je podzielić na kilka rodzajów.

Tabela 2-1 zawiera rodzaje atrybutów, które mogą znajdować się w klasie. Poniżej jest kilka uwag do jej zawartości:

- Atrybut powtarzalny powinien mieć nazwę w liczbie pojedynczej, np. Imię. Jeżeli nazwiemy go imiona, to będzie oznaczało, że mamy wiele imion (wnioskujemy to z jego nazwy, a zatem znaczenia). Jeżeli dodatkowo oznaczymy go jako powtarzalny, to całość będzie oznaczała jakąś listę imion, z której każdy element będzie miał wiele wartości (w tym przypadku imion). Czyli będzie to lista list, a raczej nie o to nam chodziło.

Tabela 2-1. Rodzaje atrybutów

Kryterium	Rodzaj	Opis	Przykład
Budowa	Prosty	Przechowuje atomowe wartości: liczba lub napis.	Pensja, nazwisko, nazwa koloru, waga

	Złożony	Składa się z atrybutów prostych lub innych atrybutów złożonych.	Adres (może się składać z ulicy, numeru domu, miasta, kodu pocztowego), współrzędne położenia w przestrzeni 3D (składają się z wartości dla osi x, y, z)
Liczność	Pojedynczy	Posiada jedną wartość.	Data urodzenia, płeć
	Powtarzalny	Posiada jedną lub więcej wartości.	Imię (bo ktoś może mieć wiele imion)
Przynależność	Obiektu	Każdy obiekt w klasie może mieć własną wartość.	Dla klasy *Pracownik* przykładami mogą być: nazwisko, imię, data urodzenia.
	Klasowy	Ma tę samą wartość dla wszystkich obiektów danej klasy.	Dla klasy *Pracownik* przykładami mogą być: minimalny wiek, maksymalny wiek.
Ustalanie wartości	Konkretny	Wartość atrybutu jest przechowywana bezpośrednio.	Data urodzenia.
	Wyliczalny	Wartość atrybutu może być wyliczona na podstawie innych danych (np. atrybutów). Czasami przechowujemy tę wyli-	Wiek (obliczony na podstawie daty urodzenia oraz aktualnej daty systemowej).

		czoną kopię.	
Obowiązkowość	Wymagany	Jeżeli nie oznaczono inaczej to atrybut musi mieć wartość.	Nazwisko w klasie pracownik.
	Opcjonalny	Atrybut może nie mieć wartości.	Nazwisko panieńskie w klasie pracownik (mężczyźni zwykle nie mają nazwiska panieńskiego).

- Jak widać z przykładu dotyczącego atrybutu obiektu, nie ma przeszkód, aby wiele (lub wszystkie) obiekty danej klasy miały taką samą wartość danego atrybutu (w przykładzie jest data urodzenia, która dla odpowiednio dużej liczby osób na pewno się powtórzy).

- Zaletą korzystania z atrybutu klasowego (ta sama wartość dla wszystkich obiektów danej klasy) jest to, że przechowujemy go tylko raz („w klasie", a nie „w obiekcie"), co oszczędza pamięć (tak, pamiętam, że mieliśmy abstrahować od języków programowania, ale to ważny argument) i znacząco ułatwia zmiany jego wartości (ponieważ atrybut klasowy nie musi mieć stałej wartości – to też jest czasami mylone). Oprócz tego korzystamy z niego w kontekście klasy (a nie obiektu), co oznacza, że możemy go używać nawet jeżeli nie istnieje żaden obiekt danej klasy.

- Atrybut wyliczalny budzi, przynajmniej na początku, sporo wątpliwości. Po co przechowywać dane dwa razy? Dla jasności: jego istnienie niekoniecznie musi oznaczać dosłowne przechowywanie. Czasami możemy wyliczać jego wartość w razie potrzeby (pewnie zrobilibyśmy tak z wiekiem). W innych sytuacjach, rozsądniej będzie zapamiętać w systemie tę obliczoną wartość (czyli wystąpi jakaś forma

redundancji danych[3]). Kryterium decyzyjnym jest oczywiście koszt obliczenia oraz częstość dostępu:

- ustalenie wieku nie kosztuje zbyt dużo czasu procesora,
- gdybyśmy jednak mieli atrybut, np. średnia wartość towarów sprzedanych przez handlowca, to w przypadku dużej firmy jego obliczenie (na podstawie faktur) może zająć trochę czasu. W takiej sytuacji rozsądne wydaje się przechowywanie jego kopii. W efekcie pewne informacje są zapisane w systemie wielokrotnie i w przypadku aktualizacji należy zadbać o zmianę wszystkich „egzemplarzy". Dlatego stosujemy specjalną notację, która nas ostrzega: „uwaga – redundancja danych"!

- Może to zabrzmi dość trywialnie (ale widziałem sporo takich błędów), ale gdy jakiś atrybut oznaczymy jako wyliczalny, to musimy się upewnić, że w systemie są dane potrzebne do ustalenia jego wartości, np. umieszczenie wyliczalnego atrybutu wiek, bez przechowywania daty urodzenia, nie jest najlepszym pomysłem.

- Pewne rodzaje atrybutów można łączyć (po jednym z każdego kryterium), np. powtarzalny, klasowy i do tego wyliczalny. Pamiętajmy jednak, że nie wszystkie kombinacje mają biznesowy sens.

- Dodatkowo, każdy z atrybutów może być oznaczony specjalnym ograniczeniem *Unique*[4]. Taki atrybut posiada unikalną wartość w ramach ekstensji, czyli może istnieć tylko jeden obiekt należący do danej klasy, który ma atrybut z daną wartością. Wykorzystujemy to wtedy, gdy chcemy, aby to system dbał o unikalność danych, np. mamy klasę Osoba i atrybut PESEL.

[3] Redundancja danych jest terminem z zakresu baz danych i w skrócie oznacza przechowywanie wielu kopii tych samych informacji. Powodami takiego podejścia mogą być m.in. bezpieczeństwo (w razie awarii mamy „zapasową" kopię) lub wydajność (każdorazowe obliczanie danej wartości jest czasochłonne).

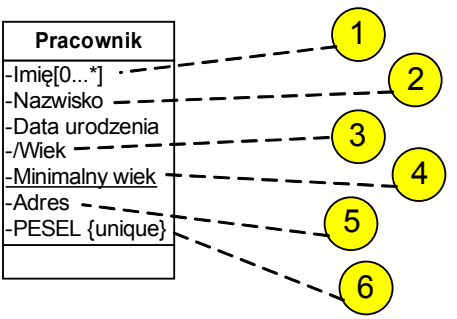
[4] Ograniczenia są jednym z mechanizmów rozszerzalności UML i ogólnie rzecz biorąc, umożliwiają umieszczenie na diagramie informacji, których nie jesteśmy w stanie wyrazić przy pomocy np. klas czy atrybutów. Mogą być zapisywane zwykłym tekstem, korzystając ze specjalnego języka OCL czy wyrażeń matematycznych. Umieszczamy je w nawiasach klamrowych.

- Atrybut opcjonalny może nie mieć wartości. Jest to użyteczne co najmniej w dwóch przypadkach:

    - Wartość nie ma sensu z biznesowego punktu widzenia, np. w firmie mamy pracowników będących mężczyznami oraz kobietami, ale z jakichś powodów nie chcemy dla nich tworzyć oddzielnych klas. Dlatego wystąpienia (obiekty) klasy *Pracownik* opisujące mężczyzn nie będą przechowywały wartości dla atrybutu *Nazwisko panieńskie*. Oczywiście można to obejść i zapamiętywać tam np. pusty ciąg tekstowy, ale z punktu widzenia systemu będzie tam wartość, tylko że specyficzna.

    - Nasze dane są niepełne i w związku z tym dla niektórych obiektów i ich niektórych atrybutów nie mamy wszystkich informacji, np. część pracowników nie podała daty swojego urodzenia.

Rysunek 2-7 zawiera notację UML służącą do oznaczania różnych rodzajów atrybutów. Pominięto sposób oznaczania typu atrybutu – będzie o tym przy okazji projektowania. Kilka uwag:



2-7 Notacja do oznaczania atrybutów

1. Atrybut powtarzalny oznaczamy pokazując jego liczności. Zapis [0…*] oznacza dowolną liczność, a np. [1…4] oznacza, że atrybut może mieć od 1 do 4 wartości. Tak naprawdę, brak informacji o liczności oznacza po prostu liczność [1].

2. Brak specjalnych symboli zakłada, że mamy atrybut z licznością [1], czyli wymagany, obiektu i konkretny.
3. Oznaczenie atrybutu wyliczalnego.
4. Atrybut klasowy. Tutaj mamy na myśli minimalny wiek, powyżej którego osoba może zostać naszym pracownikiem.
5. Adres jest przykładem atrybutu złożonego. Jak widać, ten rodzaj nie ma żadnych specjalnych oznaczeń.
6. W ten sposób mówimy systemowi, że ten atrybut ma mieć unikalną wartość wśród obiektów danej klasy.

2.4.2.2 Metody

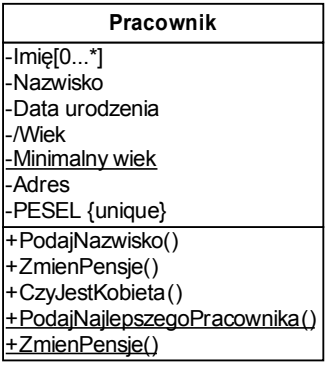
Metody służą do opisu zachowania obiektu. Każda z nich może zwracać jakąś wartość (ale nie musi) i pobierać parametry (również nie musi). W przeciwieństwie do atrybutów, gdzie zróżnicowanie było spore, tutaj mamy dość prosty podział:

- Metody obiektu. Operują na konkretnym obiekcie. Mają dostęp do jego atrybutów oraz innych metod tego obiektu (a dokładniej: innych metod zdefiniowanych w klasie, do której należy ten obiekt). Nie mamy możliwości bezpośredniego odwołania się do innego obiektu (w tym jego zawartości) – nawet z tej samej klasy. Można powiedzieć w uproszczeniu, że pojedynczy obiekt nie jest świadomy istnienia innych obiektów danej klasy (chyba że dostanie się do nich poprzez ekstensję). Przykłady to (z dokładnością do nazewnictwa):
    - `PodajNazwisko():String`, zwracamy nazwisko konkretnego pracownika (tego, na rzecz którego wywołaliśmy tę metodę).
    - `ZmienPensje(NowaPensja:Real):void`, zmieniamy pensję konkretnego pracownika.
    - `CzyJestKobieta():boolean`, pytamy czy konkretny pracownik jest kobietą.
- Metody klasowe. Mają dostęp do całej ekstensji, a zatem do wszystkich obiektów należących do danej klasy. Mogą na nich operować, ale nie muszą: może być np. metoda klasowa, która zwróci jakąś wartość bez „zaglądania” do ekstensji (dlatego stwierdzenie, że metody klasowe operują na całej ekstensji, nie jest do końca precyzyjne).

Analogicznie jak w przypadku atrybutów, metody klasowe wywołujemy na rzecz klasy, a nie konkretnego obiektu. Dzięki temu można z nich korzystać nawet, gdy nie ma jeszcze obiektów danej klasy. Przykłady:

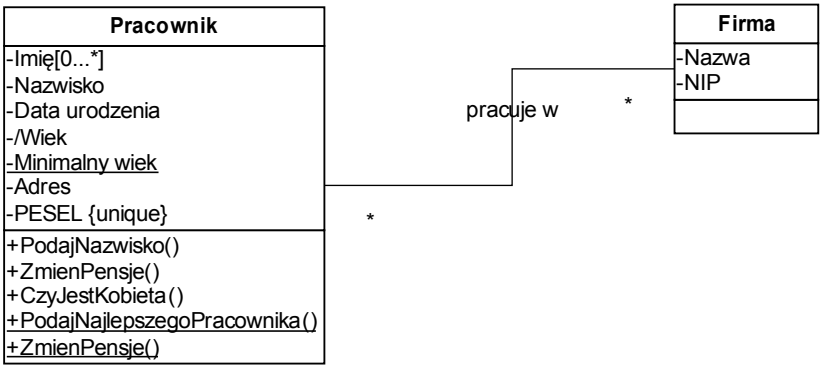
- `PodajNajlepszegoPracownika():Pracownik`,
- `ZmienPensje(NowaPensja:Real):void`, analogiczny przykład był podany dla metod obiektu, ale tutaj ma inne znaczenie; tam zwiększaliśmy pensję jednemu pracownikowi, tutaj wszystkim (całej ekstensji).

Rysunek 2-8 pokazuje notację służącą do umieszczania metod na diagramie (pominięto informacje o parametrach oraz zwracanym typie, pokazano również atrybuty). Jak widać, oznaczenie metody klasowej jest analogiczne do oznaczenia atrybutu klasowego: podkreślona nazwa.



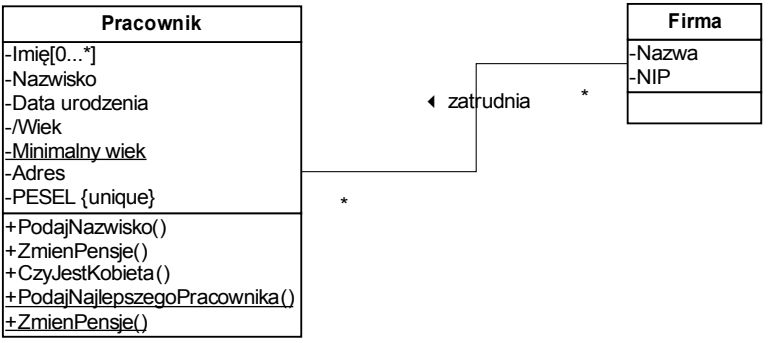
2-8 Przykładowa klasa ilustrująca wykorzystanie notacji UML

2.4.3 Asocjacje



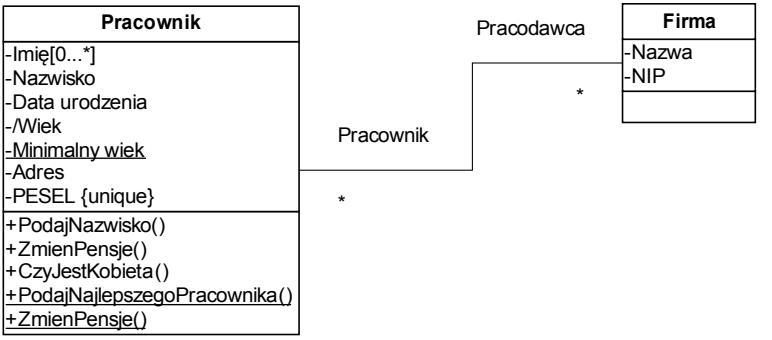
2-9 Przykład ilustrujący wykorzystanie asocjacji

Asocjacje służą do łączenia klas, które mają jakieś biznesowe zależności. Dzięki temu możemy np. zapamiętać, że *Pracownik* (klasa) *pracuje w* (asocjacja) *Firmie* (klasa). Sytuację tę ilustruje rysunek 2-9. Nazwa asocjacji powinna być tak dobrana, aby w połączeniu z nazwami klas utworzyła sensowne wyrażenie, np. Pracownik pracuje w Firmie. Domyślnie nazwa asocjacji ma poprawne znaczenie przy czytaniu od lewej do prawej. Jeżeli w powyższym przykładzie wolimy utworzyć nazwę z punktu widzenia firmy, wtedy brzmiałaby ona *zatrudnia* i należałoby ją czytać od prawej do lewej. Sytuacje tę ilustruje rysunek 2-10. Warto zwrócić uwagę na mały trójkącik przy nazwie pokazujący właściwy kierunek czytania. Podkreślmy: kierunek czytania nazwy, a nie kierunek asocjacji. Jeżeli nie zaznaczono inaczej, to asocjacja jest dwukierunkowa: w naszym przykładzie możemy przejść od firmy do pracownika oraz od pracownika do firmy.



2-10 Przykład ilustrujący wykorzystanie asocjacji. W stosunku do 2-9 zmieniono nazwę oraz jej kierunek czytania

Umieszczenie nazwy to nie jedyny sposób opisania asocjacji. Zamiast niej można wykorzystać nazwy ról (patrz rysunek 2-11). Należy pamiętać, że na diagramie są umiejscowione przy klasie docelowej, np. rola *Pracodawca* w klasie *Pracownik*. Przy takim podejściu mamy kompleksowy opis: z punktu widzenia każdej z klas. Jest to szczególnie użyteczne, gdy na dalszych etapach będziemy przekształcali diagram do wersji implementacyjnej, korzystając z narzędzia CASE. Więcej o tym będzie w kolejnych rozdziałach.



2-11 Ilustracja wykorzystania ról asocjacji

Jak zapewne zauważyli uważni czytelnicy, oprócz własności asocjacji opisanych powyżej diagram zawiera coś jeszcze. W powyższych przykładach są to znaczki „*”. Oznaczają one liczności asocjacji – spójrzmy na diagram 2-12. Notacja służąca do zdefiniowania liczności ma taki sens:

- Konkretny obiekt klasy a jest powiązany z Y obiektów klasy B,
- Konkretny obiekt klasy B jest powiązany z X obiektów klasy A.
- Zamiast X, Y wstawiamy jedną z poniższych wartości (w starszej wersji UML 1.x dopuszczano też ich kombinacje):
    - 1,
    - 0..1,
    - * oznacza zero lub więcej,
    - 1…* oznacza jeden lub więcej
    - Konkretną liczbę, np. 4.
- W starszej wersji UML (1.x) informacje o licznościach można było dowolnie ze sobą łączyć, korzystając z powyższej notacji, np. „0…4, 7, 8, 13, 45…*”. Oczywiście nie zawsze miało to biznesowy sens. Dlatego w UML2 zrezygnowano z takich „kombinowanych” liczności. Jeżeli nie podano żadnej informacji, to znaczy, że liczność wynosi 1. Zwykle stosuje się liczności „1”, „0, 1”, „1…*”.

2-12 Liczności asocjacji

Można się zastanawiać, po co w ogóle podawać informacje o licznościach. Przecież można by się umówić, że wszędzie jest „*” i możemy zapamiętać każdą sytuację biznesową. Głównym powodem jest chęć doprecyzowania modelu, ponieważ inaczej się zapamiętuje liczności typu „jeden”, a inaczej „wiele”. Dodatkową korzyścią jest pilnowanie pewnych reguł, np. jeżeli gdzieś występuje liczność „1”, to nie będziemy mogli stworzyć obiektu („system” na to nie pozwoli), który nie będzie powiązany z dokładnie jednym obiektem.

Jak, mam nadzieję, pamiętamy, wystąpieniem (instancją) klasy jest obiekt. Podobna zależność jest też dla asocjacji: jej wystąpieniem jest powiązanie (pomiędzy konkretnymi obiektami).

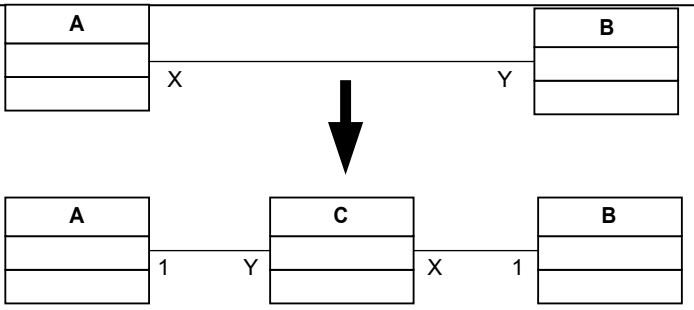
2.4.3.1 Asocjacja binarna

Asocjacja binarna jest to taka asocjacja, która występuje pomiędzy dokładnie dwoma klasami. Tak naprawdę już znamy ten rodzaj asocjacji, ponieważ to, co do tej pory napisaliśmy o asocjacjach, charakteryzuje właśnie ją.

Przydatną kwestią związaną z licznościami asocjacji binarnych jest tzw. redukcja liczności. Patrząc na klasy połączone asocjacją i jej liczności, można wyróżnić trzy główne kategorie:

- 1 do 1,
- 1 do wiele,
- Wiele do wiele.

Czasami (np. gdybyśmy chcieli przejść na model relacyjny) ta ostatnia kategoria może sprawić nam spory problem. Dlatego może zastosować przejście pokazane na rysunku 2-13. Wprowadzamy klasę pośredniczącą (C) i zamieniamy jedną asocjację „wiele-do-wielu” (pomiędzy klasami a i B) na dwie „jeden-do-wielu” (pomiędzy A-C oraz C-B). Całe przejście jest automatyczne i zawsze wykonywane według tego samego schematu. Warto zwrócić uwagę na pozorny błąd w licznościach: na nowym diagramie widzimy, że liczności X oraz Y zamieniły się miejscami. Sprawdźmy: z punktu widzenia klasy a występowała liczność Y, a z punktu widzenia klasy B była liczność X. Okazuje się, że nadal tak jest (proszę sprawdzić!). Nie zawsze da się wymyślić sensowną nazwę dla klasy A, dlatego czasami jej nazwę stanowi zlepek nazw sąsiadujących klas (np. zamiast C mogliśmy napisać AB).

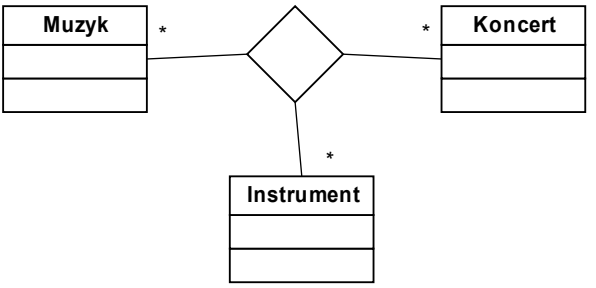


2-13 Redukcja liczności dla asocjacji

2.4.3.2 Asocjacja n-arna

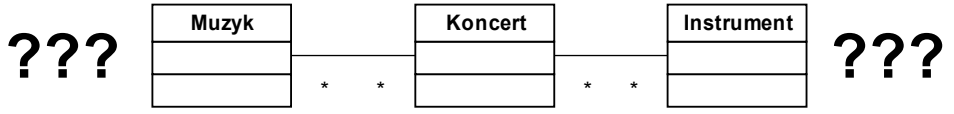
Ten rodzaj asocjacji sprawia sporo problemów i z tego powodu w praktyce jest rzadko wykorzystywany. Problemy biorą się ze sposobu określania liczności oraz biznesowego wyobrażenia sobie danej sytuacji. Najczęściej zakłada się, że asocjacja n-arna powinna mieć wszędzie liczność „wiele", a procedura szacowania tych liczności nawiązuje do asocjacji binarnych. Szacowany koniec asocjacji „konfrontujemy" z pozostałymi końcami połączonymi w jedną, wirtualną całość. Rysunek 2-14 przedstawia przykładową asocjację n-arną opisującą zależność pomiędzy muzykiem, koncertem oraz instrumentem. Nazwijmy ją (zależność oraz asocjacje) „granie". Zakładamy, że:

- na konkretnym koncercie może wystąpić wielu muzyków grających na różnych instrumentach,
- konkretny muzyk może wystąpić na wielu koncertach, grając na wielu instrumentach,
- konkretny instrument jest wykorzystywany przez wielu muzyków grających na wielu koncertach.



2-14 Przykładowa asocjacja n-arna

Pozornie mogłoby się wydawać, że wystarczy połączyć koncert z muzykiem oraz koncert z instrumentem za pomocą dwóch asocjacji binarnych. W rzeczywistości to nie wystarczy (patrz rysunek 2-15). Przy takim podejściu co prawda wiedzielibyśmy, którzy muzycy grali na danym koncercie i jakie instrumenty wykorzystano, ale nie wiedzielibyśmy, który muzyk grał na którym instrumencie (w ramach tego koncertu). Jak widać z powyższego wywodu, musimy zastosować asocjację o trzech krańcach, ponieważ angażuje ona więcej niż dwie klasy (i dlatego asocjacja binarna nie wystarczy). Na szczęście istnieje prawidłowy sposób zastąpienia asocjacji n-arnej – poznamy go w rozdziale 3 (strona 97). Jeżeli jest to biznesowo uzasadnione, stopień asocjacji (n) może być wyższy i równy 3, 4, 5 itd.



2-15 Ilustracja problemu z asocjacją n-arną

2.4.3.3 Asocjacja kwalifikowana

W odróżnieniu od asocjacji n-arnej asocjacja kwalifikowana jest dość chętnie stosowana – czasami nawet zbyt często i bez zrozumienia (a szczególnie jej implementacyjna inkarnacja – patrz podrozdział 3.2.3.4 na stronie 148). W klasycznej asocjacji binarnej, aby znaleźć konkretny obiekt znajdujący się „po drugiej stronie", musimy przeglądać wszystkie powiązania, aż znajdziemy obiekt spełniający nasze kryteria (np. pracownik z określonym numerem PESEL). I tutaj właśnie z pomocą przychodzi nam asocjacja kwalifikowana, a szczególnie kwalifikator. Spójrzmy na definicję:

kwalifikator *atrybut (lub kombinacja atrybutów) umożliwiający jednoznaczne zidentyfikowanie obiektu docelowego.*

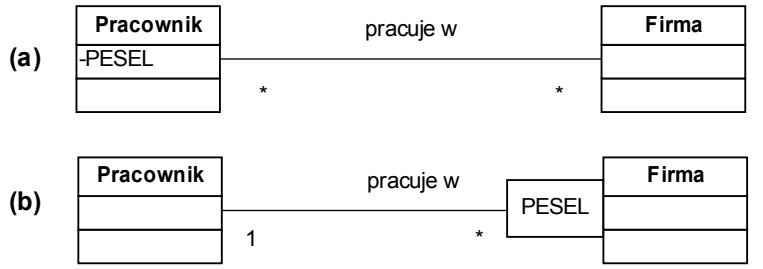
Rysunek składa się z dwóch części:

- (a) opisuje związek pracownika z firmą za pomocą klasycznej asocjacji. Chcąc odnaleźć pracownika z konkretnym numerem PESEL, musimy przeglądać wszystkich pracowników, dostępnych poprzez powiązania (jak pamiętamy, powiązanie jest wystąpieniem asocjacji) i dla każdego z nich sprawdzamy, czy posiada interesujący nas PESEL.

- (b) również opisuje związek pracownika z firmą, ale za pomocą asocjacji kwalifikowanej. Dzięki temu rozwiązaniu, mając „w ręku" konkretną wartość kwalifikatora (numer PESEL), otrzymujemy odpowiadający mu obiekt (pracownika). Warto zwrócić uwagę na dwie rzeczy:

    - Atrybut będący kwalifikatorem jest usuwany z pola atrybutów „swojej" klasy (na diagramie) i przenoszony obok klasy docelowej,

    - Ulega zmianie liczność (często też się o tym mówi redukcja liczności). Jest to spowodowane tym, że myślimy w kategoriach pary (w tym przypadku PESEL i Firma): dla konkretnej firmy i konkretnego numeru PESEL mamy powiązanie tylko do jednego pracownika; w drugą stronę: konkretny pracownik może być powiązany z wieloma takimi parami (liczność „*"), ponieważ może pracować w wielu firmach.

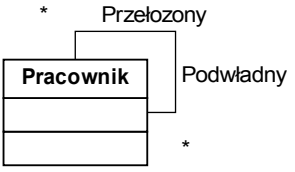


2-16 Asocjacja binarna oraz kwalifikowana

Aby kwalifikator dobrze wypełniał swoją rolę, musi być unikatowy – mówiliśmy już o tym. Ale ta unikatowość, nie musi być tak mocna jak w przypadku numeru PESEL, który jest unikalny w całej ekstensji pracowników (bo każda osoba-pracownik ma własny numer). Wystarczy, że kwalifikator będzie unikalny z punktu widzenia obiektu źródłowego (w naszym przykładzie: Firmy). Innymi słowy: mając konkretną firmę, musimy zadbać, aby jej pracownicy mieli unikatowe numery identyfikacyjne. Nic się nie stanie, jeżeli inny pracownik, ale pracujący w innej firmie będzie miał też taki numer (bo pracownika identyfikujemy z punktu widzenia konkretnej firmy).

2.4.3.4 Asocjacja rekurencyjna (zwrotna)

Asocjacja rekurencyjna (zwana też zwrotną) polega na tym, że występuje w ramach tej samej klasy. Aby móc prawidłowo zidentyfikować znaczenie każdego z końców asocjacji, obowiązkowo musimy zastosować nazwy ról. Rysunek 2-17 pokazuje klasyczny przykład umożliwiający przechowywanie informacji o zależnościach służbowych w przedsiębiorstwie (w analogiczny sposób można modelować np. informacje o stosunkach własnościowych: firma jest właścicielem innych firm).



2-17 Przykładowa asocjacja rekurencyjna (zwrotna)

Ten rodzaj asocjacji zwykle nie sprawia problemów koncepcyjnych. Należy tylko zadbać o właściwe liczności (dopuszczające „0”): w naszym

przykładzie, gdzieś na szczycie hierarchii, będzie pracownik, który nie ma szefa (ponieważ szef wszystkich szefów nie ma już szefa). Analogicznie, na samym dole będzie pracownik, który nie ma podwładnych.

2.4.3.5 Klasa asocjacji

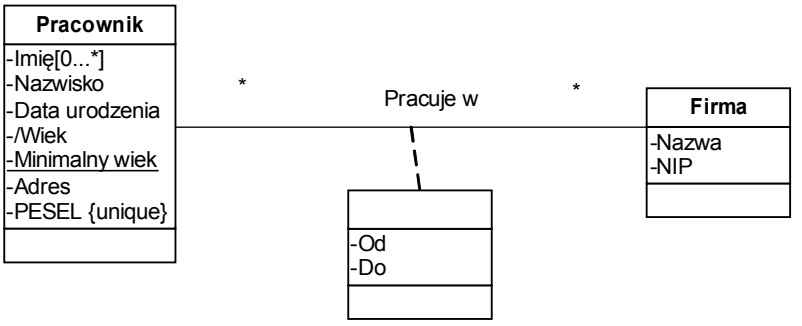
W niektórych sytuacjach zachodzi potrzeba przechowania dodatkowych informacji na temat konkretnego powiązania (wystąpienia asocjacji). Przypomnijmy sobie przykład z rysunku 2-16 część (a) (strona 42). Jak widzimy, ten diagram umożliwia nam zapamiętanie informacji o miejscach pracy poszczególnych pracowników. Załóżmy, że chcielibyśmy mieć informacje o okresie czasu, który spędzili pracując w poszczególnych firmach. Jak możemy to zrobić? Najprostszym pomysłem, jaki przychodzi do głowy, jest wstawienie odpowiednich atrybutów do którejś z klas:

- Pracownik. Załóżmy, że tak zrobiliśmy i w klasie mamy atrybuty „Data rozpoczęcia" oraz „Data zakończenia". Wygląda dobrze? Niestety nie, ponieważ pracownik może pracować w wielu miejscach, a powyższe atrybuty mogą przechować informacje tylko dla jednej pracy. W takim razie spróbujmy zastosować atrybuty powtarzalne (umożliwiają przechowywanie wielu wartości). Wygląda na to, że problem jest rozwiązany. I znowu nie: co prawda będziemy w stanie zapamiętać wiele dat rozpoczęcia i zakończenia, ale nie mamy możliwości powiązania ich z konkretnymi wystąpieniami asocjacji (ponieważ nie ma gwarancji, że kolejność poszczególnych wartości będzie stała). Innymi słowy, będziemy wiedzieli, kiedy pracownik zaczynał oraz kończył pracę, ale nie będziemy mogli tego połączyć z konkretną firmą.

- Firma. Jak zapewne wszyscy się już domyślili, wstawienie atrybutów do klasy firma spowoduje analogiczne problemy jak w przypadku klasy Pracownik.

Czyżby nie dało się tego zrobić? Oczywiście, że się da, ale musimy zastosować nową konstrukcję: klasę asocjacji (rysunek 2-18; dodaliśmy też atrybuty do istniejących klas). Cóż to takiego jest? Jest to zwykła klasa (ma takie same cechy jak każda inna), ale „podłączona" do asocjacji. Wystąpienia klasy asocjacji niosą dodatkowe informacje dotyczące konkretnego powiązania (wystąpienia asocjacji). Można o niej myśleć jak o „karteczce" przyczepionej do nitki obrazującej powiązanie.

Specjalną cechą klasy asocjacji jest ewentualny brak nazwy. Jest to o tyle korzystne, że wtedy na pierwszy rzut oka widać, iż traktowana jest ona tylko jako dodatkowa informacja dla asocjacji (która jest głównym elementem odpowiedniego fragmentu modelu). Można też do niej podłączać asocjacje oraz inne elementy charakterystyczne dla klasy. Jednakże, myślę, że należy tego unikać. Jeżeli jest to biznesowo uzasadnione i chcemy podłączyć jakąś asocjację do klasy asocjacji, to warto się zastanowić, czy nie zrobić z niej „normalnej" klasy.

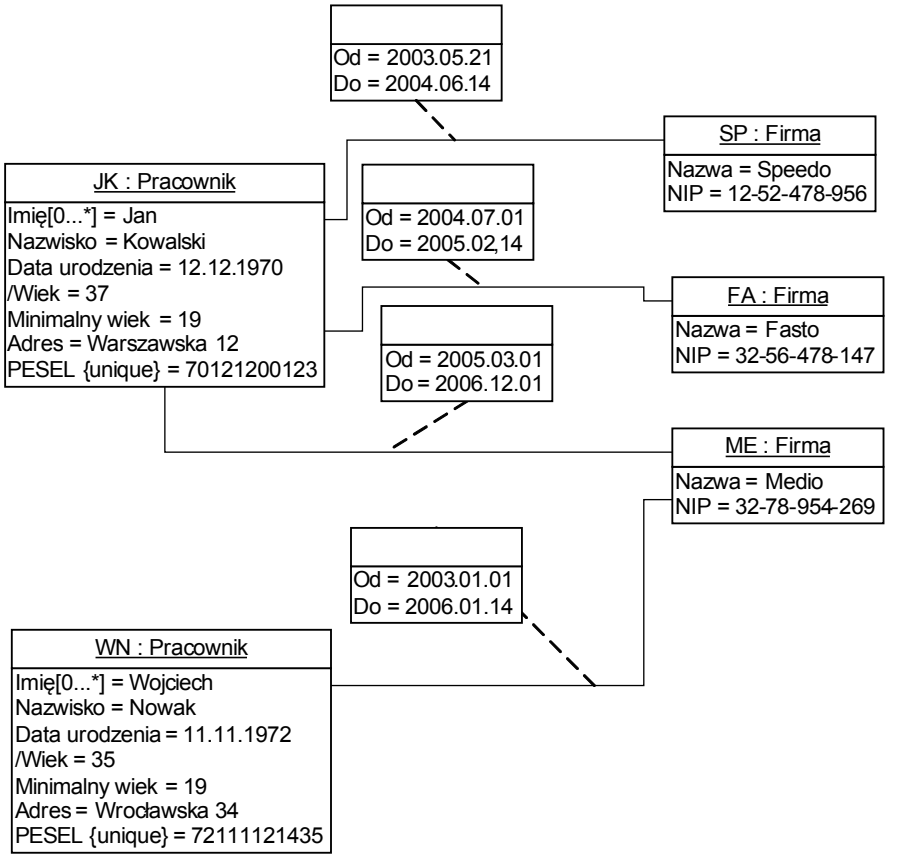


2-18 Zastosowanie klasy asocjacji

Aby ułatwić zrozumienie mechanizmu klasy asocjacji spójrzmy na rysunek 2-19, który zawiera diagram obiektów (w odróżnieniu od diagramu klas zawiera on już konkretne wystąpienia – obiekty; notacja jest wzorowana na notacji dla diagramu klas i może zawierać wartości atrybutów oraz powiązania, a nie asocjacje). Widzimy dwa wystąpienia klasy Pracownik, trzy wystąpienia klasy Firma oraz cztery instancje klas asocjacji. Na podstawie diagramu możemy powiedzieć, że:

- Jan Kowalski pracował w trzech firmach:
    - „Speedo" od 2003.05.21 do 2004.06.14,
    - „Fasto" od 2004.07.01 do 2005.02.14,
    - „Medio" od 2005.03.01 do 2006.12.01,
- Wojciech Nowak pracował tylko w firmie „Medio" od 2003.01.01 do 2006.01.14. Zwróćmy też uwagę, że jest to ta sama firma w której

pracował też Jan Kowalski. Innymi słowy, konkretny obiekt może być połączony z wieloma innymi obiektami. Naturalnie tylko wtedy gdy liczności na to pozwalają – w naszym przykładzie mamy „wiele do wielu" (rysunek 2-18), więc wszystko jest OK.



2-19 Diagram obiektów ilustrujący wykorzystanie klasy asocjacji

2.4.3.6 Agregacja i kompozycja

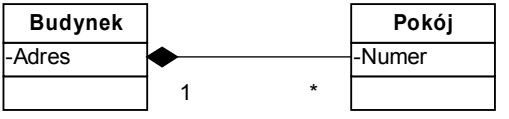
Agregacja i kompozycja są szczególnymi rodzajami asocjacji (głównie binarnych). Ich dodatkowym zadaniem, oprócz modelowania jakiejś zależności, jest podkreślanie związku typu część-całość. Rysunek 2-20 przedstawia przykładową agregację opisującą grupę składająca się z pracowników.

Symbol rombu, oznaczający właśnie agregację, umieszczamy przy „całości”. Możemy również podać informację dotyczącą liczności. Jeżeli chodzi o nazwę, to zwykle ją pomijamy, ponieważ wykorzystanie agregacji umożliwia odczytywanie tego związku jako: „składa się”, „zawiera”, „jest częścią” itp.

2-20 Przykładowa agregacja

Mocniejszą formą agregacji (a więc również asocjacją) jest kompozycja. O ile zamodelowanie agregacji nie niesie żadnych specjalnych konsekwencji, to wykorzystanie kompozycji oznacza, że:

- Część nie może być współdzielona (można to zapamiętać, myśląc, że kompozycja jest „zazdrosna”),
- Część nie może istnieć bez całości,
- Usunięcie całości oznacza też usunięcie części. Natomiast usunięcie części nie musi oznaczać usunięcia całości.



2-21 Przykładowa kompozycja

Warto zwrócić uwagę, że powyższe cechy skutkują pewnymi konkretnymi licznościami. Rysunek 2-21 zawiera przykładową kompozycję (zamalowany romb) opisującą budynek oraz jego pokoje. Ten przykład jest o tyle dosłowny, że z fizycznego punktu widzenia nie może istnieć pokój bez budynku. Nie zawsze tak musi być: można również zastosować kompozycję

w przykładzie z rysunku 2-20, jeżeli tylko mamy do tego biznesowe uzasadnienie:

- Pracownik należy tylko do jednej grupy. Warto zwrócić uwagę, że jeżeli chcielibyśmy trzymać historię przydziałów do grup, to docelowa liczność „1" nam na to nie pozwoli. Innymi słowy: nie będzie mógł istnieć obiekt klasy Pracownik połączony z wieloma obiektami klasy Grupa.

- W modelowanej firmie nie może być pracownika przypisanego do grupy,

- Usunięcie grupy oznacza też usunięcie jej pracowników.

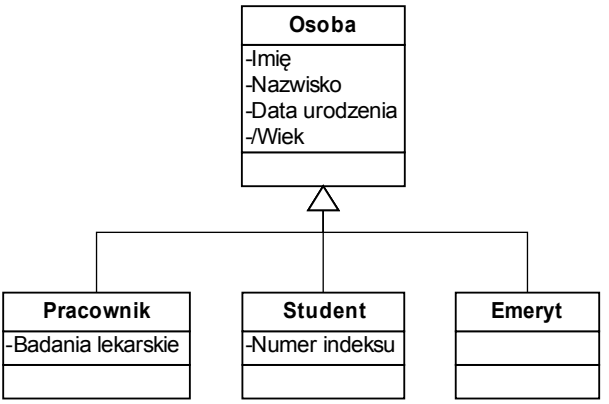
2.4.4 Dziedziczenie

Dziedziczenie jest jednym z najważniejszych pojęć w obiektowości i oznacza pewną zależność pomiędzy klasami. Owa zależność umożliwia utworzenie nowej klasy (mówimy „podklasy") na podstawie „starej" (nadklasy). Podklasa posiada wszystkie cechy swojej nadklasy plus, ewentualnie, swoje własne. Z punktu widzenia modelowania podklasa jest szczególnym przypadkiem nadklasy, czyli można dziedziczyć klasę Pracownik z klasy Osoba (bo pracownik też jest osobą, chyba że opisujemy fabrykę robotów). W świetle tego błędne jest dziedziczenie, np. pokoju z domu (bo pokój nie jest szczególnym przypadkiem domu).

Następne podrozdziały zawierają krótkie omówienie poszczególnych rodzajów dziedziczenia i tematów z tym związanych. Bardziej szczegółowe informacje można znaleźć w [Płod05], [Fowl04] czy [Wryc05].

2.4.4.1 Dziedziczenie pojedyncze

Dziedziczenie pojedyncze jest najprostszym rodzajem dziedziczenia. Rysunek 2-22 zawiera typowy przykład opisujący hierarchię dziedziczenia dla klasy Osoba. Przeanalizujmy ten diagram:



2-22 Przykładowa hierarchia dziedziczenia

- Sprawdźmy, czy w ogóle wolno było nam zastosować dziedziczenie: czy wszystkie podklasy są szczególnymi przypadkami nadklasy? Oczywiście, że tak: pracownik jest osobą, podobnie ze studentem i emerytem.

- Po co w ogóle zastosowaliśmy takie dziedziczenie? Przecież te wszystkie dedykowane atrybuty można umieścić w nadklasie i też by to „zadziałało". Naturalnie, że tak, ale:

    - W zależności od tego, jaki rodzaj osoby opisywalibyśmy, atrybuty specyficzne dla innych byłyby „niewykorzystane" (m.in. zajmowałyby pamięć),

    - Diagram słabiej opisywałby nasz biznes, ponieważ nie dałoby się stwierdzić, że występują tam: emeryt, student oraz pracownik (a przynajmniej nie na pierwszy rzut oka),

    - Nie moglibyśmy wykorzystać polimorfizmu metod i przesłaniania (będzie o tym dalej – patrz podrozdział 2.4.4.2).

- Klasa Emeryt wydaje się pusta i czy w związku z tym jest nam potrzebna? „Wydaje się" jest dobrym określeniem, ponieważ w rzeczywistości nie jest „pusta". Niesie ze sobą ważną informację – ta osoba jest emerytem.

Warto podkreślić, że ogólnie rzecz biorąc (to nie jest do końca prawda – wrócimy do tego w następnych rozdziałach), dziedziczymy wszystkie elementy znajdujące się w klasie:

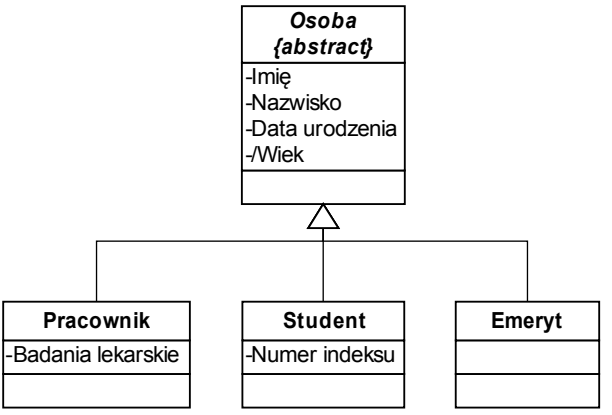
- Atrybuty,
- Metody,
- I coś jeszcze – ktoś wie? Oczywiście asocjacje – warto o tym szczególnie pamiętać, ponieważ ci, którzy unikają „pustych" podklas (patrz wyżej), o tym zapominają.

2.4.4.2 Klasa abstrakcyjna i polimorfizm metod

Spójrzmy na definicję:

Klasa abstrakcyjna — *Klasa, która nie może mieć bezpośrednich wystąpień (nie mogą istnieć obiekty należące do tej klasy).*

Myślę, że definicja jest jasna i nie stwarza problemów ze zrozumieniem. Można się tylko zastanawiać, po co nam klasa, która nie może mieć obiektów? Otóż przyda nam się (i to bardzo) do tworzenia hierarchii dziedziczenia. W systemie opisanym na powyższym przykładzie (rysunek 2-22) nie będą występowały osoby jako takie, ale ich specjalizacje: student, pracownik itd. Oznaczając klasę osoba jako abstrakcyjną, mamy pewność, że nikt (celowo lub omyłkowo) nie stworzy takich obiektów. Robimy to, pisząc jej nazwę czcionką pochyłą (co nie jest zbyt czytelne) lub używając wartości etykietowanej {abstract} (rysunek 2-23).



2-23 Ilustracja wykorzystania klasy abstrakcyjnej

Klasa abstrakcyjna może zwierać metody konkretne (tak jak każda inna klasa), ale również i abstrakcyjne. Te ostatnie możemy zdefiniować w następujący sposób:

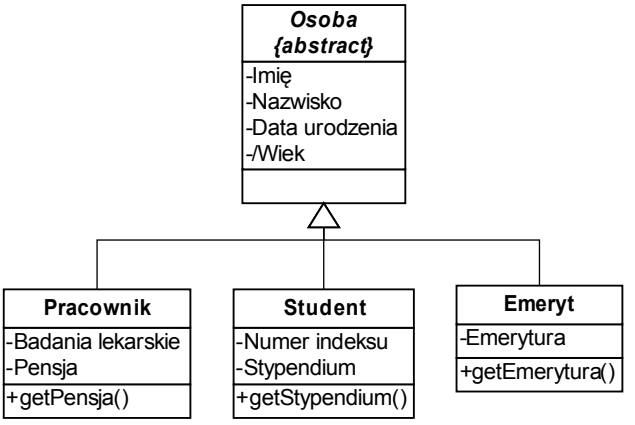
Metoda abstrakcyjna — *Metoda, która posiada tylko deklarację, ale nie posiada definicji (ciała).*

Podobnie jak przy okazji klasy abstrakcyjnej, można się zastanawiać, po co nam metoda, która nie ma ciała? Zanim odpowiemy na to pytanie, spróbujmy się zastanowić nad następującym problemem:

- Załóżmy, że osoby z diagramu pokazanego na rysunku 2-23 mają jakieś dochody:
    - Pracownik ma pensję,
    - Student ma stypendium,
    - Emeryt ma emeryturę.
- I chcielibyśmy mieć jakiś sposób zapytania o te dochody.

Najprostszym sposobem wydaje się umieszczenie atrybutów w poszczególnych klasach i dodanie odpowiednich metod – tak jak pokazano na rysunku 2-24. W zależności od rodzaju osoby, wywołamy odpowiednią metodę

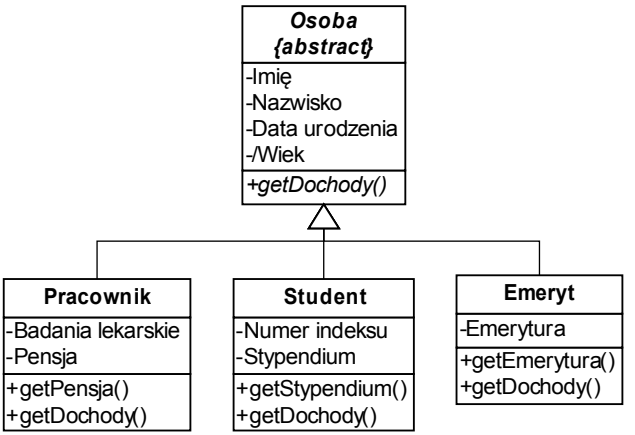
(nazwy metod zawierają przedrostek „get”, który informuje nas, że metoda coś zwraca – więcej będzie o tym w podrozdziale 4.1.1 na stronie 280).



2-24 Hierarchia dziedziczenia z dodanymi metodami

Wydaje się, że to zadziała, ale czy nie można zrobić tego lepiej? Skoro wszystkie są osobami i wszystkie mają dochody to może umieścimy metodę (np. `getDochody()`) we wspólnej nadklasie (Osoba)? No tak, ale ta metoda będzie miała takie samo znaczenie, ale sposób działania już inny. Co w niej więc umieścić? Najlepiej nic… i tutaj właśnie możemy zastosować metodę abstrakcyjną – czyż to nie właśnie ona ma deklarację, ale nie ma ciała? Ciało metody, właściwe dla poszczególnych przypadków biznesowych, umieścimy w odpowiednich podklasach (patrz rysunek 2-25; zwróć uwagę na sposób pisania nazwy metody abstrakcyjnej - kursywa):

- dla Pracownika metoda zwróci wartość atrybutu `Pensja` lub wykorzysta istniejącą metodę `getPensja()`,
- dla Studenta metoda zwróci wartość atrybutu `Stypendium` lub wykorzysta istniejącą metodę `getStypendium ()`,
- dla Emeryta metoda zwróci wartość atrybutu `Emerytura` lub wykorzysta istniejącą metodę `getEmerytura()`.



2-25 Hierarchia dziedziczenia z dodanymi metodami zwracającymi dochody

Ktoś dociekliwy mógłby zapytać, co się stanie, jak stworzymy obiekt zawierający metodę abstrakcyjną i zechcemy jej użyć? Pytanie jest zasadne bo, przecież ona nie ma ciała, więc nie da się jej użyć. Otóż twórcy obiektowości pomyśleli o tym i zdecydowali, że metody abstrakcyjne mogą być tylko w klasach abstrakcyjnych – w klasach konkretnych są po prostu zabronione. W związku z tym nie dojdzie do sytuacji, która nas zaniepokoiła (bo przypomnijmy: klasy abstrakcyjne nie mogą mieć bezpośrednich wystąpień).

Ważnym zagadnieniem, które jest nierozerwalnie połączone z dziedziczeniem, jest przesłanianie oraz polimorfizm metod (istnieje jeszcze kilka innych rodzajów polimorfizmu, ale nie będziemy się nimi zajmować). Aby to dobrze wyjaśnić, najłatwiej jest odwołać się do przykładów z języków programowania. Zrobimy to w podrozdziale 3.3.2 (strona 176). Na razie tylko powiedzmy, że:

- przesłanianie umożliwia istnienie wielu wersji metody o takiej samej nazwie (i liczbie oraz rodzaju parametrów) rozmieszczonych w poszczególnych klasach tej samej hierarchii dziedziczenia (analogicznie jak nasza metoda `getDochody()`),

- polimorfizm metod, a właściwie polimorficzne wołanie metody, umożliwia odpowiednie wybranie metody do wywołania.

W podrozdziale 2.4.2 (strona 26) pisaliśmy o ekstensji klasy, że w połączeniu z dziedziczeniem rodzi pewne problemy. Teraz, gdy już wiemy, na

czym polega dziedziczenie, jest dobry moment, aby napisać, czego dotyczy ten problem. Otóż, jak pamiętamy, obiekty z podklasy są pośrednio też wystąpieniami nadklasy (Pracownik jest też Osobą). I to jest właśnie ten problem. Pewnie trochę za bardzo skróciłem – oto pełniejsze wyjaśnienie:

- mamy ekstensję klasy Osoba, która zawiera wszystkie jej obiekty, np. taką, którą zawiera rysunek 2-22 (oczywiście zakładamy, że klasa Osoba nie jest abstrakcyjna),
- klasa Pracownik z niej dziedziczy, więc Pracownik też jest osobą.
- Czy w związku z powyższym twierdzeniem powinien być w ekstensji klasy Osoba?
- Czy w związku z powyższymi twierdzeniami obiekty klasy Osoba powinny być w ekstensji klasy pracownik (to chyba wydaje się najmniej właściwe)?

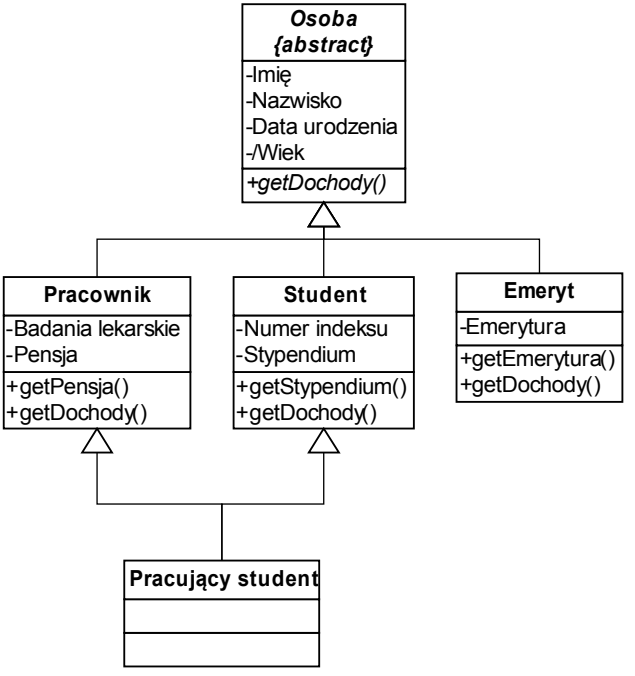
Zdania naukowców są podzielone. Najbardziej popularny pogląd jest taki, że obiekty klasy Pracownik powinny być w ekstensji klasy Osoba, ale „przykrojone" do zestawu cech osoby (bez dodatkowych atrybutów, metod i asocjacji). W praktyce możemy sami zdecydować, jakie mamy biznesowe potrzeby i wybrać właściwe rozwiązanie. Wrócimy do tego zagadnienia w podrozdziale 3.3.8 na stronie 204.

2.4.4.3 Dziedziczenie wielokrotne

Dziedziczenie wielokrotne zwane także wielodziedziczeniem różni się od poprzedniego rodzaju jednym faktem: dziedziczymy z więcej niż jednej nadklasy.

Ta jedna, z pozoru niewielka zmiana, powoduje spory problem. Przeanalizujmy przykład przedstawiony na rysunku 2-26. W stosunku do tego, co było na diagramie 2-25, dodaliśmy klasę dziedziczącą z Pracownika oraz Studenta i nazwaliśmy ją pracujący student. Najpierw sprawdźmy, czy wolno nam tak zrobić. Czy podklasa jest szczególnym przypadkiem obydwóch nadklas? Naturalnie, że tak: pracujący student jest i pracownikiem, i studentem. Skoro już to sprawdziliśmy, to wróćmy do tego problemu, o którym wspomnieliśmy parę zdań wcześniej. Spójrzmy: zgodnie z zasadami dziedziczenia z nadklas otrzymujemy wszystkie inwarianty (stałe elementy takie jak atrybuty, metody, asocjacje). W obydwóch nadklasach znajdują się takie same atrybuty (odziedziczone z klasy osoba): imię, nazwisko itd. W związku

z tym, klasa pracujący student dziedziczy jakby dwa komplety atrybutów, a raczej nie chcemy mieć w klasie dwóch nazwisk, dwóch imion itp. Kłopot z atrybutami można próbować jakoś rozwiązać, myśląc w ten sposób: co to znaczy, że dziedziczymy atrybut? Oznacza to zdolność do przechowania pewnej wartości, czyli jeżeli klasa posiada już tę zdolność (np. atrybut imię), to umieszczanie kolejnych takich samych atrybutów możemy próbować zignorować (bo wszystkie one są takie same). Poważniejszy problem występuje z metodami, ponieważ one mogą się różnić (i przeważne tak jest). W naszym przykładzie mamy dwie wersje metody `getDochody()` (pochodzącą ze studenta oraz pracownika) – którą z nich wybrać? Pod względem biznesowym żadna nie jest odpowiednia. Powinniśmy stworzyć kolejna wersję, która zsumuje dochody pochodzące z „części pracowniczej" oraz „części studenckiej".



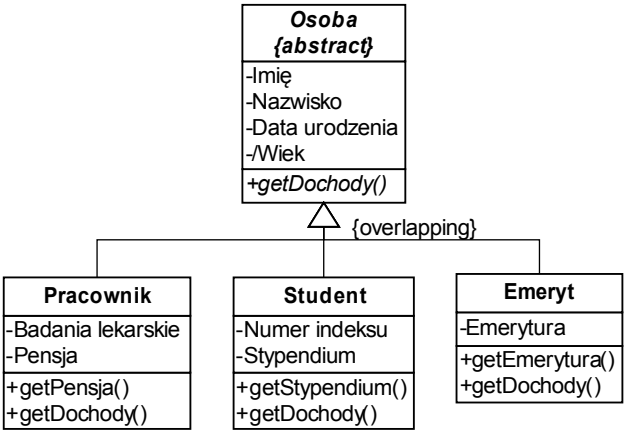
2-26 Przykładowa hierarchia wielodziedziczenia

Jakie w takim razie jest rozwiązanie powyższego problemu z dublowaniem się inwariantów? Niestety, nie ma jednego dobrego sposobu – zwykle

musimy dokładnie określić, o który element nam chodzi, albo ręcznie zlikwidować konflikt.

2.4.4.4 Dziedziczenie typu overlapping

Klasyczne dziedziczenie, to o którym mówiliśmy w podrozdziale 2.4.4.1 (strona 47) nosi miano rozłącznego (ang. *disjoint*), ponieważ obiekt należy albo do jednej klasy, albo do drugiej. Jego przeciwieństwem jest dziedziczenie typu *overlapping*. Koncepcyjnie jest trochę zbliżone do wielodziedziczenia.



2-27 Dziedziczenie typu *overlapping*

Spójrzmy na przykład z rysunku 2-27. Wygląda prawie tak samo jak rysunek 2-25. I w tym przypadku *prawie* robi wielką różnicę. Napis *overlapping* oznacza, że w systemie możemy spodziewać się obiektów należących do kilku klas: w tym przypadku obiekt może należeć do klasy Pracownik i/lub Student i/lub Emeryt. Może być ich dowolną kombinacją, co oznacza, że będzie miał ich wszystkie cechy. Warto zwrócić uwagę, że może to prowadzić do podobnych problemów jak przy okazji wielodziedziczenia.

2.4.4.5 Dziedziczenie wieloaspektowe

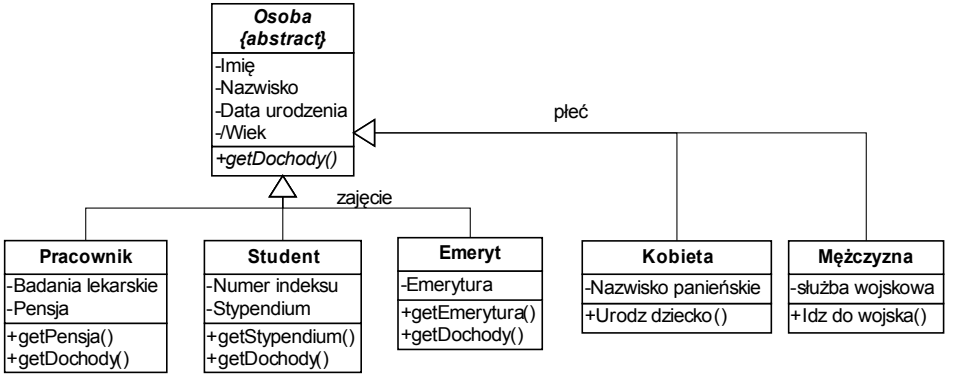
Omawiane do tej pory rodzaje dziedziczenia uwzględniały tylko jeden aspekt (kryterium podziału). W większości przypadków jest to wystarczające, ale nie zawsze. Wróćmy do przykładu z diagramu 2-25 (strona 52). Załóżmy, że chcielibyśmy uwzględniać płeć wymienionych tam osób. I w zależności od niej przechowywać pewne dodatkowe informacje oraz wykony-

wać specyficzne operacje. Dla kobiet będzie to nazwisko panieńskie i urodzenie dziecka, a dla mężczyzn stosunek do służby wojskowej i pójście do wojska. Zastanówmy się, jak możemy to zrealizować:

- W klasie osoba umieścić odpowiednie atrybuty (i dla mężczyzn, i dla kobiet) oraz metody. Jeżeli obiekt będzie opisywał mężczyznę, to „kobiece' cechy nie będą wykorzystywane, i odwrotnie. Jak łatwo się domyślić nie jest to najlepszy sposób,

- Zmodyfikować istniejące podklasy i dodać nowe, uwzględniając płcie: „Pracownik Kobieta", „Pracownik Mężczyzna", „Student Kobieta", „Student Mężczyzna" itd. To też chyba nie jest najlepszy pomysł: wielokrotnie dublujemy atrybuty i metody. Dla naszych trzech klas i dwóch płci to nie prowadzi do dużej ilości kombinacji, ale gdybyśmy potrzebowali dokonać podziały dla trzech kategorii, których każda miałaby po trzy opcje, to liczba robi się już pokaźna.

Jak łatwo się domyślić, z pomocą przyjdzie nam właśnie dziedziczenie wieloaspektowe. Spójrzmy na rysunek 2-28. Dodaliśmy dwie klasy: kobieta oraz mężczyzna. Zawierają one odpowiednie atrybuty oraz metody. Nazwy aspektów umieszczamy przy znaku dziedziczenia (są one obowiązkowe).



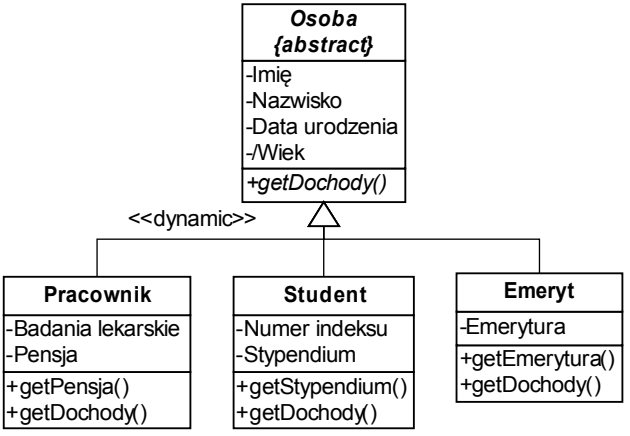
2-28 Przykładowe dziedziczenie wieloaspektowe

Widząc taki diagram, możemy spodziewać się obiektów należących do klasy, np. pracownik-Kobieta, Emeryt-Mężczyzna itd. Zasada jest prosta: bierzemy po jednej, dowolnej klasie z każdego aspektu (chyba że występuje

tam np. *overlapping*, to wtedy możemy wykorzystać dowolną kombinację klas z tego aspektu) i je ze sobą „sklejamy".

2.4.4.6 Dziedziczenie dynamiczne

Jak do tej pory, wszystkie obiekty należące do omawianych klas były „stałe w uczuciach": jak zostały stworzone jako Pracownik, to kończyły swój komputerowy żywot również jako Pracownik. Nie jest to najwygodniejsze podejście. W życiu bywa tak, że ktoś jest np. studentem, później ta sama osoba jest pracownikiem, a następnie np. emerytem. Skoro modelujemy prawdziwe biznesowe sytuacje, dobrze by było mieć mechanizm, który to umożliwia. I do tego właśnie służy dziedziczenie dynamiczne. Spójrzmy na diagram przedstawiony na rysunku 2-29. Stereotyp <<dynamic>> oznacza to, o co nam właśnie chodziło: dziedziczenie dynamiczne, czyli obiekty z podklas mogą dowolnie zmieniać swoja przynależność.



2-29 Przykład dziedziczenia dynamicznego

2.4.5 Ograniczenia

W czasie omawiania rodzajów atrybutów wspomnieliśmy o specjalnym rodzaju wyrażeń UML zwanych ograniczeniami (strona 31). Jest to jeden z mechanizmów rozszerzalności, umożliwiający doprecyzowanie modelu. Innymi słowy, wszystkie informacje, których nie jesteśmy w stanie umieścić na diagramie w „standardowy" sposób, możemy zapisać jako ograniczenia. W tym celu możemy korzystać ze specjalnego języka OCL (*Object Constraint Language*), wyrażeń matematycznych (arytmetyka, logika), zwykłego

tekstu czy stosując kombinacje wymienionych technik. Niezależnie od wybranej techniki treść ograniczenia umieszczamy w nawiasach klamrowych.

Ograniczenia można podzielić na:

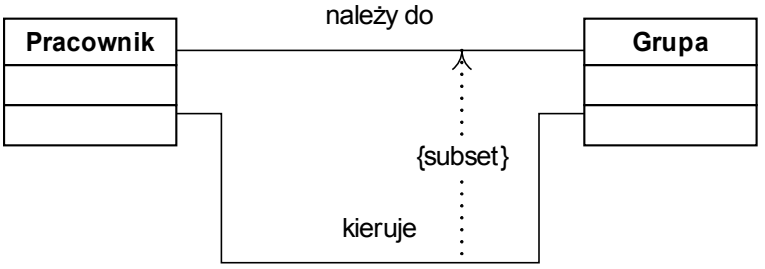
- Dynamiczne. Przy sprawdzaniu warunku istotny jest poprzedni stan elementu, którego ograniczenie dotyczy. Przykładami (zresztą dość optymistycznymi) mogą być: „{pensja nie może zmaleć}" czy „{wzrost pensji musi być większy niż 10%}".

- Statyczne. Przy sprawdzaniu warunku poprzedni stan elementu, którego ograniczenie dotyczy, nie ma znaczenia, np. „{pensja > 2000,00 PLN}".

Istnieje też pewna predefiniowana grupa ograniczeń (sformułowana przez twórców notacji UML), którą się teraz zajmiemy.

2.4.5.1 Ograniczenie {subset}

Ograniczenie {subset} może być nakładane na dwie asocjacje (lub agregacje). Spójrzmy na rysunek 2-30. Aby można było utworzyć powiązanie w ramach asocjacji B („kieruje"), musi już istnieć powiązanie w ramach asocjacji a („należy do"). Oczywiście obydwie asocjacje powinny być pomiędzy tymi samymi klasami oraz obydwa powiązania powinny być pomiędzy tymi samymi obiektami.



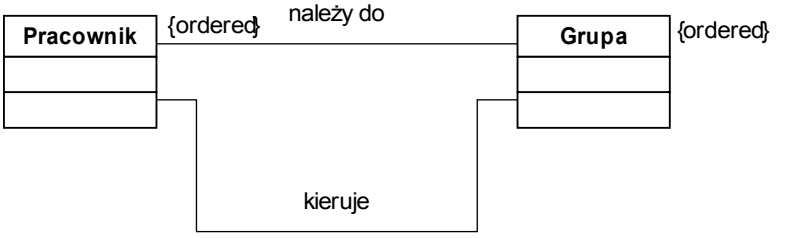
2-30 Ilustracja ograniczenia {subset}

2.4.5.2 Ograniczenie {ordered}

Ograniczenie {ordered} może dotyczyć (rysunek 2-31):

- Asocjacji. W takim przypadku oznacza, że powiązania są przechowywane (oraz otrzymywane i przetwarzane) w pewnej ustalonej kolejności.

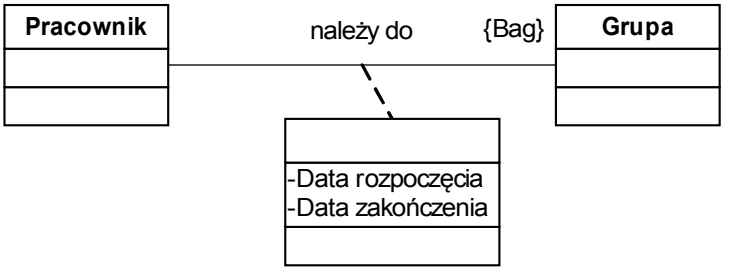
- Klasy. W tej sytuacji obiekty w ekstensji są przechowywane (oraz otrzymywane i przetwarzane) w pewnej ustalonej kolejności.



2-31 Ilustracja ograniczenia {ordered}

2.4.5.3 Ograniczenie {bag} oraz {history}

Ograniczenia {bag} oraz {history} są do siebie podobne. Obydwa umożliwiają przechowywanie duplikatów elementów (rysunek 2-32). W przypadku asocjacji oznacza to, że może istnieć wiele powiązań pomiędzy tymi samymi obiektami. Taka sytuacja jest niedozwolona dla klasycznych asocjacji.

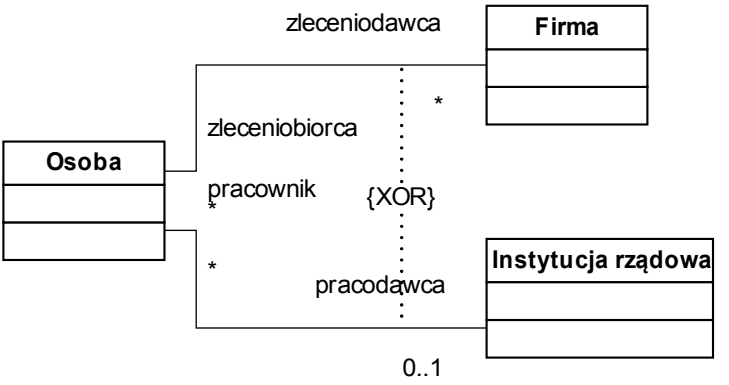


2-32 Ilustracja ograniczenia {bag}

Różnica pomiędzy {bag} oraz {history} jest dość płynna. Obydwa ograniczenia pozwalają na duplikaty powiązań, ale {history} podkreśla aspekt zmian w czasie.

2.4.5.4 Ograniczenie {xor}

Ograniczenie {xor} dotyczy co najmniej dwóch asocjacji. Zapewnia, że będzie istniało tylko jedno powiązanie w ramach asocjacji, które ogranicza.



2-33 Ilustracja ograniczenia {xor}

Na rysunku 2-33 mamy odpowiedni przykład. Jeżeli osoba pracuje w instytucji rządowej, to nie może współpracować z firmą i na odwrót.

2.4.6 Diagram klas dla wypożyczalni wideo

Gdy szczęśliwie przebrnęliśmy przez wszystkie podrozdziały dotyczące diagramu klas, możemy wreszcie się zabrać do skonstruowania diagramu dla naszej wypożyczalni.

Długo się zastanawiałem, w jaki sposób przedstawić ten diagram – brałem pod uwagę dwie możliwości:

- Zaprezentowanie finalnej wersji i skomentowanie jej,
- Pokazanie krok po kroku sposobu dochodzenia do postaci spełniającej wszystkie nasze oczekiwania.

Ostatecznie wybrałem to drugie podejście. Umożliwia ono mniej zaawansowanym czytelnikom prześledzenie całego procesu, razem z informacjami, dlaczego pewne konstrukcje nie są odpowiednie (czy wręcz błędne). Oznacza to, że w trakcie tworzenia będziemy czasami podejmowali też błędne decyzje projektowe, po to by lepiej zilustrować właściwe rozwiązania.

Osoby zaawansowane mogą od razu przeskoczyć do finalnej wersji (strona 93).

Powiadają, że najtrudniej jest zacząć coś robić, a później już jakoś idzie. W przypadku tworzenia diagramu klas mamy do dyspozycji co najmniej dwa sposoby postępowania:

- Tworząc diagram, będziemy mieli w myślach wszystkie wymagania i na bieżąco je uwzględniali,

- Krok po kroku czytamy wymagania i przekształcamy je w odpowiednie konstrukcje diagramu klas, bazując tylko na tym, co aktualnie odczytaliśmy z wymagań.

Ten drugi sposób jest bardziej pracochłonny i przeznaczony dla mniej doświadczonych/zaawansowanych analityków. I właśnie dlatego z niego skorzystamy. Poniżej zamieszczamy numer odnoszący się do poszczególnych punktów wymagań dla wypożyczalni wideo (strona 11), numer strony oraz odpowiedni komentarz.

- 1, 2, 3 (str. 11) Z naszych wymagań wynika, że w wypożyczalni możemy mieć do czynienia z dwoma rodzajami klientów:

  - Osoba prywatną,

  - Firmą.

  Dla każdego z nich przechowujemy specyficzny rodzaj informacji. Gdyby nie to wymaganie, można by się pokusić o stworzenie jednej klasy klient, w której umieścilibyśmy atrybut informujący nas o rodzaju klienta. Niestety, w tej sytuacji należy stworzyć jedną nadklasę Klient oraz odpowiednie podklasy: Osoba oraz Firma (rysunek 2-34). Niektórzy analitycy nie zgodziliby się z takim przedstawieniem tej sytuacji. Dlaczego? Wydaje się, że wszystko jest w porządku: w naszej wypożyczalni osoba jest specjalnym rodzajem klienta i firma też jest specjalnym rodzajem klienta. Wątpliwości może budzić fakt umieszczenia w jednej hierarchii dziedziczenia i firmy i osoby. Gdybyśmy podzielali te wątpliwości, to jak powinniśmy postąpić? Możemy użyć agregacji lub wręcz kompozycji (rysunek 2-35). No tak, a jeżeli ktoś utworzy i jedno, i drugie połączenie w tej samej chwili? Przecież będzie to sprzeczne z naszymi

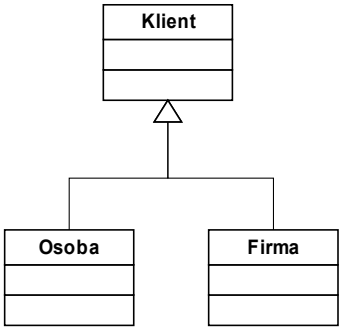
uwarunkowaniami biznesowymi. Na szczęście możemy użyć specjalnej konstrukcji, która nas przed tym uchroni: ograniczenie XOR (zwykły OR dopuszcza istnienie dwóch powiązań). Odpowiedni diagram zawiera rysunek 2-36. Warto jeszcze zwrócić uwagę, że nie podłączyliśmy klas: Osoba oraz Klienta, ale klasy: Informacje o osobie i Informacje o firmie. Jest to spowodowane tym, że klient nie składa się (bo tak odczytujemy kompozycję) m.in. z osoby czy z firmy, ale z informacji o nich.

Zatem mamy dwa rozwiązania: które z nich wybrać? Oczywiście (tradycyjnie już) nie ma idealnej odpowiedzi. Każdy analityk, po jakimś czasie, ma swoje przyzwyczajenia, różne intuicje i m.in. na tej podstawie podejmuje decyzje. Rozwiązanie z kompozycjami oznacza dla nas dodatkową pracę; będziemy musieli jakoś zaimplementować funkcjonalność, która umożliwi łatwy dostęp do informacji o kliencie, w zależności od jego rodzaju (sięgniemy do jednej lub drugiej części). Generalnie dziedziczenie jest dużo wygodniejsze: to, co musimy oprogramować ręcznie w przypadku kompozycji, tutaj mamy „za darmo". Dlatego zdecydujemy się na rozwiązanie z dziedziczeniem.

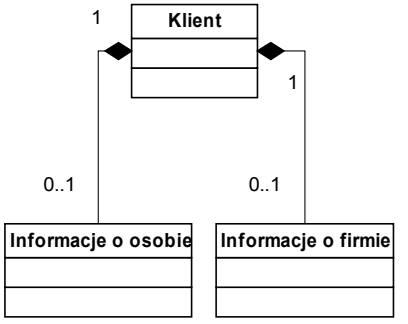
Celem dziedziczenia jest m.in. zunifikowanie posługiwania się pewnymi obiektami, bez wchodzenia w zbędne szczegóły. Zauważmy, że w naszym rozwiązaniu mamy taką oto sytuację: aby np. wyświetlić informacje o kliencie będącym osobą, odczytujemy imię i nazwisko, a w przypadku klienta-firmy, pobieramy nazwę. Jak temu zaradzić? Wykorzystamy kilka cech obiektowości, które już znamy; polimorficzne wołanie metod oraz przesłanianie (patrz podrozdział 2.4.4.2, strona 49). Stworzymy abstrakcyjną metodę o nazwie, np. `getEtykieta()` (ang. *getLabel*) i przesłonimy ją w podklasach tak, aby zwracała właściwe dane.

Warto jeszcze zwrócić uwagę na ostatni omawiany tu punkt wymagań: czy należy przechowywać informację o wieku klientów? Jeżeli ktoś ma mniej niż 16 lat, to po prostu go nie zapisujemy do wypożyczalni i dlatego informacja o nim nie znajdzie się w systemie. Wrócimy do tej kwestii później.

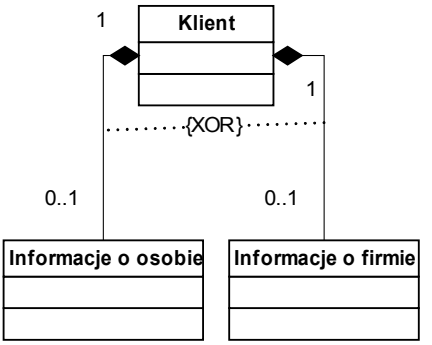
Prawidłowy fragment diagramu klas, uzupełniony o atrybuty i metody, przedstawiony jest na rysunku 2-37.



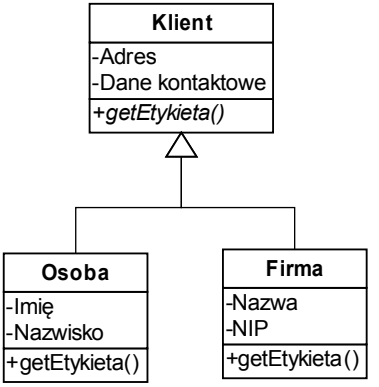
2-34 Tworzenie diagramu klas dla wypożyczalni – krok 1



2-35 Tworzenie diagramu klas dla wypożyczalni – krok 2



2-36 Tworzenie diagramu klas dla wypożyczalni – krok 3



2-37 Tworzenie diagramu klas dla wypożyczalni – krok 4

- 4 (str. 12) Najprostszym sposobem zapamiętania informacji, o których mowa w tym punkcie jest stworzenie dwóch klas: kaseta oraz płyta razem z odpowiednimi atrybutami (patrz rysunek 2-38). Niestety, w tym przypadku najprostszy nie znaczy najlepszy. Zastanówmy się dlaczego? Najważniejszym powodem jest to, że informacje o filmach będą się powtarzały (ponieważ prawdopodobnie będziemy mieli wiele kaset/płyt z tym samym filmem). W takim razie stwórzmy klasę, która będzie przechowywała informacje o filmach i połączmy ją z informacjami o kasetach i płytach. Rysunek 2-39 odzwierciedla ten pomysł. Nadal nie

wygląda to dobrze – dlaczego? Pamiętasz naszą chęć pozostawania na stosunkowo wysokim poziomie abstrakcji i niewchodzenia w szczegóły? Tutaj nadal mamy z tym problem. Oddzielnie połączyliśmy film z kasetą oraz płytą. Z punktu widzenia filmu, do pewnego momentu, ważne jest tylko, że ten film jest na jakimś nośniku, nie jest istotne, czy to będzie kaseta, czy płyta. No właśnie: nośnik! Zróbmy wspólną nadklasę dla kasety oraz płyty i to ją połączmy z filmem. Czy tak można? Sprawdźmy: czy kaseta jest szczególnym rodzajem nośnika? Czy płyta jest szczególnym rodzajem nośnika? w obydwóch przypadkach odpowiedź jest twierdząca, a zatem można to tak zrealizować. Od razu odpowiedzmy na pytanie, które może przyjść niektórym osobom do głowy: czy można dziedziczyć film z kasety (lub odwrotnie)[5]? NIE, ponieważ ani film nie jest szczególnym rodzajem kasety, ani kaseta nie jest szczególnym rodzajem filmu.

Rysunek 2-40 przedstawia poprawny diagram, uwzględniający powyższe rozważania. Można się tylko jeszcze zastanowić nad licznościami:

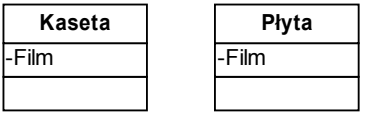
- Napisaliśmy, że konkretny nośnik może zawierać jeden film. Nie zawsze może to być prawdą, np. na jednej płycie może być kilka odcinków serialu, z których każdy jest katalogowany jako oddzielny tytuł. Tego typu decyzję biznesową należy uzgodnić ze zleceniodawcą.

- Z diagramu wynika, że obiekt klasy Film może nie być połączony z obiektem klasy Nośnik. Innymi słowy, w systemie będą filmy, dla których nie mamy kaset/płyt. Może to mieć sens, ponieważ przechowywanie informacji w systemie jest bardzo tanie[6], a nasi klienci, widząc interesujący ich tytuł, może

[5] Niektórym to pytanie może wydaje się absurdalne – to bardzo dobrze. Niestety, w czasie ćwiczeń zadawano mi je kilka razy, więc na wszelki wypadek odpowiedzmy na nie.

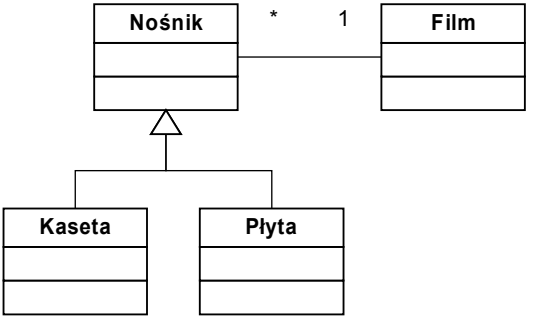
[6] We współczesnych systemach komputerowych przechowywanie informacji o 10 tys. czy 100 tys. filmów jest tak samo tanie.

poproszą o jego sprowadzenie. To też jest decyzją biznesową, którą należy uzgodnić ze zleceniodawcą.



2-38 Tworzenie diagramu klas dla wypożyczalni – krok 5

2-39 Tworzenie diagramu klas dla wypożyczalni – krok 6



2-40 Tworzenie diagramu klas dla wypożyczalni – krok 7

- 5 (str. 12) Wygląda na to, że wreszcie jakiś prosty fragment wymagań. Umieścimy odpowiednie atrybuty w klasie Film i problem rozwiązany (rysunek 2-41). Zwróćmy uwagę, że atrybut przechowujący koszt wypożyczenia filmu jest oznaczony jako klasowy (podkreślona nazwa) – jest to właściwa decyzja, ponieważ ma on być taki sam dla wszystkich filmów.

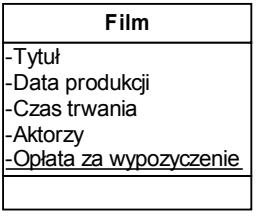
  Ale czy na pewno jest to właściwe podejście? Co może budzić wątpliwości? Myślę, że można lepiej przecho-

wywać informacje dotyczące aktorów. Zwykle będzie ich wielu, więc jak ich zapamiętamy w jednym atrybucie? Możemy wypisać ich imiona i nazwiska rozdzielone przecinkami. To „zadziała", ale czy nie mamy specjalnej konstrukcji dla atrybutów mających wiele wartości? Oczywiście, że mamy (poznaliśmy ją w podrozdziale 2.4.2.1 na stronie 28): atrybut powtarzalny (czyli właśnie taki, który może przechowywać wiele wartości). Rysunek 2-42 przedstawia zmodyfikowaną wersję klasy Film. Warto zwrócić uwagę, że zmieniliśmy nazwę atrybutu na Aktor (dlaczego? – patrz odpowiedni komentarz w podrozdziale 2.4.2.1).

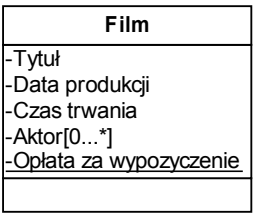
Czy to już wreszcie koniec z klasą Film? Obawiam się, że jeszcze nie (ale to już prawie ostatnie usprawnienie). Co możemy ulepszyć? Nadal zarządzanie informacjami o aktorach. Zauważmy, że aktorzy będą się powtarzali w wielu filmach. Czy w związku z tym będą się powtarzali też w wystąpieniach obiektu klasy Film? Niestety tak, ponieważ są to wartości atrybutu, które nie mogą być współdzielone. Jak temu zaradzić? Stworzymy nową klasę i połączymy ją asocjacją z klasą film – proste, prawda? Efekt naszych poczynań zawiera rysunek 2-43. Warto skomentować jeszcze liczności:

- Z filmem może być połączonych wielu aktorów (lub żaden, np. film przyrodniczy),

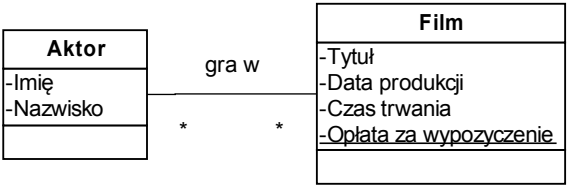
- Konkretny aktor może grać w wielu filmach lub w żadnym (sytuacja podobna jak w przypadku nośników i filmów).



2-41 Tworzenie diagramu klas dla wypożyczalni – krok 8



2-42 Tworzenie diagramu klas dla wypożyczalni – krok 9



2-43 Tworzenie diagramu klas dla wypożyczalni – krok 10

- 1…5 (str. 11) Myślę, że jest właściwy moment na obejrzenie tego, co do tej pory stworzyliśmy. Diagram podsumowujący jest pokazany na rysunku 2-44.

  No i jak to się prezentuje? Ładne? a może coś warto byłoby zmienić? Pomyślmy… Wygląda na to, że mamy takie same atrybuty w dwóch klasach: Aktor i Osoba. Taka sytuacja może być przyczynkiem do zastosowania dziedziczenia, ale nie musi, np. nie zrobimy dziedziczenia pomiędzy klasami monitor i dom nawet gdyby obie zawierały atrybut kolor. W przypadku naszego diagramu sytuacja jest inna: można założyć, że aktor jest szczególnym rodzajem osoby, więc takie dziedziczenie będzie poprawne. Odpowiedni fragment diagramu klas zawiera rysunek 2-45.

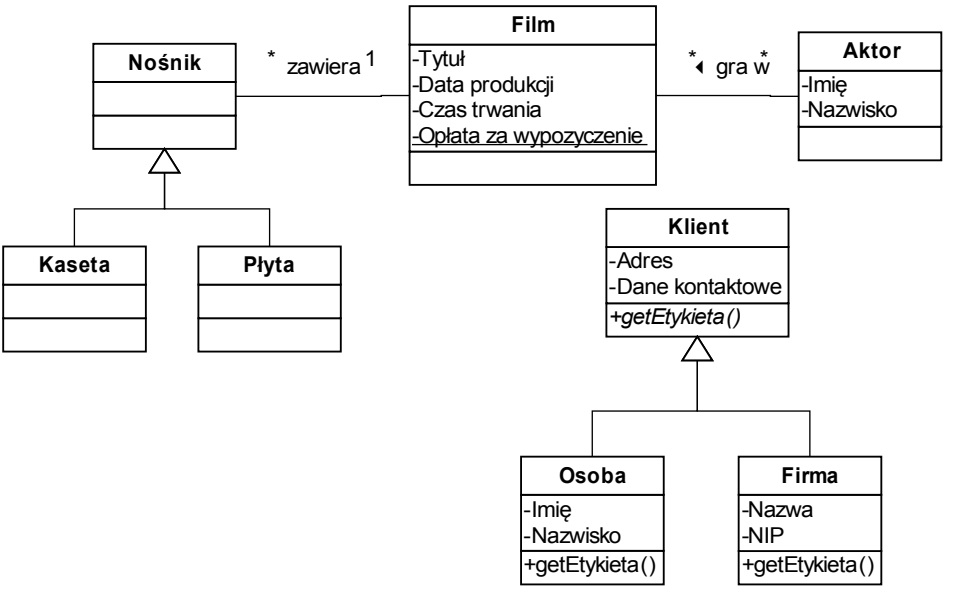
  Takie przedstawienie aktora powoduje pewien biznesowy problem. Jaki? (spróbuj sam to odkryć) Otóż, z naszego aktualnego diagramu (rysunek 2-45) wynika nie tylko, że aktor jest osobą (to dobrze), osoba jest klientem (też dobrze), ale również, że aktor jest naszym klientem (co może byłoby miłe, ale nie wynika z naszych wymagań

na system). Co z tym zrobimy? Mamy kilka możliwości:

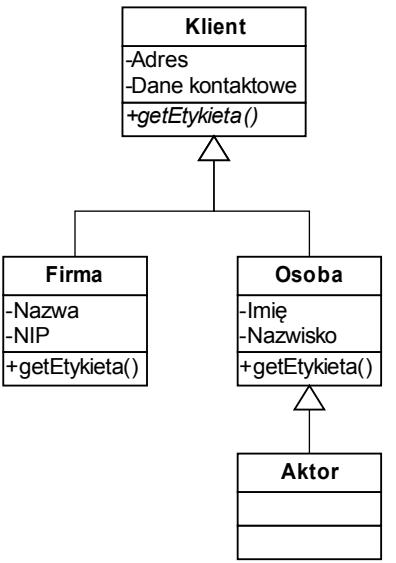
- Powrócimy do wersji diagramu z rysunku 2-44 (aktor i osoba nie są ze sobą połączone),
- Wrócimy do rozwiązania w którym klient, osoba, firma nie były połączone dziedziczeniem.
- Zmodyfikujemy nasz ostatni diagram tak, aby aktor nie był (pośrednio) klientem.

Myślę, że zdecydujemy się na ostatnią możliwość, która nadal umożliwia korzystanie z dobrodziejstw dziedziczenia. Rysunek 2-46 zawiera jedno z rozwiązań:

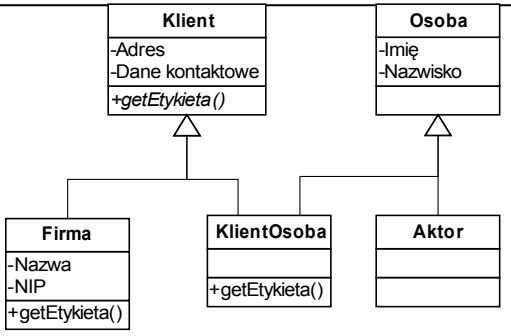
- Dodaliśmy klasę KlientOsoba, która dziedziczy i z Klienta, i z Osoby. Oznacza to, że mamy do czynienia z wielodziedziczeniem (ponieważ mamy więcej niż jedną nadklasę).
- Klasa Osoba nie dziedziczy z klasy Klient,
- Aktor nadal dziedziczy z osoby.



2-44 Tworzenie diagramu klas dla wypożyczalni – krok 11

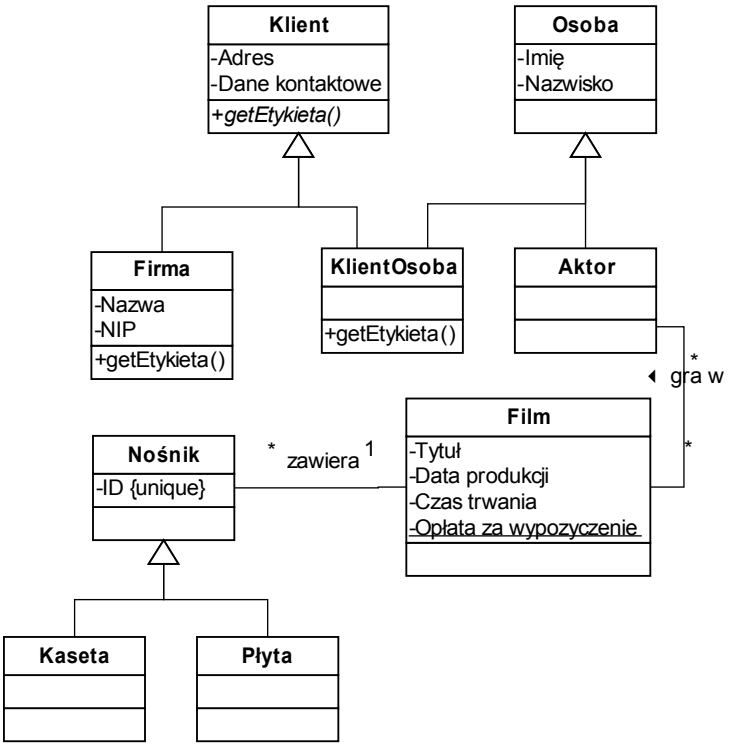


2-45 Tworzenie diagramu klas dla wypożyczalni – krok 12



2-46 Tworzenie diagramu klas dla wypożyczalni – krok 13

- 6 (str. 12) Wreszcie coś rzeczywiście łatwego. Dodajemy atrybut ID do klasy Nośnik oraz ograniczenie, które zapewni jego unikalność. Rysunek 2-47 zawiera nasz diagram z uwzględnieniem wszystkich dotychczasowych zmian.



2-47 Tworzenie diagramu klas dla wypożyczalni – krok 14

- 7 (str. 12) Na początek zastanówmy się, jakie w ogóle są możliwości przechowywania informacji na temat rodzaju/kategorii czegoś. Wydaje się, że mamy trzy podejścia:
    - Umieszczenie atrybutu w klasie, który jako tekst lub liczba będzie przechowywał taką informację (rysunek 2-48 (a)). Niewątpliwą zaletą takiego sposobu jest łatwość implementacji; poza tym ma chyba same wady:
        - Wyszukiwanie będzie utrudnione, ponieważ trzeba przejrzeć całą ekstensję, aby znaleźć obiekty z konkretną wartością atrybutu. Dodatkowo istnieje możliwość opisa-

nia tego samego (z biznesowego punktu widzenia) za pomocą różnych słów oraz popełnienia błędów (literówek w nazwach, co skutecznie uniemożliwi odszukanie).

- Nie mamy możliwości przechowywania specyficznych informacji dla danej kategorii w konkretnych wystąpieniach, np. siła ciągu konkretnego modelu samolotu odrzutowego.
- Wielokrotnie przechowujemy te same nazwy/informacje, np. nazwę kategorii.
- Utrudniona przynależność do kilku kategorii naraz.

- Stworzenie klasy opisującej rodzaj (np. właśnie o nazwie Rodzaj) i połączenie jej z opisywaną klasą (rysunek 2-48 (b)). W stosunku do poprzedniego sposobu:
  - mamy łatwość wyszukiwania (obiekt opisujący konkretny rodzaj jest połączony ze wszystkimi wystąpieniami tego rodzaju).
  - Informację o rodzaju przechowujemy tylko raz,
  - Łatwość zamodelowania przynależności do kilku rodzajów (ograniczona tylko licznościami asocjacji),
  - Łatwość dodawania nowych rodzajów w czasie działania systemu,
  - Brak prostego sposobu na przechowywanie wartości specyficznych dla konkretnego rodzaju (da się to zrobić, ale trzeba wprowa-

dzić klasę pośredniczącą – jak? To dobre ćwiczenie – można skorzystać z informacji zawartych w podrozdziale 2.4.3.1, strona 38 oraz 2.4.3.5, strona 43).

- Skorzystanie z dziedziczenia: poszczególne kategorie będą zapamiętane jako szczególne przypadki pewnej nadklasy, np. „samolot" – „samolot odrzutowy" (rysunek 2-48 (c)). Uwagi dotyczące tego podejścia:

    - Największą zaletą tego podejścia jest czytelność (na pierwszy rzut oka widzimy, jakie rodzaje są dopuszczalne) oraz możliwość przechowywania informacji specyficznych dla konkretnego rodzaju, np. siła ciągu dla samolotu odrzutowego (prosty atrybut w podklasie),

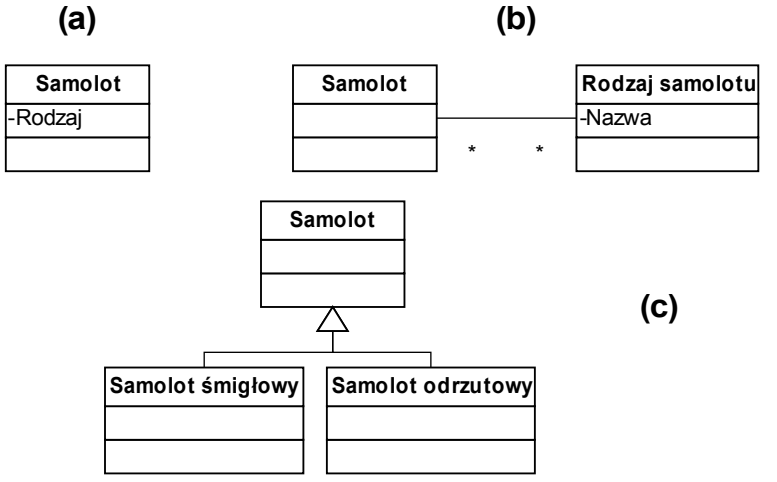
    - Wadą jest sposób dodawania nowych rodzajów. Ponieważ musimy dodać nową (pod)klasę, należy przerwać działanie systemu, dopisać odpowiedni kod, uruchomić, poprawić błędy, znowu uruchomić i tak dalej. Wada ta nie jest specjalnie dokuczliwa, jeżeli poszczególne rodzaje są znane w momencie tworzenia systemu i raczej nie będą się zmieniać.

    - Wyszukiwanie mamy ułatwione, ponieważ istnieje dedykowana ekstensja dla każdej klasy (w tym przypadku również rodzaju).

W takim razie, który z powyższych sposobów wybrać dla naszej wypożyczalni i informacji o kategoriach filmów? Ze względu na ostatnie zdanie tego punktu („dostosowany do przechowywania informacji specyficznych dla poszczególnych kategorii"), nie mamy wielkiego wyboru i zastosu-

jemy sposób z dziedziczeniem.

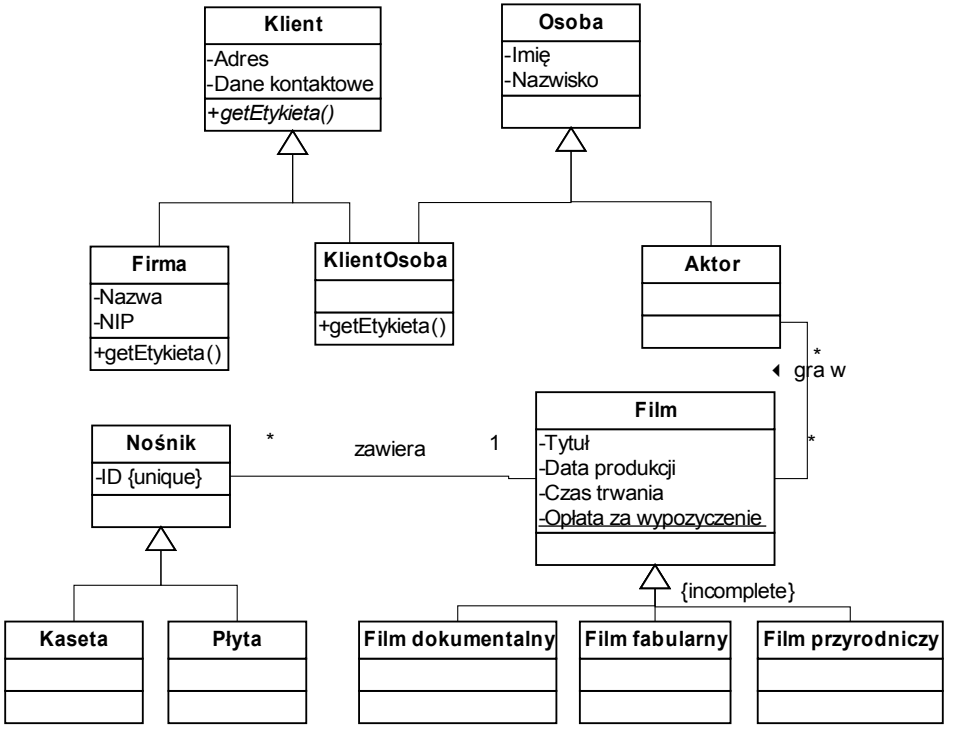
Rysunek 2-49 przedstawia kompletny (uwzględniający nasze dotychczasowe prace) diagram umożliwiający przechowywanie informacji o filmach różnych kategorii. Warto zwrócić uwagę na ograniczenie {incomplete}, wykorzystane przy dziedziczeniu z klasy film. Oznacza ono, że zdajemy sobie sprawę, że będzie więcej podklas (mówi o tym wyrażenie „np." występujące w wymaganiach), ale na razie ich nie znamy. Podobną konstrukcją jest elipsa (umieszczana jako „…" zamiast podklasy), która mówi, że znamy dodatkowe podklasy, ale z jakichś powodów (np. czytelności) ich nie umieszczono.



2-48 Ilustracja możliwości przechowywania informacji o rodzaju (np. samolotu)

- 8 (str. 12) Bogatsi o wiedzę pochodzącą z analizy poprzedniego punktu, a dotyczącą sposobów przechowywania informacji o kategoryzacji, łatwo znajdziemy właściwe rozwiązanie: dziedziczenie (ponieważ musimy przechowywać informacje specyficzne dla konkretnych filmów). Zanim jednak weźmiemy się do rysowania diagramu, przypomnijmy sobie rozwiązanie zastosowane w poprzednim punkcie. Tak, niestety, tam też było dziedziczenie (rysunek 2-49). Zatem

w jaki sposób dołożymy to nowe dziedziczenie (pochodzące z tego punktu) do już istniejącego? Sprawa wydaje się bardzo trudna, aż do momentu gdy przypomnimy sobie dziedziczenie wieloaspektowe (podrozdział 2.4.4.5, strona 55). Przecież jest to wręcz podręcznikowy przykład na jego zastosowanie![7] Spójrzmy, jak będzie wyglądał odpowiedni fragment diagramu (rysunek 2-50). Kilka uwag:



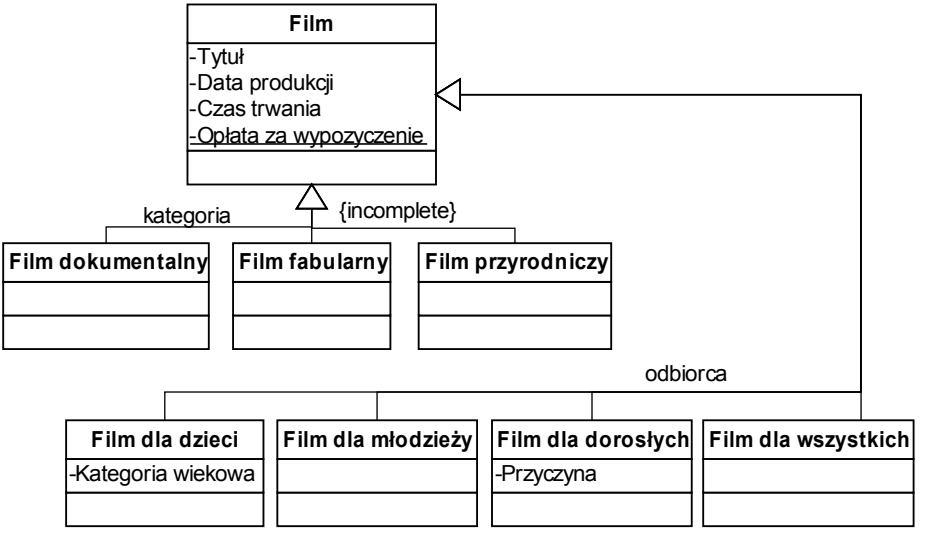
2-49 Tworzenie diagramu klas dla wypożyczalni – krok 15

- Ponieważ mamy do czynienia z dziedziczeniem wieloaspektowym, obowiązkowe jest umieszczenie nazw aspektów (kategoria, odbiorca),

[7] Zapewne to nie przypadek sprawił, że znalazł się w tej książce…

- We właściwych podklasach umieściliśmy atrybuty („Kategoria wiekowa”, „Przyczyna”).

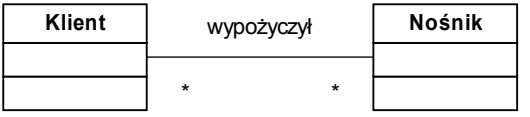
- 9, 10, 11 (str. 12) Pierwszym rozwiązaniem, które większości osób przychodzi do głowy, jest połączenie klasy klient z… no, właśnie z czym: z filmem czy nośnikiem? Załóżmy, że:
  - Z filmem. Wiemy, który film klient wypożyczył, ale czy wiemy, który nośnik (egzemplarz)? Ktoś może powiedzieć, że jest asocjacja łącząca film z nośnikiem (rysunek 2-49). To prawda, ale należy zwrócić uwagę, że informuje ona nas o wielu nośnikach połączonych z konkretnym filmem (liczność wiele „*”). Czyli raczej ten sposób nie jest dla nas najlepszy.



2-50 Tworzenie diagramu klas dla wypożyczalni – krok 16

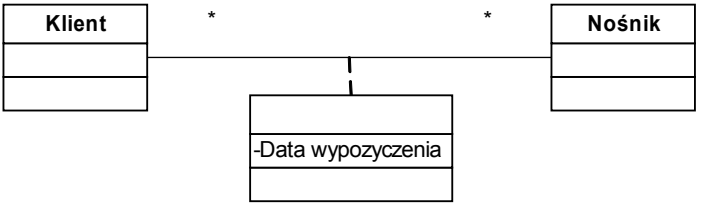
- Z nośnikiem. Czy tutaj nie będzie podobnego problemu? Patrzymy na diagram i widzimy, że nie: konkretny nośnik jest połączony z jednym obiektem klasy film.

A zatem, wiemy już, że asocjację należy utworzyć pomiędzy klientem a nośnikiem (rysunek 2-51). No tak, ale w wymaganiach jest powiedziane, że musimy pamiętać datę wypożyczenia – jak to osiągnąć? Najpierw spróbujmy czegoś prostego: może jakiś atrybut? Dodajmy atrybut data wypożyczenia w klasie klient. Teraz możemy zapamiętać datę wypożyczenia, ale wypożyczeń będzie wiele (oby, bo inaczej nasz biznes zbankrutuje). No, to zróbmy atrybut powtarzalny i przechowamy wiele wartości (dat). Ale która data będzie dotyczyła którego wypożyczenia? Tego nie uda nam się ustalić, ponieważ nie mamy gwarancji kolejności powiązań oraz wartości (raz pewna wartość/atrybut będzie otrzymana jako dziewiąta, a raz może być jako trzecia). Może w takim razie, umieścimy te atrybuty w klasie Nośnik? Tak też się nie uda – z tych samych powodów.



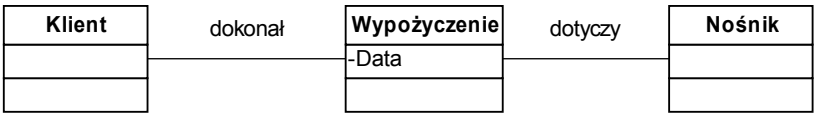
2-51 Tworzenie diagramu klas dla wypożyczalni – krok 17

Próbowaliśmy prostego podejścia (z atrybutem), ale nie spełniło naszych oczekiwań. Trzeba użyć czegoś bardziej wyrafinowanego. Co możemy zastosować? Jakieś pomysły? Oczywiście: atrybut asocjacji! Będzie przechowywał datę dla konkretnego wypożyczenia dokonanego przez konkretnego klienta (rysunek 2-52).



2-52 Tworzenie diagramu klas dla wypożyczalni – krok 18

Przeczytajmy teraz uważnie punkt 2.2 (strona 12) naszych wymagań. Czy da się to zapamiętać przy pomocy zaproponowanej konstrukcji? Chyba tak… I znowu nie. W wymaganiach jest jasno napisane, że jedno wypożyczenie może zawierać do trzech nośników. A nasze wypożyczenie? Tylko jeden nośnik – możemy, co prawda to jakoś obejść, zapamiętując w systemie trzy wypożyczenia zawierające po jednym nośniku. Ale my chcielibyśmy jedno wypożyczenie z trzema nośnikami. A, to nie jest to samo. Analogiczna sytuacja występuje w sklepie: jedne zakupy obejmują wiele towarów (z których każdy może być w dowolnej cenie oraz ilości). Nie chcemy tego „udawać" za pomocą wielu transakcji po jednym towarze.



2-53 Tworzenie diagramu klas dla wypożyczalni – krok 19

W takim razie, co możemy zrobić, skoro poprzednie rozwiązanie nie było dobre? Nie poddawać się i próbować dalej. A może dodamy nową klasę? Nazwijmy ją Wypożyczenie i połączmy z klasą Klient oraz Nośnik. Rysunek 2-53 zawiera diagram ilustrujący ten pomysł. Musimy tylko oszacować liczności (spróbuj to najpierw zrobić sam):

- Konkretny obiekt klasy Klient (opisujący jakiegoś Jana Kowalskiego) może być połączony z wieloma wypożyczeniami (albo z żadnym, gdy jeszcze nie zdążył nic wypożyczyć) – liczność wiele („*").

- Konkretne wypożyczenie dotyczy konkretnego klienta (abyśmy wiedzieli, kto ma nasze nośniki) – liczność „1" (nie dopuszczamy „0" – dlaczego?).

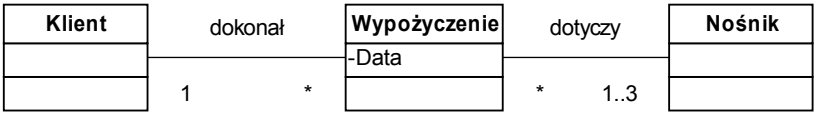
- Konkretny Nośnik może być powiązany z wieloma wypożyczeniami (bo chcemy pamiętać historię wypożyczeń) albo z żadnym, jeżeli nikt go jeszcze nie

wypożyczył – liczność wiele („*”).

- Konkretne wypożyczenie dotyczy 1, 2, 3 nośników (bo wypożyczenie bez nośników nie ma sensu oraz taka jest maksymalna, dozwolona wymaganiami liczba) – liczność „1..3”.

Rysunek 2-54 zawiera diagram już razem z licznościami. Warto zwrócić uwagę na nazwę drugiej asocjacji: „dotyczy”. Co w niej takiego niezwykłego? Jest to swoiste słowo-klucz: jeżeli nie wiemy, jak nazwać asocjację, to w większości przypadków możemy skorzystać właśnie z takiej nazwy. Naturalnie nie należy je nadużywać, bo diagram będzie nieczytelny.

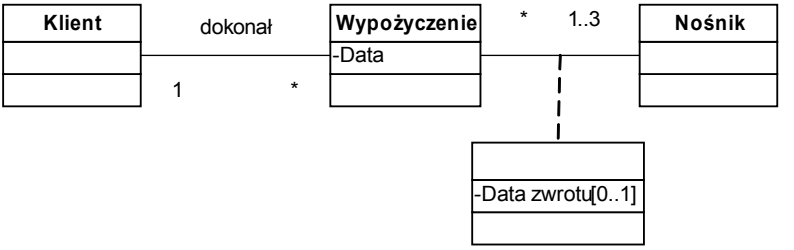
Czy wreszcie możemy uznać, że osiągnęliśmy nasz cel? Zwróćmy uwagę na fragment punktu nr 2.2 (strona 12): „Każdy z pobranych nośników może być oddany w innym terminie”. Czy nas system jest w stanie przechować takie informacje? Niestety, jak na razie, to nie.



2-54 Tworzenie diagramu klas dla wypożyczalni – krok 20

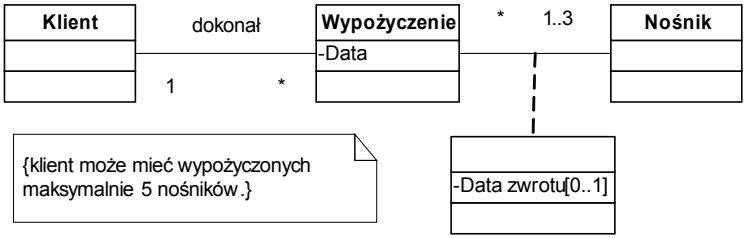
Spójrzmy na diagram z rysunku 2-54 i pomyślmy, w jaki sposób zapisać informację o dacie zwrotu konkretnego nośnika? Można założyć, że te dane dodatkowo opisują asocjację pomiędzy klasą Wypożyczenie oraz Zwrot. Teraz już chyba sprawa jest jasna: atrybut asocjacji (jednak go wykorzystamy!). Musimy jeszcze tylko wziąć pod uwagę pewien mały szczegół: załóżmy, że tworzymy instancję klasy Wypożyczenie i łączymy ją z wystąpieniem klasy Nośnik. W tym momencie tworzona jest też instancja atrybutu pamiętającego datę zwrotu – co tam wpiszemy? Przecież nie znamy daty zwrotu. Oczywiście możemy wpisać tam dowolną wartość i potem (przy przyjęciu nośnika) ją uaktualnić, ale to jest stosowanie sztuczek. Staramy się unikać ta-

kiego podejścia. Co zrobimy? Mamy odpowiednią konstrukcję: atrybut opcjonalny będzie tutaj idealny. Poprawioną wersję diagramu możemy obejrzeć na rysunku 2-55.



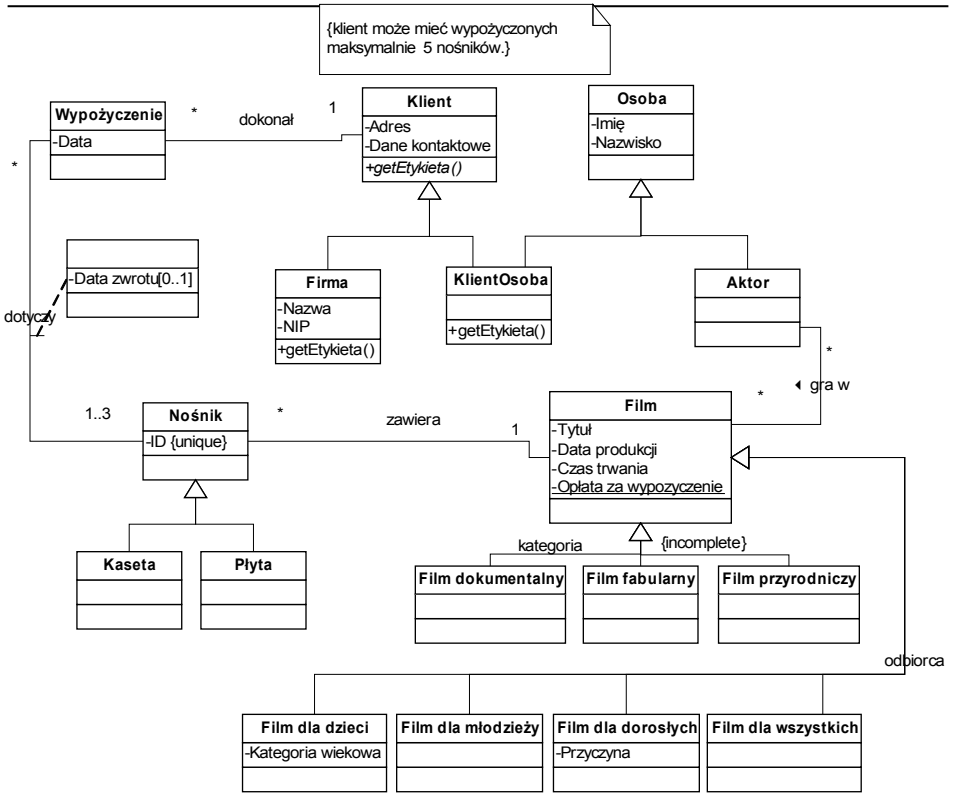
2-55 Tworzenie diagramu klas dla wypożyczalni – krok 21

Teraz pozostała nam już naprawdę tylko jedna kwestia związana z wypożyczeniami – pkt 2.2 (strona 12). W jaki sposób to uwzględnimy? Czy dodamy/zmodyfikujemy jakieś liczności? Spróbujmy się zastanowić, z czym związane jest to wymaganie (ale spróbuj najpierw sam). Tak naprawdę zostanie to uwzględnione w operacji wypożyczania nośnika. Innymi słowy, w kodzie metody. Jak wiemy, diagram klas pokazuje informacje, jakie metody występują, ale nie widzimy ich kodu w języku programowania. Czy to znaczy, że nic nie możemy zrobić i mamy zignorować ten punkt wymagań? Nie – użyjemy ograniczenia. Ogólnie rzecz biorąc, ograniczenie pozwala nanieść nam na diagram dowolne informacje m.in. W języku naturalnym. Umieszczamy je w nawiasach klamrowych (lub w specjalnej notatce – patrz dalej) w pobliżu elementu, którego dotyczą. No właśnie dotyczą – czego będzie dotyczyć nasze ograniczenie? W zależności od sytuacji biznesowej może w ogóle nie pozwolić na wypożyczenie nośnika (wtedy dotyczy stworzenia obiektu klasy Wypożyczenie) lub nie zezwoli na dodanie kolejnego nośnika do wypożyczenia (wtedy dotyczy stworzenia powiązania). Jak widać, mamy dwa kandydujące miejsca, gdzie możemy umieścić nasze ograniczenie (trudny wybór). Dlatego zlokalizujemy je w neutralnym miejscu – przy kliencie. Finalne rozwiązanie przedstawia diagram 2-56.

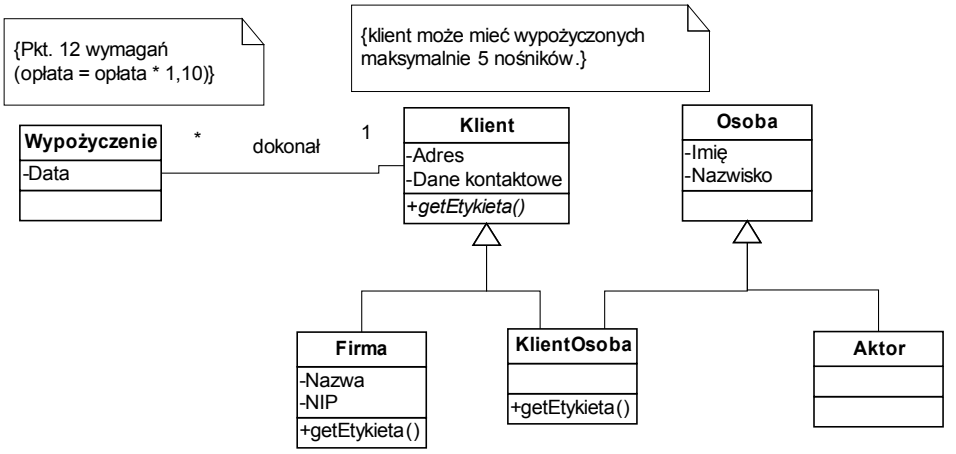


2-56 Tworzenie diagramu klas dla wypożyczalni – krok 22

Jeżeli dobrnąłeś do tego miejsca, to mam dobrą wiadomość: nie jest to, co prawda, koniec rysowania naszego diagramu klas dla wypożyczalni, ale najtrudniejsze elementy mamy już za sobą. Dlatego warto obejrzeć diagram podsumowujący nasze dotychczasowe rozwiązanie (rysunek 2-57).

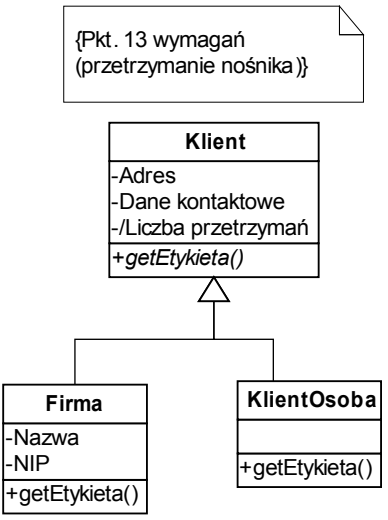


2-57 Tworzenie diagramu klas dla wypożyczalni – krok 23



2-58 Tworzenie diagramu klas dla wypożyczalni – krok 24

- 12 (str. 12) Co wynika dla naszego diagramu z tego punktu wymagań? Zastanówmy się, o czym tak naprawdę jest tam mowa. Otóż opisano pewną procedurę, którą należy zastosować w momencie, gdy klient spóźni się z oddaniem nośnika. Czy wymienione są tam jakieś informacje (dane), które trzeba będzie przechowywać? Nie. A zatem, jak już wiemy z naszych poprzednich rozważań, efektem tego punktu wymagań będzie po prostu ograniczenie. W tym przypadku będzie dotyczyło klasy wypożyczenie (a dokładniej jednej z jej metod). Odpowiedni fragment diagramu pokazany jest na rysunku 2-58.



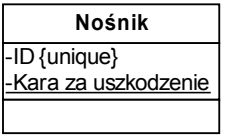
2-59 Tworzenie diagramu klas dla wypożyczalni – krok 25

- 13 (str. 13) Zastanówmy się, czy aktualna wersja diagramu klas jest w stanie przechowywać wszystkie informacje niezbędne do sprawdzenia danych potrzebnych w tym punkcie. Mamy informacje o wypożyczeniach danego klienta, możemy sprawdzić każde z nich i stwierdzić, czy zawiera spóźnione zwroty. Spóźnienia zliczamy i wiemy, czy klient przekroczył graniczną wartość trzech przetrzymań. W związku z tym, wydaje się, że nie trzeba modyfikować diagramu (oczywiście oprócz dodania odpowiedniego ograniczenia zawierającego ten warunek). Czy na pewno? Rozważmy procedurę zliczania spóźnień (opisaną powyżej): wygląda na to, że w przypadku aktywnych klientów (takich, którzy mają dużo wypożyczeń i zwrotów) obliczenie konkretnej wartości może zająć sporo czasu. Czy możemy coś na to poradzić? Pewnie już się domyśliłeś, drogi Czytelniku, że tak. Spróbuj się zastanowić i podać właściwe rozwiązanie, zanim o nim przeczytasz w następnym zdaniu. W tego typu sytuacjach idealnym rozwiązaniem jest atrybut wyliczalny (czyli taki, który przechowuje swoistą kopię danych – patrz podrozdział 2.4.2.1, strona 28). W jaki sposób go wykorzy-

stamy? Nazwijmy go „/Liczba przetrzymań”. System (a właściwe specjalna metoda) w momencie zwrotu nośnika sprawdzi, czy nie nastąpiło spóźnienie; jeżeli tak, to po prostu zwiększy jego wartość. Opóźnienie działania systemu będzie niezauważalne. Rysunek 2-59 zawiera zmodyfikowaną wersję klasy Klient razem z ograniczeniem.

- 14 (str. 13) W sprawdzaniu stanu technicznego nośnika nasz system nam raczej nie pomoże (ale może o tym przypomnieć odpowiednim komunikatem wyświetlonym na ekranie). Z tego powodu, pracownik wypożyczalni musi dokonać takiej oceny samodzielnie. Kwestią otwartą jest, skąd będzie wiedział, ile pieniędzy ma zapłacić klient za zniszczony nośnik? Można wyobrazić sobie kilka scenariuszy:

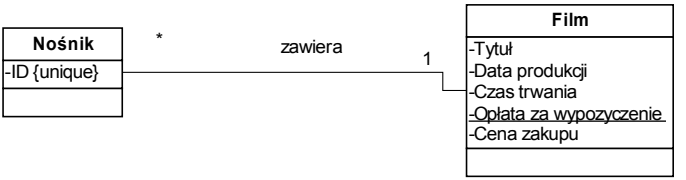
    - Zniszczenie dowolnego nośnika kosztuje tyle samo, np. 100 PLN. W takiej sytuacji informacja o kwocie może być przechowana w klasie nośnik jako, no właśnie jak? Czy już wiesz? Naturalnie jako atrybut klasowy (ponieważ jest taka sama dla wszystkich nośników – obiektów klasy nośnik). Ilustruje to rysunek 2-60 (atrybut klasowy „Kara za uszkodzenie”).



2-60 Tworzenie diagramu klas dla wypożyczalni – krok 26

    - Opłata za zniszczenie nośnika jest uzależniona od filmu. W takiej sytuacji, zamiast przechowywać ją w klasie Nośnik (gdzie by się powtarzała, bo jest wiele nośników z tym samym filmem), zapamiętamy ją w klasie film. System rejestrując informację o zwrocie uszkodzonego nośnika, sprawdza jego cenę w powiązanym z nim filmie (atrybut „Cena za-

kupu” na rysunku 2-61).



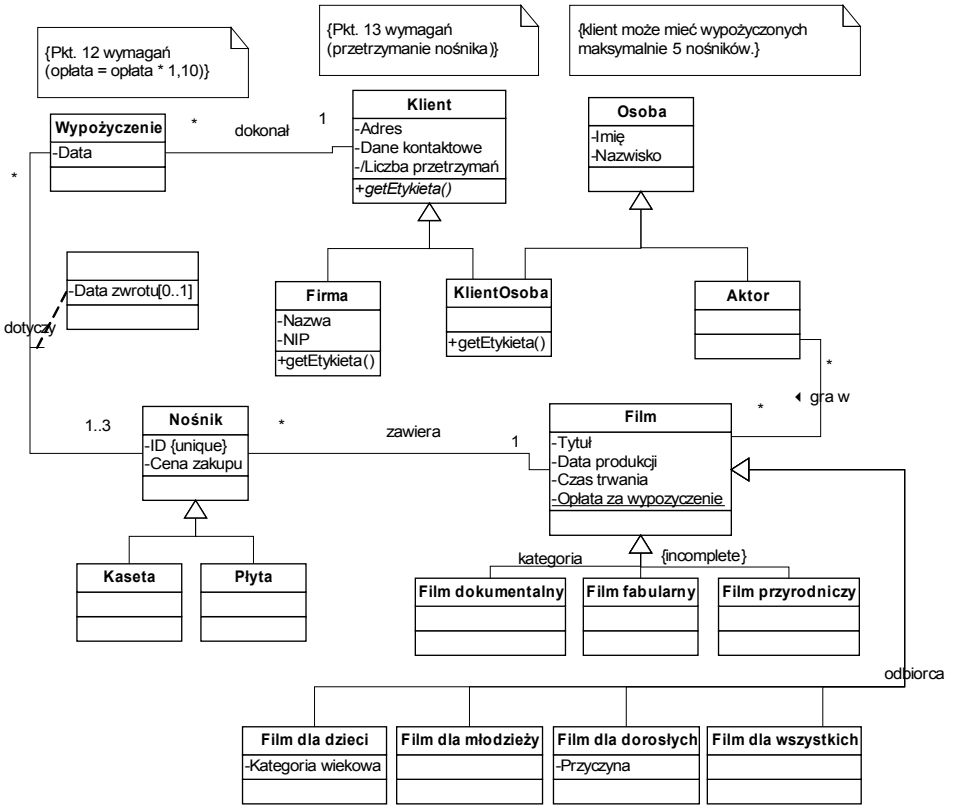
2-61 Tworzenie diagramu klas dla wypożyczalni – krok 27

- Powyższy sposób jest dobry, ale co wtedy, gdy nośniki z tym samym filmem były kupowane w różnych cenach (np. z powodu okresowych promocji)? w takiej sytuacji umieścimy atrybut „Cena zakupu” w klasie Nośnik. To podejście jest najbardziej elastyczne (bo możemy różnicować ceny dla poszczególnych egzemplarzy nośników), ale zajmuje najwięcej pamięci – coś za coś. Mimo wszystko właśnie to podejście zastosujemy.

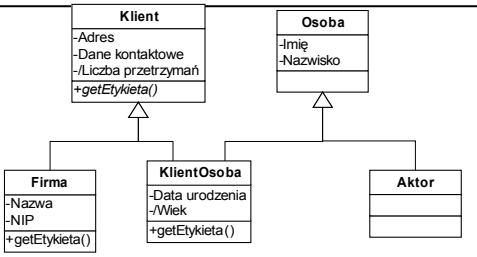
Rysunek 2-62 zawiera kompletny diagram uwzględniający ostatnie zmiany.

- 15 (str. 13) Rezultatem tego punktu będzie na pewno zastosowanie specjalnej procedury służącej do wypożyczania niektórych filmów. Czy tylko to? Nie, ponieważ, jak dotąd, nie mamy informacji o wieku naszych klientów. Czyli dodajemy atrybut wiek do klasy KlientOsoba. Czy na pewno? (spróbuj najpierw sam odpowiedzieć na to pytanie). Załóżmy, że zapisał się dzisiaj do naszej wypożyczalni hipotetyczny Jan Kowalski, który ma 22 lata. Ile będzie miał za rok? Oczywiście 23. Ale skąd system ma wiedzieć, kiedy dokładnie to się stanie? a jak już będzie wiedział, to ma codziennie przeglądać, czy któryś z klientów nie ma urodzin i zwiększać ich wiek? Trochę to absurdalne. Pewnie już się domyślasz, że należy pamiętać datę urodzenia, a wiek obliczać na podstawie aktualnej daty systemowej. Czyli dodajemy atrybuty „Data urodzenia” oraz „/Wiek” (jako wyliczalny!) do klasy KlientOsoba (rysunek 2-63).

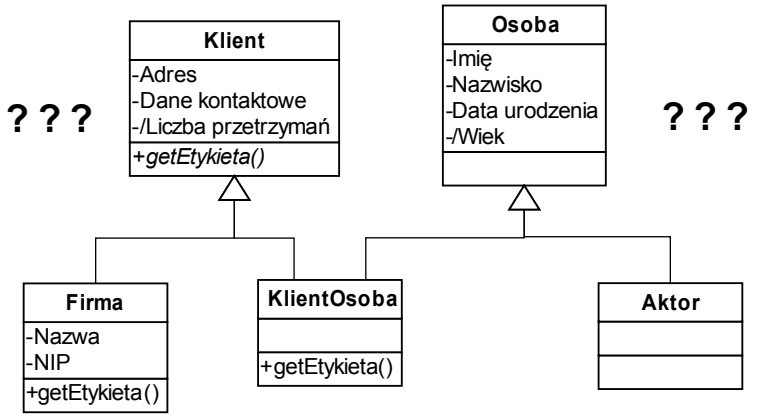
Chwileczkę, spójrzmy jeszcze raz na diagram z rysunku 2-63. Czy ta data urodzenia i wiek nie powinny być w klasie Osoba? Przecież są to cechy jak najbardziej „osobowe”! Zastanówmy się nad rysunkiem 2-64, który uwzględnia tę modyfikację. No tak, ale takie podejście oznacza, że informacje o wieku będziemy też przechowywali dla aktorów (a o tym nie ma mowy w naszych wymaganiach). Reasumując: zostajemy przy rozwiązaniu przedstawionym na rysunku 2-63 (atrybuty w klasie KlientOsoba).



2-62 Tworzenie diagramu klas dla wypożyczalni – krok 28



2-63 Tworzenie diagramu klas dla wypożyczalni – krok 29



2-64 Tworzenie diagramu klas dla wypożyczalni – krok 30

- 16 (str. 1313) Jaki wpływ ma taki zapis na diagram klas? Po pierwsze musimy się upewnić, czy takie informacje da się odzyskać z systemu. Sprawdźmy:

  - Czy w systemie znajdują się kompletne (opisane w wymaganiach) informacje dotyczące filmu? Tak – znamy jego tytuł, datę produkcji, czas trwania oraz, pośrednio (korzystając z asocjacji), grających aktorów.

  - Czy mamy wszystkie niezbędne informacje dotyczące wypożyczeń konkretnego klienta? Jak najbardziej – instancja klasy Wypożyczenie połączona z klasami Klient oraz Nośnik.

Reasumując: mamy odpowiednie informacje. Musimy tylko zadbać o właściwe przypadki użycia oraz implementację funkcjonalności, która zapewni nam do nich dostęp.

- 17, 18, 19 (str. 13)

Jak potraktujemy wymagania dotyczące raportów? Trzeba sprawdzić, czy wszystkie informacje, które mają się znaleźć w raporcie, są dostępne w systemie. Po sprawdzeniu (robiliśmy to przy okazji poprzednich punktów) dochodzimy do wniosku, że system jest w stanie dostarczyć nam potrzebne dane. A zatem, czy to koniec? Niezupełnie. Musimy przedyskutować kwestię podejścia do przechowywania danych.

Jak już ustaliliśmy, wszystkie dane do sporządzania raportów, mamy już w systemie. Możemy powiedzieć, że powtórne ich przechowywanie (w postaci raportów) byłoby redundancją danych. Jednocześnie pamiętamy, że redundancja danych w pewnych zastosowaniach ma sens. Zastanówmy się czy jest tak i w tym przypadku.

Gdybyśmy nie chcieli powtórnie przechowywać danych w postaci raportów, to możemy stworzyć funkcjonalność (metody), która w razie potrzeby wygeneruje raport na podstawie aktualnych danych. Taki raport zostanie wyświetlony czy wydrukowany, a następnie „zniknie" z systemu. Jeżeli ktoś po jakimś czasie będzie chciał otrzymać raport np. sprzed roku, to system ponownie go wygeneruje na podstawie aktualnych danych. Na pierwszy rzut oka wszystko wygląda dobrze. Czy zatem jest tu gdzieś jakiś „haczyk"? Niestety tak. Zwróćmy uwagę na sformułowanie „aktualnych danych". Siłą rzeczy będą to dane aktualne na teraz, nawet jeżeli dotyczą czasu sprzed roku. W czym tkwi problem? Otóż, może się tak zdarzyć, że np. omyłkowo pobierzemy pieniądze od klienta za wypożyczenie nośników, których on tak naprawdę nie brał. System to zanotuje, a następnie wykaże w raporcie dziennym, a później miesięcznym. Po jakimś czasie, klient lub my sami, zorientujemy się, że zaszła pomyłka. Nie ma problemu: naprawimy ją – zmodyfikujemy dane w systemie. Jeszcze później ktoś wygeneruje raport za ten sam okres czasu, gdy wystąpiła pomyłka. Ale ponieważ naprawiliśmy pomyłkę (i zmodyfiko-

waliśmy dane w systemie), ten „nowy" raport będzie inny (bo nie będzie uwzględniał błędnych danych). Myślę, że teraz widać, na czym polega problem z generowaniem raportów.

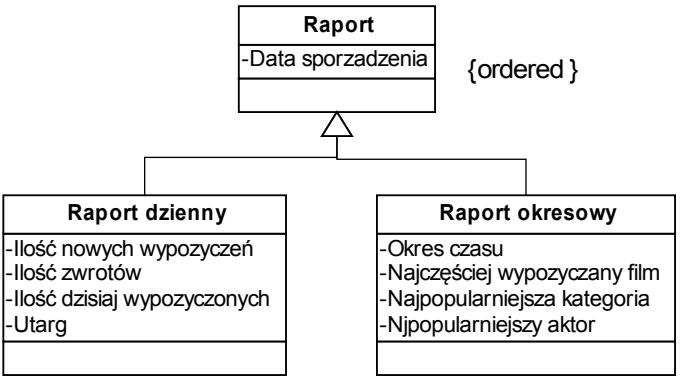
Jeżeli chcielibyśmy uwzględnić powyższe uwagi, to należałoby fizycznie zapisać zawartość raportu w momencie jego generowania. Jak to zrobić? Mamy dwa główne sposoby:

- Zapamiętać wygenerowany plik (np. PDF) lub całkowity tekst raportu,

- Zapamiętać kopię danych znajdujących się w raporcie (np. jako stringi), ale bez tekstu, który je otacza. Dzięki temu całość zajmie mniej miejsca, ale co ważniejsze, dane będą miały pewną strukturę, co ułatwi np. wyszukiwanie.

Biorąc wszystkie za i przeciw, zdecydujemy się na rozwiązanie drugie (zapamiętywanie wygenerowanego raportu) i do tego w wariancie drugim (jako dane częściowo ustrukturalizowane). Jak to zrobimy? Stworzymy odpowiednie klasy razem z dedykowanymi atrybutami. Jakie będą te atrybuty? Raczej wyliczalne (chociaż to jest trochę dyskusyjne, ale nie będziemy się w to już zagłębiać).

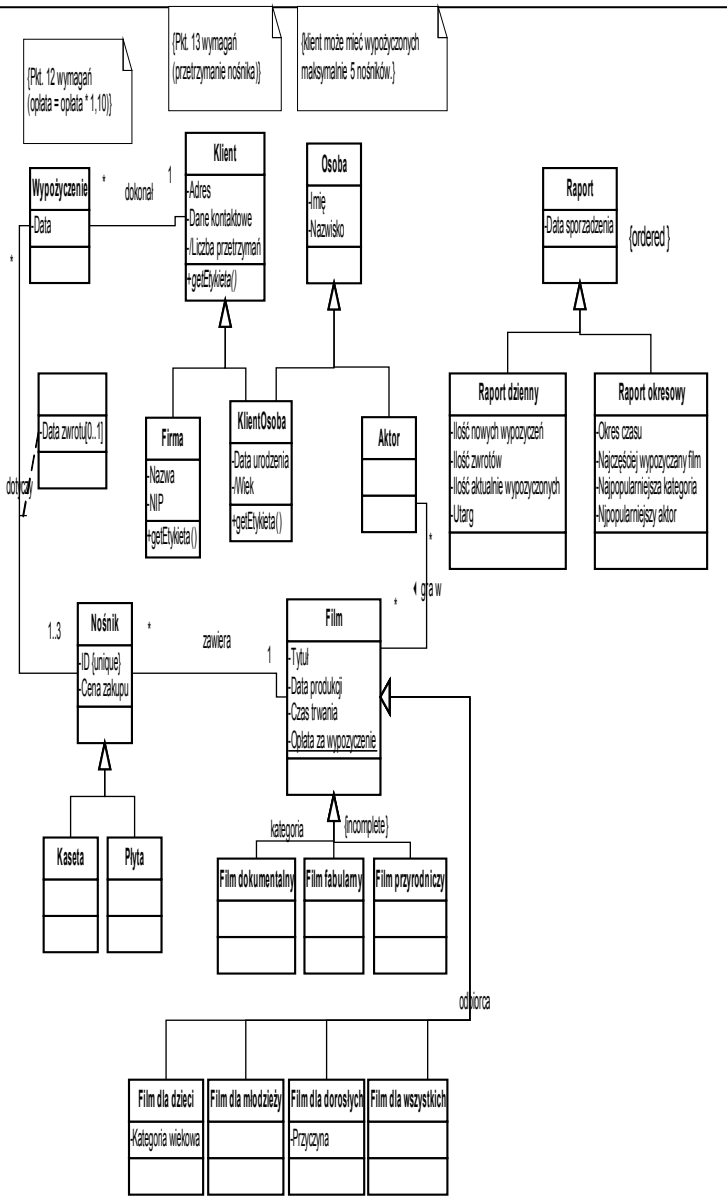
Warto jeszcze zwrócić uwagę na punkt 2.2 (strona 13) wymagań. Aby móc coś uporządkować chronologicznie, trzeba mieć daty. Oznacza to, że obydwa rodzaje raportów mają wspólny atrybut: data sporządzenia. A to (mam nadzieję, że wszyscy to zauważą) w tym przypadku uzasadnia powołanie wspólnej nadklasy Raport, z której będą dziedziczyć bardziej szczegółowe raporty. Rysunek 2-65 ilustruje nasz pomysł.



2-65 Tworzenie diagramu klas dla wypożyczalni – krok 31

Rysunek 2-66 zawiera podsumowanie naszych dotychczasowych osiągnięć. Niektórym może się wydać dziwne, że klasy raportów nie są niczym połączone z resztą elementów i tworzą jakby samotną „wysepkę" na diagramie. Czy nie powinniśmy mieć jakichś asocjacji umożliwiających powiązanie danych ze sobą? Czy bez nich będziemy w stanie dotrzeć do odpowiednich danych? (spróbuj najpierw na to odpowiedzieć sam, drogi czytelniku). Z czym mielibyśmy to połączyć? Ze wszystkimi danymi potrzebnymi do wygenerowania raportów? Sporo by tego było. Otóż, do odczytania danych, bez „przechodzenia" po asocjacjach (powiązaniach), wykorzystamy… metody klasowe. Wykonają one za nas „czarną robotę" i znajdą najpopularniejszego aktora czy kategorię. Przypomnijmy, że metody klasowe wywoływane są na rzecz klasy (np. Aktor) i mają dostęp do wszystkich jej wystąpień (np. Aktorów). Dzięki temu będą np. w stanie policzyć, w ilu wypożyczonych filmach grali poszczególni aktorzy.

Miło mi jest zakomunikować, że wreszcie dobrnęliśmy do końca tworzenia diagramu klas dla wypożyczalni wideo. Jak na razie brakuje na nim metod, ale uzupełnimy je podczas dalszych prac analityczno-projektowych.

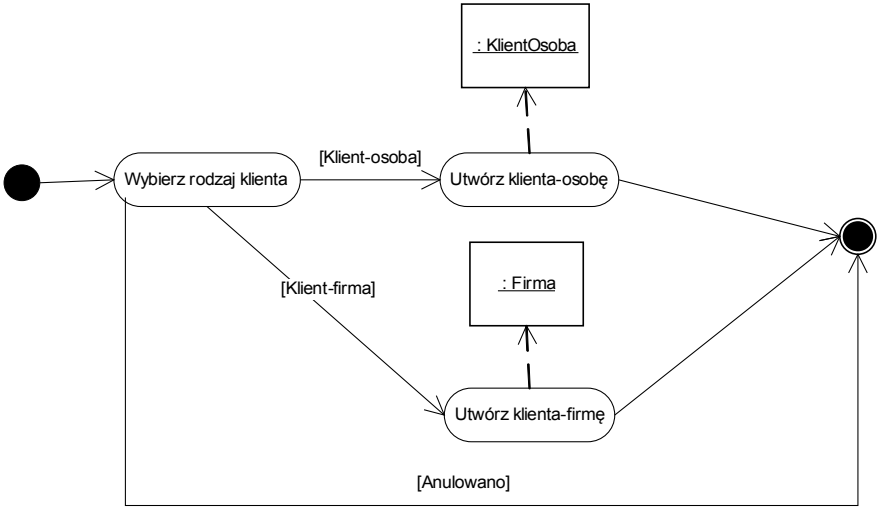


2-66 Diagram klas dla wypożyczalni wideo (bez metod)

2.5 Diagram aktywności

Diagramy aktywności służą do opisu działania systemu. Można o nich myśleć jako o sposobie zapisu reguł biznesowych w projektowanym systemie. W większości przypadków posłużą nam do stosunkowo precyzyjnego zapisania algorytmów dla bardziej skomplikowanych fragmentów aplikacji. Wspominając o algorytmach, mamy na myśli raczej dość wysoki poziom abstrakcji, a nie konkretne problemy informatyczne. Innymi słowy, opiszemy raczej sposób dodawania nowego klienta wypożyczalni, a nie metodę sortowania danych. Więcej informacji na temat diagramów aktywności można znaleźć np. w [Fowl04] lub [Płod05]).

I taki właśnie przykład (dodawanie nowego klienta) został przedstawiony na rysunku 2-67. Zwróćmy uwagę, że diagram pokazuje również utworzenie nowych obiektów opisujących wybrany rodzaj klienta.



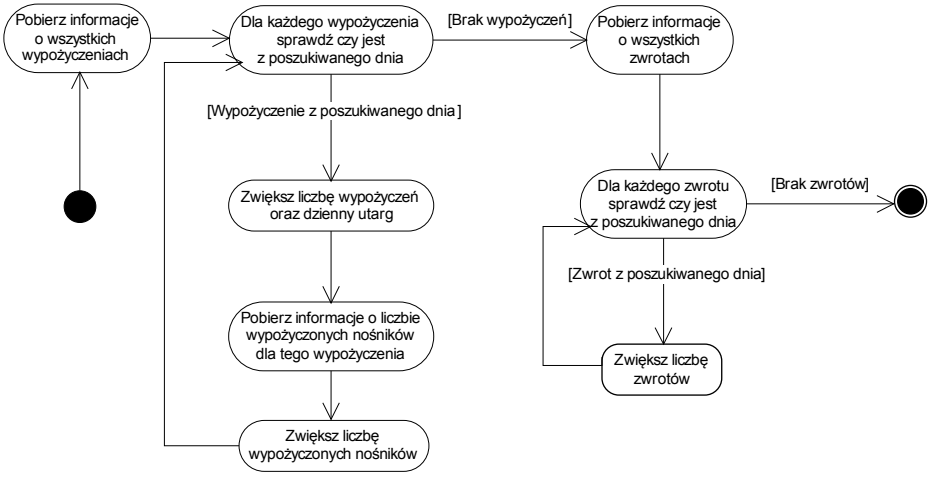
2-67 Diagram aktywności przedstawiający dodawanie nowego klienta

Inny diagram aktywności jest pokazany na rysunku 2-68. Precyzuje on sposób generowania raportu dziennego zawierającego informacje o:

- dziennej liczbie wypożyczeń,
- utargu,

- liczbie wypożyczonych oraz zwróconych nośników.

Diagramy aktywności powinny być wykonane dla większości funkcji systemu. Dzięki temu projektant oraz programista będą dokładniej wiedzieli, jaki ma być efekt działania danej funkcjonalności.



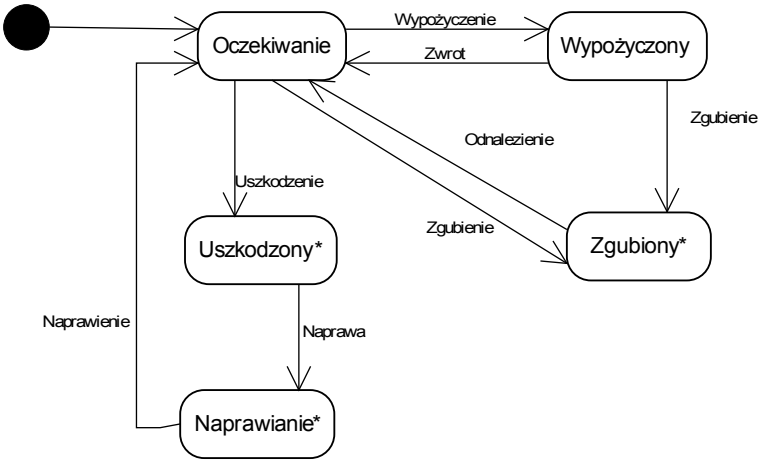
2-68 Diagram aktywności przedstawiający generowanie raportu dziennego

2.6 Diagram stanów

Diagramy stanów mogą być tworzone dla całego systemu lub konkretnych klas. Ze względu na zbyt wysoki poziom ogólności, pierwsze podejście jest dość rzadko stosowane. Na stan obiektu składa się stan jego wszystkich składowych, czyli atrybutów oraz powiązań. W związku z tym liczba potencjalnych, różnych stanów jest bardzo duża. Dlatego, analizę stanów instancji pojedynczej klasy przeprowadzamy w oparciu o pewne kategorie stanów, np. dla nośnika zauważymy przejście od stanu wypożyczony do np. zwrócony, ale raczej nie będziemy analizowali wszystkich możliwych stanów określonych przez jego różne numery identyfikacyjne. Więcej informacji na temat diagramów stanów można znaleźć np. w [Fowl04] lub [Płod05]).

Diagram stanów tworzymy dla jakiejś klasy czy klas. Zwykle są to kluczowe, biznesowe dane w naszym systemie. Mając taki diagram, musimy się upewnić, że aplikacja pozwala na przejścia pomiędzy stanami. Innymi słowy, czy są odpowiednie funkcje w systemie dostępne przez dedykowane GUI.

W przypadku naszej wypożyczalni wideo taką analizę warto przeprowadzić dla klasy nośnik. Przykładowy diagram jest pokazany na rysunku 2-69.



* - te stany nie są uwzględniane przy implementacji aktualnej wersji . Być może zostaną oprogramowane w przyszłości.

2-69 Przykładowy diagram stanów dla klasy Nośnik

Tak naprawdę to część stanów nie będzie wykorzystywana w aktualnej wersji systemu (oznaczono je *). Mimo wszystko warto je umieścić (oczywiście z odpowiednią adnotacją). Dzięki temu będziemy mieli mniej pracy w przyszłości. Warto również zwrócić uwagę, że ten diagram mógłby być bardziej skomplikowany, np. uszkodzenie nośnika mogłoby następować również w czasie wypożyczenia, a naprawienie nie zawsze musi kończyć się sukcesem. Tego typu zasady są zależne od uwarunkowań biznesowych i powinny być uzgodnione ze zleceniodawcą.

3 Projektowanie

Projektowanie jest jedną z kolejnych faz wytwarzania oprogramowania. O ile w fazie analizy odpowiadamy na pytanie „co?" ma zostać zaimplementowane, o tyle w fazie projektowania staramy się już określić „jak?" to ma być zrobione. Innymi słowy, faza ta określa:

- Technologię (jedną lub więcej), której będziemy używać,
- Jak system ma działać. Dokładność tego opisu może być różna. Idealne podejście zakłada, że programista, w oparciu o projekt systemu, w następnej fazie wytwarzania oprogramowania (implementacja), dokładnie wie, jak ma go stworzyć. Im mniejsze ma pole manewru, tym lepiej. Zakłada się również, że w przypadku niejasności nie podejmuje własnych decyzji (np. dotyczących zakładanej funkcjonalności), ale zwraca się do osób odpowiedzialnych za przygotowanie poprzednich faz.

W klasycznym modelu kaskadowym faza projektowania występuje po etapie analizy. W praktyce wiele firm/programistów traktuje ją trochę „po macoszemu". Często jest tak, że programista zaczyna implementację, nie tylko nie mając projektu systemu, ale nawet nie znając dokładnych wymagań na system (pochodzących z fazy analizy). Takie podejście nie jest najlepszym sposobem na stosunkowo szybkie wytworzenie w miarę bezbłędnego oprogramowania spełniającego wymagania klienta. I to wszystko za rozsądne pieniądze. Po raz kolejny powtórzę: błędy popełniane we wstępnych fazach są najbardziej kosztowne:

- Wyobraźmy sobie, że gdy dostarczyliśmy wytworzony system do klienta, nie ma on zaimplementowanych pewnych funkcjonalności – gdzie został popełniony błąd? Naturalnie w fazie analizy. Konsekwencje jego mogą być bardzo poważne, ponieważ może się okazać, iż przy obecnych założeniach projektu dodanie owych funkcjonalności może być nawet niemożliwe!
- W fazie projektowania podjęto określone decyzje, np. dotyczące sposobu przechowywania informacji o pewnych danych. Jeżeli te decyzje były błędne, to może się okazać, że pewnych danych nie da się w ogóle zapamiętać.

- Wbrew pozorom błędy z fazy implementacji są stosunkowo niegroźne i dość łatwe do naprawienia. Pod warunkiem, że jesteśmy w stanie odtworzyć sytuację prowadzącą do błędu (co nie zawsze się udaje).

Cały ten wywód ma na celu przekonanie Czytelników, że prawidłowo przeprowadzona faza projektowania jest nie tylko potrzebna, ale się po prostu opłaca.

Jakie są „narzędzia" projektanta? Podobnie jak w fazie analizy będziemy wykorzystywać diagramy UML. Ktoś mógłby zapytać: „w takim razie po co rysować dwa razy to samo?" Otóż, nawet gdy korzystamy z takich samych instrumentów, można to robić w inny sposób. Jak pokażemy, diagram klas z fazy analizy różni się (czasami mniej, a czasami bardzo mocno) od diagramu klas z fazy projektowania. Jak wspomnieliśmy, w fazie projektowania podejmujemy decyzję co do technologii, której będziemy używać. Obejmuje to też język programowania. Wynika z tego, że już na diagramie klas musimy uwzględnić konkretny język programowania, np. w zakresie konstrukcji, których nie obsługuje (a które były na diagramie klas z fazy analizy).

Następne podrozdziały będą poświęcone implementacji poszczególnych konstrukcji występujących na diagramie klas i będą się odnosiły do odpowiadających im podrozdziałów z podrozdziału 2.4 (strona 24 i dalsze). Oczywiście nie należy przedstawionych rozwiązań traktować jako prawd objawionych, ale raczej jako pewien wzorzec oraz sugestię ułatwiającą samodzielne podjęcie decyzji projektowych.

Jakkolwiek rozdział ten jest poświęcony projektowaniu, a nie implementacji, to zdecydowałem się umieścić w nim przykładowe sposoby implementacji poszczególnych konstrukcji (takich jak asocjacje, dziedziczenie itp.). Mam nadzieję, że dzięki temu całość będzie bardziej zrozumiała.

Zachęcam również do przysyłania mi własnych pomysłów oraz rozwiązań.

3.1 Klasy

3.1.1 Obiekt

Zanim zajmiemy się klasami, wróćmy na chwilę do pojęcia obiektu. Co mówi definicja:

Obiekt — byt, który posiada dobrze określone granice i własności oraz jest wyróżnialny w analizowanym fragmencie dziedziny problemowej.

Gdy w podrozdziale 2.4.1 (strona 26) omawialiśmy tę definicję, wspomnieliśmy o zasadzie tożsamości obiektu. Polega ona na tym, że obiekt jest rozpoznawany na podstawie samego faktu istnienia, a nie za pomocą jakiejś kombinacji swoich cech (bo przecież możemy mieć obiekt, który będzie ich pozbawiony). Jak pewnie się domyślasz, drogi Czytelniku, ta definicja jest dobra do teoretycznych rozważań, ale nie dla komputera, który potrzebuje bardziej „namacalnych" sposobów na rozróżnianie takich bytów. Zwykle realizowane jest to za pomocą wewnętrznego identyfikatora, który może przyjmować postać adresu w pamięci, gdzie obiekt jest przechowywany. Często określa się go mianem referencji do obiektu (np. w języku Java czy w MS C#).

3.1.2 Klasa

W popularnych językach programowania (Java, C#, C++) obiekt należy do jakiejś klasy. Przypomnijmy, co mówi definicja:

Klasa — nazwany opis grupy obiektów o podobnych własnościach.

Mniej formalnie można stwierdzić, że klasa opisuje obiekt, jego:

- zdolność do przechowywania informacji: atrybuty oraz asocjacje,
- zachowanie: metody.

Dobra wiadomość jest taka, że w powyższych językach programowania klasy występują w sposób zgodny z przytoczoną definicją. Niestety nie dotyczy to wszystkich pojęć znanych z obiektowości (UML).

Załóżmy, że potrzebna nam jest klasa opisująca film w wypożyczalni wideo (nawiązujemy do naszego głównego przykładu). Odpowiedni kod w języku Java mógłby wyglądać tak jak na listingu 3-1. Jak widać, jest on bardzo prosty i składa się tylko z dwóch słów kluczowych:

- `public` określa operator widoczności klasy. W zależności od użytego operatora klasa może być dostępna np. tylko dla innych klas z tego

samego pakietu. Więcej informacji na ten temat można znaleźć w dokumentacji języka Java lub dedykowanych książkach, np. [Ecke06].

```
**
 * Informacje o filmie.
*/
public class Film {
        /* Cialo klasy */
}
```

3-1 Kod klasy w języku Java

- `class` informuje kompilator, że dalej (w nawiasach klamrowych) znajduje się definicja klasy (jej szczegóły poznamy później).

3.1.3 Ekstensja klasy

Kolejnym ważnym pojęciem, chociaż niewystępującym wprost na diagramie klas, jest ekstensja klasy. Przypomnijmy definicję:

Ekstensja klasy	Zbiór aktualnie istniejących wszystkich wystąpień (innymi słowy: instancji lub jak kto woli: obiektów) danej klasy.

Myślę, że definicja jest dość jasna, więc od razu sprawdźmy, jak ma się do języków programowania. Otóż krótka i treściwa odpowiedź brzmi: „nijak". W językach takich jak Java czy C# nie ma czegoś takiego jak ekstensja klasy. Mamy na myśli fakt, że programista nie ma domyślnie dostępu do wszystkich aktualnie istniejących obiektów danej klasy. Czyli nie ma jakiegoś specjalnego słowa kluczowego języka czy innej konstrukcji, która udostępni odpowiednią listę. Możemy sami zaimplementować ekstensję klasy, korzystając z istniejących konstrukcji języka. Podkreślmy jeszcze raz: to, co zrobimy, jest pewnego rodzaju obejściem problemu, a nie natywnym, w myśl obiektowości, rozwiązaniem problemu. Z punktu widzenia kompilatora, nadal nie będzie ekstensji klasy, a tylko pewne konstrukcje, które programista będzie w ten sposób traktował.

Myślę, że dość oczywistym rozwiązaniem jest utworzenie dedykowanego kontenera, który będzie przechowywał referencje do poszczególnych wystąpień klasy (czyli instancji, czyli obiektów). Kwestią otwartą jest:

- Gdzie trzymać ten kontener?

- Kiedy i w jaki sposób dodawać do niego obiekty?

Jeżeli chodzi o pierwsze pytanie (Gdzie trzymać ten kontener) to mamy dwa ogólne podejścia:

- W ramach tej samej klasy biznesowej.
- Przy użyciu klasy dodatkowej, np. klasa Film, jej ekstensja np. klasa Filmy lub klasa Film, a jej ekstensja np. FilmEkstensja.

Obydwa podejścia mają swoje zalety i wady:

- Ktoś mógłby powiedzieć, że pierwsze rozwiązanie wprowadza nam do klas biznesowych pewne elementy techniczne,
- Ktoś inny mógłby krytykować drugi sposób za to, że mnoży byty: dla każdej klasy biznesowej tworzy odpowiadającą jej klasę zarządzającą ekstensją.

Jak widać, nie ma jednego, idealnego rozwiązania. Z powodów, które staną się jasne już niedługo, mimo wszystko chyba pierwszy sposób jest lepszy.

3.1.3.1 Implementacja ekstensji klasy w ramach tej samej klasy

Aby zaimplementować ekstensję klasy w języku programowania typu Java czy C++, musimy stworzyć kontener, który będzie przechowywał referencje do obiektów. Ponieważ chcemy go umieścić w klasie biznesowej, a jej wszystkie obiekty muszą mieć do niego dostęp, użyjemy atrybutu ze słowem kluczowym `static`.

Oprócz umieszczenia kontenera warto do klasy dodać odpowiednie metody, które ułatwią czynności dodawania czy usuwania obiektów.

```
     public class Film {
            // Implementacja czesci biznesowej
            public Film() {
                   // Dodaj do ekstensji
(1)                dodajFilm(this);
            }

            // Implementacja ekstensji

            /** Ekstensja. */
(2)         private static Vector<Film> ekstensja = new
            Vector<Film>();
```

```
(3)         private static void dodajFilm(Film film) {
                    ekstensja.add(film);
            }
(4)         private static void usunFilm(Film film) {
                    ekstensja.remove(film);
            }

            /** Wyswietla ekstensje. Metoda klasowa */
(5)         public static void pokazEkstensje() {
                    System.out.println("Ekstensja klasy Film: ");
                    for(Film film : ekstensja) {
                            System.out.println(film);
                    }
            }
    }
```

3-2 Implementacja zarządzania ekstensją w ramach tej samej klasy

Kod zawierający przykładową implementację jest przedstawiony na listingu 3-2. Ciekawsze rozwiązania (poniższe numery w nawiasach odnoszą się do odpowiednich miejsc na listingu):

- (1). Wróćmy na chwilę do naszego drugiego pytania dotyczącego sposobu dodawania obiektów do ekstensji. Pierwsze rozwiązanie, jakie się nasuwa, to po prostu ręczne, wywoływane przez programistę, dodawanie nowo utworzonego obiektu do kontenera. Czy to będzie działać? Oczywiście – chyba że ktoś zapomni to zrobić. Wtedy część obiektów będzie w ekstensji (bo pamiętał o wywołaniu), a część nie. Czy można coś na to poradzić? Użyjemy konstruktora, a konkretniej dodamy odpowiednie wołanie metody z kontenera w konstruktorze. Takie podejście gwarantuje nam, że każdy utworzony obiekt tej klasy będzie umieszczony w kontenerze.

- (2). Tutaj znajduje się deklaracja naszego kontenera. Skorzystaliśmy z tzw. klasy parametryzowanej, umożliwiającej przechowywanie określonego typu (oraz typów z niego dziedziczących). Musieliśmy skorzystać ze słowa kluczowego `static`, aby zapewnić dostęp do tego elementu ze wszystkich obiektów tej klasy. Opisywana implementacja wykorzystuje typ `Vector`, ale w zależności od konkretnych potrzeb można użyć innych ich rodzajów, np. `ArrayList`.

- (3), (4). Metody ułatwiające operowanie ekstensją: dodają oraz usuwają podane elementy. Ponieważ kontener, na którym operują, jest zadeklarowany jako `static`, to i metody muszą też tak być skonstruowane.

- (5). Metoda pomocnicza umożliwiająca wyświetlenie (na konsolę) obiektów znajdujących się w ekstensji klasy. Warto zwrócić uwagę na „nową" pętle `for`. Dzięki temu nie musimy wykorzystywać iteratorów, które są mniej wygodne w użyciu.

Przykładowy kod testujący to rozwiązanie umieszczono na listingu 3-3, a jego wynik działania przedstawia rysunek konsoli 3-1 Konsola. Czytelnik, który uruchomi ten program, otrzyma inne liczby (są one adresami w pamięci gdzie przechowywane są dane obiekty).

```
public static void main(String[] args) {
        // Test: Implementacja ekstensji w ramach tej samej klasy
        Film film1 = new Film();
        Film film2 = new Film();

        Film.pokazEkstensje();
}
```

3-3 Kod testujący implementację ekstensji w ramach tej samej klasy

```
Ekstensja klasy Film:
mt.mas.Film@126804e
mt.mas.Film@b1b4c3
```

3-1 Konsola po uruchomieniu przykładu z listingu 3-3

3.1.3.2 Implementacja ekstensji klasy przy użyciu klasy dodatkowej

Innym sposobem implementacji zarządzania ekstensją jest stworzenie dodatkowej klasy „technicznej" dla każdej klasy biznesowej, np. `Filmy` dla klasy biznesowej `Film` lub `FilmEkstensja`. Dzięki takiemu podejściu cała funkcjonalność związana z ekstensją umieszczona jest w oddzielnej klasie i nie „zaśmieca" nam klasy biznesowej. Dodatkową, potencjalną korzyścią jest możliwość operowania wieloma różnymi ekstensjami dla np. klasy `Film`. Chociaż praktyczna przydatność tego może być dyskusyjna. Przykładowy kod jest przedstawiony na listingu 3-4.

```
(1)  public class Film {
             /* Cialo klasy */
     }

     public class FilmEkstensja {
(2)          private Vector<Film> ekstensja = new Vector<Film>();

             public void dodajFilm(Film film) {
(3)                  ekstensja.add(film);
             }
             public void usunFilm(Film film) {
(4)                  ekstensja.remove(film);
             }
             public void pokazEkstensje() {
(5)                  System.out.println("Ekstensja klasy Film: ");
                     for(Film film : ekstensja) {
                             System.out.println(film);
                     }
             }
     }
```

3-4 Implementacja ekstensji klasy jako klasy dodatkowej

Ciekawsze fragmenty kodu implementacji (z listingu 3-4):

- (1). Klasa biznesowa, której ekstensją chcemy zarządzać.
- (2). Kontener przechowujący referencje do poszczególnych obiektów należących do ekstensji. Odwrotnie niż w poprzednim rozwiązaniu (podrozdział 3.1.3.1, strona 101), atrybut ten nie jest oznaczony jako `static` (chociaż może być). Dzięki temu, tworząc kolejne instancje klasy `FilmEkstensja`, możemy tworzyć wiele ekstensji dla jednej klasy.
- (3), (4) Metody umożliwiające dodawanie oraz usuwanie obiektów do/z ekstensji.
- (5). Pomocnicza metoda wyświetlająca ekstensję.

Porównajmy ze sobą dwie implementacje, a właściwie listingi (3-2 oraz 3-4) i zastanówmy się, czy w tym ostatnim czegoś nie brakuje? No tak - nie ma automatycznego dodawania do ekstensji w konstruktorze klasy biznesowej. Czy to przeoczenie? Oczywiście nie. Przy takiej implementacji nie znamy obiektu zarządzającego ekstensją i dlatego nie możemy napisać bezpo-

średniego odwołania. Możemy przekazywać go np. jako parametr do konstruktora obiektu biznesowego – tylko czy to jest wygodne? Pewnie nie.

Listing 3-5 pokazuje omawiane podejście w działaniu. Ciekawsze miejsca:

- (1). Tworzymy obiekt klasy `FilmEkstensja`, który będzie zarządzał ekstensją klasy biznesowej `Film`.
- (2). Tworzymy biznesowy obiekt opisujący pojedynczy film.
- (3). Ręczne dodanie utworzonego obiektu do ekstensji klasy. Bez tego nowego obiektu nie będzie w ekstensji.

```
    public static void main(String[] args) {
            // Test: Implementacja ekstensji przy uzyciu klasy
            dodatkowej

(1)         FilmEkstensja filmEkstensja = new FilmEkstensja();

            Film film1 = new Film();
(2)         filmEkstensja.dodajFilm(film1);
(3)
            Film film2 = new Film();
            filmEkstensja.dodajFilm(film2);

            filmEkstensja.pokazEkstensje();
    }
```

3-5 Klasa FilmEkstensja w działaniu

Tak jak wspomnieliśmy, pierwsze podejście (to z implementacją ekstensji w ramach tej samej klasy biznesowej) wydaje się wygodniejsze. Wrócimy do tego tematu jeszcze trochę później – podrozdział 3.1.7, strona 124.

3.1.4 Atrybuty

W podrozdziale 2.4.2.1 (strona 28 i dalsze) omawialiśmy różne rodzaje atrybutów. Teraz zobaczymy, jak ich teoretyczne wyobrażenia mają się do języków programowania (głównie do języka Java).

3.1.4.1 Atrybuty proste

Tutaj sprawa jest dość łatwa. Ten rodzaj atrybutów występuje we wszystkich popularnych językach programowania, w takiej postaci jak w obiektowości (w UML). Listing przedstawia przykładowy atrybut przechowujący cenę dla klasy `Film`.

```
public class Film {
        private float cena;
}
```

3-6 Przykładowy atrybut prosty dla klasy Film

3.1.4.2 Atrybuty złożone

Atrybut złożony jest opisywany za pomocą dedykowanej klasy (np. data). Klasa ta może być dostarczana przez twórców danego języka programowania, bibliotekę zewnętrzną lub stworzona przez użytkownika (programistę). W efekcie:

- W klasie biznesowej (np. Film) przechowujemy referencję do jego wystąpienia, a nie (złożoną) wartość.

- W związku z powyższym możemy go współdzielić, np. inny obiekt może przechowywać referencję do tej samej daty. Stoi to (trochę) w sprzeczności do „teoretycznej" semantyki atrybutu złożonego, który nie powinien być współdzielony. W języku C++ sytuacja jest trochę inna – tam można przechowywać obiekt innej klasy jako „wartość" (więcej na ten temat można znaleźć w książkach poświęconych C++, np. [Gręb96]).

```
public class Film {
        private Date dataDodania;
}
```

3-7 Przykładowy atrybut złożony dla klasy Film

Możemy rozważyć jeszcze jedno podejście, polegające na bezpośrednim umieszczeniu zawartości atrybutu złożonego w klasie. Przykładowo zamiast atrybutu złożonego adres, możemy w klasie umieścić atrybuty proste: ulica, numer domu, nr mieszkania, miasto i kod pocztowy. Czasami takie rozwiązanie może być wygodniejsze niż tworzenie oddzielnej, dedykowanej klasy.

3.1.4.3 Atrybuty wymagane oraz opcjonalne

W tym przypadku musimy indywidualnie przeanalizować dwa rodzaje atrybutów:

- Proste. Każdy atrybut prosty przechowuje jakąś wartość – nie może nie przechowywać. Nawet jeżeli podstawimy tam zero to i tak jest to jakaś wartość – właśnie 0.

- Złożone. Atrybut złożony przechowuje referencję do obiektu „będącego jego wartością". Ponieważ jest to referencja, może mieć wartość `null`, czyli „brak wartości". Dla bezpieczeństwa (aby nie otrzymać wyjątku), należy „ręcznie" sprawdzać, czy jest różna od `null`.

Z powyższego wywodu wynika, że atrybuty złożone mogą bez problemu być albo wymagane, albo opcjonalne. Gorzej jest z atrybutami prostymi – czy nic się nie da zrobić i nie jesteśmy w stanie przechować opcjonalnej informacji o np. pensji? Oczywiście, że się da – tylko będzie to trochę bardziej kłopotliwe. Atrybut prosty zaimplementujemy jako klasę przechowującą prostą wartość. Dzięki temu będziemy w stanie podstawić `null`, czyli właśnie informację o braku wartości. W języku Java dla podstawowych typów istnieją już takie klasy opakowujące, np. `Integer` dla typu `int`.

3.1.4.4 Atrybuty pojedyncze

Dokładnie taka sama semantyka jak w obiektowości. Jeden atrybut przechowuje jedną wartość, np. imię (pod warunkiem, że ktoś ma tylko jedno; jeżeli nie, to patrz dalej).

3.1.4.5 Atrybuty powtarzalne

Wiele wartości dla takiego atrybutu przechowujemy w jakimś kontenerze lub zwykłej tablicy. To pierwsze rozwiązanie jest preferowane, gdy nie wiemy, ile będzie tych elementów lub ich liczba będzie się zmieniać. Rodzaj wybranego kontenera może zależeć od sposobu pracy z takim atrybutem, np. czy częściej dodajemy elementy, czy może raczej odczytujemy itp.

Teoretycznie można sobie wyobrazić jeszcze jedno podejście, ale nie jest ono zalecane i zaliczyłbym je do „sztuczek" (a te, jak wiadomo, wcześniej czy później są przyczyną kłopotów w programowaniu). Możemy przechować wiele np. nazwisk w pojedynczym `Stringu`, oddzielając je np. przecinkami. Poziom komfortu oraz bezpieczeństwa pracy z tego typu „atrybutem powtarzalnym" jest bardzo niski.

3.1.4.6 Atrybuty obiektu

Taka sama semantyka jak w obiektowości (w UML). Każdy obiekt w ramach konkretnej klasy może przechowywać własną wartość w ramach takiego atrybutu.

3.1.4.7 Atrybuty klasowe

Sposób realizacji zależy od podejścia do ekstensji:

- Ekstensja w ramach tej samej klasy. Stosujemy atrybuty klasowe w tej samej klasie ze słowem kluczowym `static`,
- Ekstensja jako klasa dodatkowa. Implementujemy atrybuty klasowe w klasie dodatkowej (bez słowa kluczowego `static`).

3.1.4.8 Atrybuty wyliczalne

W języku Java czy C++ nie ma natywnego sposobu na implementację atrybutów wyliczalnych. Ich działanie „symulujemy" w oparciu o metody, co oznacza, że tak naprawdę w kodzie odnosimy się do metod, a nie do atrybutów. W przypadku hermetyzacji ortodoksyjnej[8] nie jest to wielkim problemem, ponieważ wszystkie atrybuty i tak są ukryte, a dostęp do nich odbywa się w oparciu o metody (dla Java tak zwane *settery* i *gettery,* czyli takie metody, które umożliwiają zmianę wartości oraz jej odczyt). Specjalne traktowanie atrybutu zaimplementowane jest w ciele metody udostępniającej/zmieniającej jego wartość.

Chociaż książka ta bazuje głównie na języku Java, nie sposób nie wspomnieć przy tej okazji o doskonałym mechanizmie zaimplementowanym w języku MS C#: właściwości (*properties*). Polega on na tym, że możemy

[8] Ogólnie można powiedzieć, że hermetyzacja polega na ukrywaniu informacji ze szczególnym uwzględnieniem atrybutów. Dostęp do takich ukrytych atrybutów jest uzyskiwany za pomocą metod. Dzięki temu mamy większą kontrolę i możemy np. przeciwdziałać nieautoryzowanemu zmienianiu ich wartości. Programiści od dawna spierają się, czy takie podejście jest użyteczne (bo wymaga trochę więcej pracy na pisanie tych metod – chociaż w tym coraz częściej mogą nas wyręczać nowoczesne środowiska programistyczne). Hermetyzacja ortogonalna polega na tym, że decyzja na temat ewentualnego ukrycia dowolnego elementu (atrybut, metoda) zależy tylko od programisty. Zgodnie z hermetyzacją ortodoksyjną wszystkie atrybuty są ukryte i nie ma możliwości ich udostępniania. Oczywiście języki C++, C# oraz Java są zgodne z filozofią hermetyzacji ortogonalnej.

definiować specjalne „metody", których używamy dokładnie tak jak atrybutów. Przykład dla atrybutu `cena` pokazany jest na listingu 3-8.

```
private float cena {
        get {
                return cena_netto * 1.22;
        }
        set {
                cena_netto = value / 1.22;
        }
}
```

3-8 Przykład wykorzystania właściwości w języku MS C#

3.1.5 Metody

W tym podrozdziale omówimy dwa zasadnicze rodzaje metod oraz kilka pojęć z nimi związanych.

3.1.5.1 Metoda obiektu

Metody obiektu w języku Java, C# czy C++ mają taką samą semantykę jak w obiektowości (UML). Konkretna metoda operuje na obiekcie, na rzecz którego została wywołana. Ma dostęp do jego wszystkich elementów: atrybutów oraz innych metod. W ciele metody możemy używać specjalnego słowa kluczowego `this`, które jest referencją na obiekt, na rzecz którego metoda została wywołana (więcej na ten temat w książkach poświęconych językowi Java, np. [Ecke06]. Przykładowy kod bardzo prostej metody jest pokazany na listingu 3-9.

```
public float getCena() {
        return cena;
}
```

3-9 Przykładowy kod metody w języku Java

3.1.5.2 Metoda klasowa

Metody klasowe w rozumieniu obiektowości niestety nie występują w popularnych językach programowania. Tutaj niektórzy Czytelnicy mogliby zaprotestować: chwileczkę, a słowo kluczowe `static`? Przecież rozwiązało problem atrybutów klasowych – czy tutaj nam nie pomoże? Częściowo tak, ale nie do końca – proponuję przypomnieć sobie, co pisaliśmy o metodach klasowych w podrozdziale 2.4.2.2 (strona 33). A może ktoś wie bez zaglądania?

Metoda klasowa charakteryzuje się następującymi cechami:

- Można ją wywołać na rzecz klasy. Dzięki temu możemy jej używać nawet gdy nie ma żadnych obiektów danej klasy. I to możemy uzyskać za pomocą słowa kluczowego `static` występującego i w Java, i w C++, jak również w C#.

- Drugą, bardzo ważną cechą jest dostęp do ekstensji klasy. I tutaj zaczyna się problem. Powiedzieliśmy, że w językach, które nas interesują, nie ma ekstensji klasy, co oczywiście oznacza, że i metoda ze słowem `static` nie może mieć do niej automatycznie dostępu.

W efekcie musimy sami jakoś zaimplementować metodę klasową. Nasze podejście zależy od sposobu zarządzania ekstensją:

- Ekstensja w ramach tej samej klasy. Metody klasowe w tej samej klasie ze słowem kluczowym `static`. Ponieważ w tej samej klasie istnieje kontener też zadeklarowany ze słowem kluczowym `static`, nasza metoda klasowa ma bezproblemowy dostęp do niego,a co za tym idzie, do ekstensji.

- Ekstensja jako klasa dodatkowa. Metody klasowe umieszczamy w klasie dodatkowej – tym razem bez słowa kluczowego `static`.

3.1.5.3 Przeciążenie metody

Przeciążenie metody nie jest konstrukcją czysto obiektową, ponieważ nie wykorzystuje jakichś szczególnych pojęć z obiektowości. Mimo wszystko jest wykorzystywana chyba we wszystkich współczesnych językach obiektowych i dlatego warto ją omówić. Przeciążenie metody polega na stworzeniu metody o takiej samej nazwie jak metoda przeciążana, ale różnej liczbie i/lub typie parametrów. Po co nam druga „taka sama" metoda? Czy to nam się do czegoś przyda? Spójrzmy na przykład z listingu 3-10. W jakiejś klasie, np. kaseta jest metoda zwracająca jej cenę netto. Co zrobić, jeżeli chcemy dowiedzieć się o cenę brutto? Możemy utworzyć metodę, np. `getCenaBrutto()`. Innym sposobem jest przeciążenie podstawowej metody za pomocą parametru określającego stawkę VAT (w procentach). Dzięki temu, odwołując się do niej w kodzie programu, nie musimy przypominać sobie jej szczególnej nazwy – po prostu wołamy metodę zwracającą cenę uzupełnioną o parametr.

```
public float getCena() {
   return cena;
}
public float getCena(float stawkaVAT) {
   return cena * (1.0f + stawkaVAT / 100.0f);
}
```

3-10 Przykład wykorzystania przeciążania metod

3.1.5.4 Przesłonięcie metody

Aby dobrze zrozumieć przesłanianie metod, należy najpierw dobrze orientować się w kwestii dziedziczenia. W związku z tym wrócimy do tego zagadnienia później (patrz podrozdział 3.3.2, strona 176).

3.1.6 Trwałość ekstensji

Ekstensja klasy jest trwała, gdy jej obiekty „przeżyją" wyłączenie systemu – po ponownym włączeniu systemu będziemy mieli te same obiekty. O ważności tej cechy nie trzeba chyba nikogo przekonywać. W takiej czy innej formie występuje w prawie wszystkich systemach komputerowych: poczynając od gier (zapamiętujemy stan gry, który jest właśnie definiowany przez stan obiektów w grze), poprzez aplikacje biurowe (np. edytor, w którym piszę tę książkę), a kończąc na bazach danych.

W językach programowania takich jak Java, MS C# czy C++ cecha ta nie występuje bezpośrednio. Mam tu na myśli fakt, że nie ma jakiegoś specjalnego słowa kluczowego, którego użycie razem z np. definicją klasy zapewni trwałość jej ekstensji[9]. Dlatego trzeba ją jakoś zaimplementować. Jednym ze sposobów jest „ręczne" zapamiętywanie ekstensji na dysku, a następnie wczytywanie. Inne, bardziej nietypowe może być wykorzystywanie takich pamięci w komputerze, które nie tracą swojego stanu po wyłączeniu zasilania (np. *Flash*). Ale to raczej temat na inną książkę.

Najczęstszym sposobem realizacji trwałości jest skorzystanie z pliku dyskowego. Polega to na zapisie danych do pliku oraz ponownym ich wczytaniu do pamięci po ewentualnym wyłączeniu oraz włączeniu oprogramowania. Takie podejście niesie ze sobą pewien problem – ktoś wie jaki? Najkrócej można go scharakteryzować w sposób następujący: po wczytaniu nie mamy tych samych obiektów, ale takie same. Czy to jest problem? Owszem może być. Po pierwsze, nie jest to do końca idealna trwałość, bo w niej

[9] Może to wynika częściowo też z tego, że w wymienionych językach nie ma ekstensji klasy ;)

mamy te same obiekty. Po drugie, co ważniejsze z praktycznego punktu widzenia, narażamy się na problemy wynikające z faktu istnienia powiązań pomiędzy obiektami. Jeżeli używamy natywnych referencji języka programowania, to oznaczają one konkretne miejsca w pamięci, gdzie rezydują powiązane obiekty. Jeżeli wczytamy je z pliku i na nowo utworzymy, to na 99% będą umieszczone gdzie indziej, a odczytane referencji pokazują na „stare" miejsca. Trzeba sobie z tym jakoś poradzić – pokażemy, jak to zrobić, w następnych rozdziałach.

Na szczęście nowsze języki programowania takie jak Java oraz C# (zauważ, Czytelniku, że tym razem nie wymieniamy C++) udostępniają mechanizm zwany serializacją. Technika ta znacząco automatyzuje i ułatwia zapewnienie trwałości danych. W następnych rozdziałach przeanalizujemy te dwa podejścia: ręczne oraz (pół) automatyczne.

3.1.6.1 Ręczna implementacja trwałości danych

W porównaniu z wykorzystaniem serializacji ręczna implementacja ma prawie same zalety – oprócz jednej. Tą jedyna wadą jest dość duża pracochłonność – szczególnie, jeżeli chce się optymalnie rozwiązać problem z powiązaniami obiektów. Wracając do zalet tego podejścia:

- Szybkość. Bezpośrednio wybieramy elementy, które mają być zapisywane oraz tryb (np. binarny, tekstowy, strukturalno-tekstowy) tego zapisu. Dzięki temu wykorzystujemy maksymalną szybkość oferowaną przez dany język programowania.

- Duża kontrola nad efektem końcowym. To my – programiści jesteśmy odpowiedzialni za całą implementację, co umożliwia całkowitą kontrolę tego, co zapisujemy (bo nie musimy chcieć zapamiętywać wszystkiego) oraz jak zapisujemy.

- Większa odporność na zmiany w kodzie utrwalanych klas. Do tego wrócimy przy okazji opisu metody z serializacją.

- Mały plik. Dzięki temu, że zapisujemy tylko wybrane elementy, plik może być tak mały jak tylko się da to uzyskać, chcąc zapamiętać określoną ilość danych, np. w jednym z projektów nad którym pracowałem, zapisywane pliki były dość duże. Przyczyną było zapamiętywanie kolejnych współrzędnych położenia na ekranie pewnych elementów (jako liczba typu `int` – zajmują 4 bajty każda). Ponieważ

ich wartości były z zakresu 0 – 1000, zdecydowaliśmy się na typ `short` (2 bajty). Po analizie danych zastosowaliśmy jeszcze sprytniejsze rozwiązanie – ktoś ma jakiś pomysł? Zapamiętujemy tylko różnice pomiędzy współrzędnymi. Ponieważ są one stosunkowo niewielkie to mieszczą się w zakresie 0 - 255. Dzięki temu możemy je zapisywać korzystając z jednego bajta. Jak widać, czasami sposób zapisu danych ma niewiele wspólnego z ich fizycznym przechowywaniem w klasie.

Oczywiście ręcznych sposobów implementacji trwałości danych jest bardzo dużo. Jedno z możliwych rozwiązań znajduje się na listingu 3-11.

W każdej z klas biznesowych umieszczamy metody odpowiedzialne za zapis oraz odczyt danych. Komentarze do listingu 3-11:

- (1). Standardowe atrybuty przechowujące dane biznesowe.

- (2). Metoda dokonująca zapisu danych biznesowych do pliku. A dokładniej rzecz biorąc, do strumienia. (Dlaczego nie do pliku? Dzięki tak podzielonej funkcjonalności można zapisywać również inne ekstensje do tego samego strumienia. Dzięki temu otrzymujemy jeden plik zawierający wszystkie ekstensje z naszego systemu.) w jej ciele, po kolei zapisujemy, atrybut po atrybucie.

- (3). Metoda odczytująca dane. Zwróćmy uwagę na kolejność – jest ona identyczna z kolejnością zapisu.

```
     public class Film {
(1)          private String tytul;
             private float cena;
             private Date dataDodania;

(2)          private void write(DataOutputStream stream) throws
             IOException {
                     stream.writeUTF(tytul);
                     stream.writeFloat(cena);
                     stream.writeLong(dataDodania.getTime());
             }

(3)          private void read(DataInputStream stream) throws
             IOException {
                     tytul = stream.readUTF();
                     cena = stream.readFloat();
                     long time = stream.readLong();
                     dataDodania = new Date(time);
             }
     }
```

3-11 Przykładowa implementacja „ręcznej" trwałości danych – zapis i odczyt stanu obiektu

Oprócz metod zapisujących i odczytujących stan pojedynczego obiektu, potrzebujemy metod zapisujących oraz odczytujących stan całej ekstensji. Naturalnie umieszczamy je w klasie zarządzającej ekstensją lub w ramach samej klasy biznesowej (jeżeli takie rozwiązanie wybraliśmy dla przechowywania ekstensji). Przykładowa implementacja jest pokazana na listingu 3-12. Ważniejsze miejsca:

- (1). Ze względu na oszczędność miejsca nie pokazano atrybutu klasowego przechowującego ekstensję oraz konstruktora dodającego nowo utworzony obiekt do kontenera. Elementy te są na listingu 3-2 (strona 102).
- (2). Metoda zapisująca ekstensję do podanego strumienia.
- (3). Na początku zapisujemy liczbę obiektów w ekstensji – będzie to potrzebne przy odczycie.
- (4). Następnie iterujemy po całym kontenerze zawierającym referencje do poszczególnych obiektów i wywołujemy metodę zapisującą pojedynczy obiekt (pokazaną na listingu 3-11).
- (5). Metoda odczytująca zawartość ekstensji z podanego strumienia.
- (6). Najpierw odczytujemy liczbę zapisanych obiektów.
- (7). Czyścimy dotychczasową zawartość ekstensji (usuwamy wszystkie aktualnie istniejące obiekty danej klasy).
- (8). Znając liczbę zapisanych obiektów, w pętli odczytujemy ich zawartość, wywołując metodę dla każdego z nich.

```
(1)  public class Film {
(2)          public static void zapiszEkstensje(DataOutputStream
```

```
            stream) throws IOException {
                    stream.writeInt(ekstensja.size());
(3)                 for(Film film : ekstensja) {
(4)                         film.write(stream);
                    }
            }

            public static void odczytajEkstensje(DataInputStream
(5)         stream) throws IOException {
                    Film film = null;
                    int liczbaObiektow = stream.readInt();
                    ekstensja.clear();
(6)                 for(int i = 0; i < liczbaObiektow; i++) {
(7)                         film = new Film();
(8)                         film.read(stream);
                    }
            }
    }
```

3-12 Przykładowa implementacja „ręcznej" trwałości danych – zapis i odczyt ekstensji

Przykład z listingu 3-13 pokazuje zapis oraz odczyt przykładowej ekstensji. Ważniejsze fragmenty:

- (1). Określenie lokalizacji pliku na dysku, który będzie przechowywał ekstensję.
- (2). Utworzenie dwóch przykładowych instancji (obiektów) klasy Film. Dzięki specjalnej konstrukcji konstruktora obiekty są automatycznie dodawane do ekstensji.
- (3). Utworzenie strumienia wyjściowego podłączonego do pliku.
- (4). Wywołanie metody klasowej powodującej zapisanie ekstensji do podanego strumienia.
- (5). Zamknięcie strumienia. Jest to bardzo ważny element, którego pominięcie może skutkować różnymi błędami, np. blokadą pliku czy utratą części „zapisanych"[10] danych.

[10] w przypadku strumieni buforowanych zapis nie odbywa się natychmiast, ale pewnymi paczkami. Dzięki temu mocno zyskujemy na wydajności, ale ryzykujemy, że w przypadku, np. odcięcia zasilania czy wystąpienia jakiegoś błędu, dane tak naprawdę nie zostaną zapisane do pliku.

- (6). Utworzenie strumienia do czytania z pliku.
- (7). Wywołanie metody klasowej odczytującej zawartość pliku.
- (8). Zamknięcie strumienia.
- (9). Niezbędna obsługa wyjątków – nie pokazana ze względu na oszczędność miejsca.

```
(1)   final String ekstensjaPlik = "d:\\Ekstensja.bin";

(2)   Film film1 = new Film("Terminator 1", new Date(), 29.90f);
      Film film2 = new Film("Terminator 2", new Date(), 34.90f);

      try {
             // Zapisz ekstensje do strumienia
             DataOutputStream out2 = new DataOutputStream(new
                 BufferedOutputStream(new
(3)          FileOutputStream(ekstensjaPlik)));
             Film.zapiszEkstensje(out2);
             out2.close();

(4)          // Odczytaj ekstensje ze strumienia
(5)          DataInputStream in2 = new DataInputStream(new
             BufferedInputStream(new
                     FileInputStream(ekstensjaPlik)));
(6)          Film.odczytajEkstensje(in2);
             in2.close();
      } catch ( … ) {
(7)          // …
(8)   }
(9)
```

3-13 Kod testujący zapis oraz odczyt przykładowej ekstensji

Rysunek pokazuje zawartość konsoli po uruchomieniu programu z listingu 3-13. Warto zwrócić uwagę na fakt, że liczby po znaku „@” (jak wspomnieliśmy są to adresy w pamięci) są inne przed zapisem i po odczycie. Jest to ilustracja naszego stwierdzenia, że wykorzystanie zapisu do pliku skutkuje tym, iż otrzymujemy takie same obiekty (bo tytuły filmów się zgadzają), ale nie te same obiekty (bo ich komputerowa tożsamość (referencje - adresy) jest inna).

```
Ekstensja klasy Film:
Film: Terminator 1, id: mt.mas.Film@27391d
Film: Terminator 2, id: mt.mas.Film@116ab4e
Ekstensja klasy Film:
Film: Terminator 1, id: mt.mas.Film@1434234
Film: Terminator 2, id: mt.mas.Film@af8358
```

3-2 Efekt działania programu utrwalającego ekstensję klasy

Przedstawiony sposób ręcznej implementacji trwałości jest bardzo prosty. Sprawdza się całkiem nieźle przy zapisie poszczególnych instancji. Niestety nie pozwala na właściwe traktowanie powiązanych obiektów. Spójrzmy na rysunek 3-70. Przedstawia on fragment dwóch ekstensji klas: Film oraz Aktor. Obiekty z tych klas są ze sobą powiązane w dwóch kierunkach, tzn. film ma połączenie do aktorów, którzy w nim grali oraz aktor jest połączony ze wszystkimi filmami, w których grał. Na czym polega problem? Otóż zastanówmy się, jak byśmy zapisywali te ekstensje do pliku oraz później je odczytywali:

- Tak jak w zaprezentowanym podejściu, próbujemy zapisać obiekty z klasy film – jeden po drugim. Ale jak zapamiętamy informacje o aktorach danego filmu? Powiedzieliśmy wcześniej, że zapis referencji jako liczby nie zadziała, ponieważ przy odczycie obiekt znajdzie się w innym miejscu pamięci (więc i referencja powinna być inna).

- Możemy spróbować zapisywać w ten sposób:

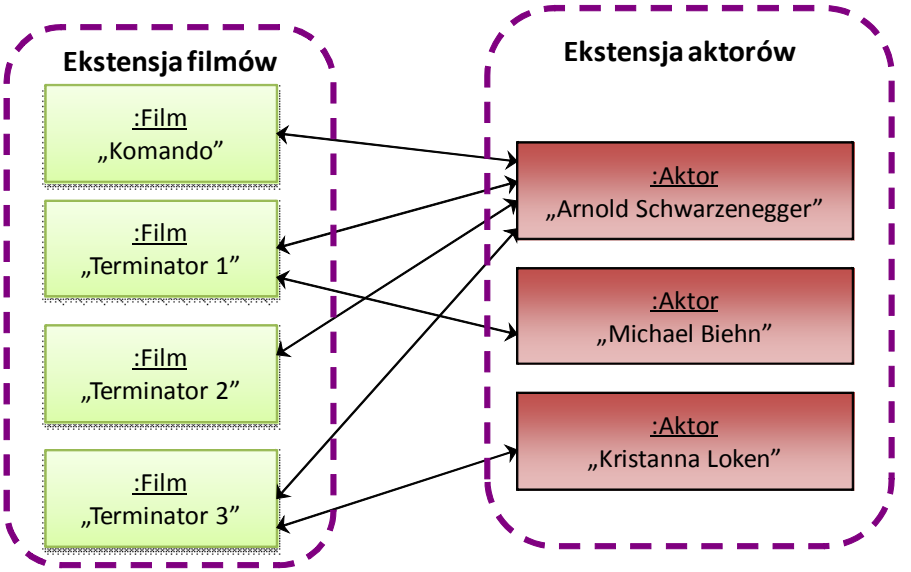
    - Tak jak dotychczas zapisujemy poszczególne atrybuty,

    - Gdy natrafimy na obiekt, to wywołujemy na jego rzecz metodę `write()` (z listingu 3-11, strona 114). Analogicznie jak wywoływaliśmy taką metodę z poziomu zapisu ekstensji. Czy to zadziała? Widzę tu dwa potencjalne problemy: zapętlenie się (z filmu wywołamy zapis aktora, z aktora zapis filmu itd. – trzeba to jakoś rozwiązać) oraz wielokrotny zapis tych samych elementów.

    - Wielokrotny zapis tych samych elementów polega na tym, że na ten sam obiekt może pokazywać wiele różnych obiektów. Stosując taką prostą metodę, informacje o aktorze „Ar-

nold Schwarzenegger” zapiszemy przy okazji utrwalania każdego przykładowego filmu (bo ten aktor grał w każdym z nich).

- Jak widać, nie jest to najlepszy sposób. Można spróbować innego podejścia:
    - Każdy z obiektów ma własny, unikatowy identyfikator.
    - Gdy natrafimy na konieczność zapisu informacji o obiekcie, zapamiętujemy jego id, a nie referencję.
    - Odczyt tak zapisanego pliku przebiega dwutorowo: najpierw wczytujemy dane z identyfikatorami, a następnie podmieniamy identyfikatory na natywne referencje (bo zazwyczaj chcemy pracować z referencjami – wrócimy do tego tematu później w podrozdziale 3.2 na stronie 131).



3-70 Przykład ilustrujący powiązania pomiędzy obiektami

Jak widać, prawidłowa implementacja trwałości obiektów nie jest sprawą trywialną - oczywiście, jeżeli chcemy optymalnie gospodarować zasobami. Nie przedstawiamy przykładowej implementacji, bo byłaby ona dość rozbudowana, ale zachęcamy do samodzielnych prób w tym zakresie.

3.1.6.2 Implementacja trwałości danych w oparciu o serializację

Serializacja jest mechanizmem zaimplementowanym w ramach bibliotek języka Java. Umożliwia automatyczne:

- zapisywanie grafu obiektów do strumienia,
- odczytywanie grafu obiektów ze strumienia.

Co to znaczy, że zapisujemy/odczytujemy cały graf obiektów? Na tym właśnie polega użyteczność tego mechanizmu. Wróćmy na chwilę do przykładu z rysunku 3-70 (strona 118). Zapisując informacje o obiekcie klasy film o nazwie „Terminator 1", zapisujemy również informacje o wszystkich obiektach, na które on pokazuje czyli „Arnold Schwarzenegger" oraz „Michael Biehn" (oba z klasy `Aktor`). Ale jak pewnie się domyślasz, drogi Czytelniku, zapisywane są też informacje o wszystkich obiektach wskazywanych przez te wskazywane obiekty, czyli w tym przykładzie wszystkie filmy oraz wszyscy aktorzy. Innymi słowy, mechanizm serializacji dba, aby wszystkie osiągalne obiekty (nieważne przez ile obiektów pośredniczących trzeba „przejść") były w prawidłowym stanie. I robi to tak sprytnie, że rozwiązuje problemy, które wcześniej wymieniliśmy: zapętlenia się oraz wielokrotnego zapisu tych samych obiektów.

Jeżeli chcemy, aby zapis odbywał się z optymalnym wykorzystaniem zasobów, to musimy wszystkie serializowane elementy wysyłać do jednego strumienia – nawet te pochodzące z różnych klas. W przeciwnym wypadku „komputer" nie będzie w stanie wychwycić tych powtórzeń. Z tego powodu, serializacja do wielu plików (np. jeden plik na jedną ekstensję) jest ewidentnym błędem – i tak w każdym pliku zapisują się całe grafy powiązanych obiektów, obejmujące wiele ekstensji (gdy obiekty z wielu ekstensji są ze sobą powiązane).

Jedynym wymogiem, który trzeba spełnić, aby korzystać z serializacji, jest „specjalna" implementacja przez klasę (oraz wszystkie jej elementy składowe) interfejsu `Serializable`. Owa specjalność implementacji interfejsu polega na tym, że w najprostszym przypadku deklarujemy jego implementację przez klasę, ale nie musimy ręcznie implementować jego metod. Tym zajmie się „kompilator" języka Java. Interfejs ten musi być zaimple-

mentowany nie tylko przez obiekt, od którego zaczyna się zapis (graf obiektów), ale przez wszystkie obiekty, które są zapisywane. W przeciwnym wypadku zobaczymy informację o wyjątku. Dobra wiadomość jest taka, że większość klas udostępnianych przez języka Java czy C# implementuje ten interfejs.

Z punktu widzenia programisty możemy wymienić następujące cechy takiego podejścia do trwałości ekstensji:

- Łatwość użycia. W najprostszym przypadku polega to na dodaniu dwóch słów do definicji klasy.
- Mniejsza szybkość działania.
- Większy plik niż w przypadku zapisu ręcznego. Te dwie ostatnie cechy są spowodowane tym, że razem z biznesowymi danymi klasy zapisywane są różnego rodzaju informacje techniczne. Nie dość, że zajmują miejsce, to jeszcze ich pozyskanie trochę spowalnia cały proces. Kolejną przyczyną spowolnienia jest konieczność uzyskania informacji o budowie klasy w trakcie działania programu.
- Dość duża wrażliwość na zmiany w kodzie klas. Tak się składa, że większość zmian, które wprowadzimy do kodu źródłowego wykorzystującego serializację, spowoduje brak kompatybilności zapisanych danych. Innymi słowy: serializujemy obiekty klasy, wprowadzamy zmiany do kodu źródłowego, a przy próbie odczytu danych możemy otrzymać wyjątek. Taka sytuacja może mieć miejsce nawet wtedy, gdy nie zmienimy atrybutów klasy (bo to byłoby dość oczywiste), ale nawet wtedy, gdy np. dodamy jakąś metodę.

Na szczęście mamy pewne możliwości kontrolowania sposobu działania serializacji:

- Dodanie i przesłonięcie (własną implementacją zapisu/odczytu – podobnie jak robiliśmy w listingu 3-11) poniższych metod:
  - `private void writeObject(ObjectOutputStream stream) throws IOException`
  - `private void readObject(ObjectInputStream stream) throws IOException, ClassNotFoundException`

- Oznaczenie wybranych atrybutów słowem kluczowym `transient`. Dzięki temu nie będą one automatycznie zapisywane.

Listing 3-14 zawiera kod klasy biznesowej z zadeklarowaną implementacją interfejsu `Serializable`. Tak jak wspomnieliśmy, warto zwrócić uwagę, że tak naprawdę w klasie nie umieszczamy metod znajdujących się w tym interfejsie. Czyli tak jak obiecaliśmy, cała sprawa polega na dodaniu dwóch słów: `implements Serializable`. Odpowiednikiem tego kodu dla implementacji ręcznej jest kod z listingu 3-11 (strona 114), a szczególnie metody `read()` oraz `write()`. Prawda, że prostsze?

```
public class Film implements Serializable {
        private String tytul;
        private float cena;
        private Date dataDodania;
}
```

3-14 Implementacja interfejsu `Serializable`

Listing 3-15 pokazuje metody zapisujące oraz odczytujące ekstensję. Warto zwrócić uwagę, że zapisujemy po prostu kolekcję. Dzięki temu, że serializacja zapisuje cały graf (patrz wcześniej), to nie musimy zapisywać obiektu po obiekcie. Warto to porównać z rozwiązaniem ręcznym – listing 3-12 (strona 115). Prostsze, nieprawdaż? Ważniejsze miejsca programu:

- (1). Metoda zapisująca ekstensję do podanego strumienia.

- (2). Wywołanie metody z klasy `ObjectOutputStream`, która zapisuje podany obiekt (a tak naprawdę cały graf obiektów, poczynając od tego, który podaliśmy).

- (3). Metoda odczytująca ekstensję z podanego strumienia. Zauważ, drogi Czytelniku, że nie musimy czyścić kontenera (jak poprzednio) – po prostu jego zawartość zostaje podmieniona po odczytaniu.

- (4). Wywołanie metody z klasy `ObjectInputStream`, która odczytuje zapisany obiekt i go zwraca. Warto zwrócić uwagę, że dokonywana jest konwersja na typ, który podamy. Stąd wniosek, że musimy dokładnie wiedzieć, co odczytujemy.

```
public class Film implements Serializable {
```

```
            public static void zapiszEkstensje(ObjectOutputStream
(1)         stream) throws IOException {
                    stream.writeObject(ekstensja);
(2)         }

            public static void odczytajEkstensje(ObjectInputStream
            stream) throws IOException {
(3)                 ekstensja = (Vector<Film>) stream.readObject();
            }
(4)  }
```

3-15 Utrwalenie ekstensji za pomocą serializacji

I pozostało nam jeszcze zademonstrowanie wykorzystania serializacji – jest bardzo podobne do podejścia manualnego – listing 3-16. Jak widać, tworzymy strumień do zapisu (1) oraz wołamy metodę zapisującą (2). Następnie tworzymy strumień do odczytu (3) oraz wołamy metodę odczytującą (4). Oczywiście musimy pamiętać o właściwej obsłudze wyjątków (5).

```
    try {
            // Zapisz ekstensje do strumienia
(1)         ObjectOutputStream out = new ObjectOutputStream(new
            FileOutputStream(ekstensjaPlik));
(2)         Film.zapiszEkstensje(out);

            // Odczytaj ekstensje ze strumienia
(3)         ObjectInputStream in = new ObjectInputStream(new
            FileInputStream(ekstensjaPlik));
(4)         Film.odczytajEkstensje(in);
(5)  } catch (FileNotFoundException e) { ...
```

3-16 Wykorzystanie serializacji

I jeszcze na koniec informacja, że faktycznie pliki zserializowane są większe od tych ręcznie wytworzonych. Plik wygenerowany przez przykład zajmował:

- 56 bajtów w przypadku ręcznej implementacji,
- 354 bajty dla pliku powstałego przez serializację (dla tych samych danych).

Dla większych porcji danych te różnice są mniejsze – około dwóch razy na niekorzyść serializacji.

3.1.6.3 Inne sposoby uzyskiwania trwałości danych

Powyżej opisane sposoby utrwalania ekstensji (danych) nie wykorzystują dodatkowych komponentów – czy to w postaci bibliotek, czy zewnętrznych aplikacji. To nie są jedyne możliwości, jakie ma programista. Można wyróżnić jeszcze co najmniej dwa podejścia:

- Wykorzystanie bazy danych.
    - Do największych wad tego rozwiązania należy na pewno zaliczyć konieczność mapowania struktur języka Java na konstrukcje z bazy danych. Ponieważ aktualnie najpopularniejsze rozwiązania z baz danych są oparte na technologiach relacyjnych, użytkownik jest zmuszony do zrezygnowania z większości[11] konstrukcji obiektowych. Dlatego też należy zastąpić wszystkie te pojęcia występujące w obiektowości, równoważnymi konstrukcjami z modelu relacyjnego. Wspomniana równoważność prowadzi zwykle do znacznego skomplikowania modelu projektowego. Więcej na ten temat można znaleźć w rozdziale poświęconym modelowi relacyjnemu (podrozdział 3.5, strona 217).
    - Niewątpliwą (i do tego ogromną) zaletą tego podejścia jest możliwość skorzystania z języka zapytań (zwykle różne dialekty SQL). Dzięki temu bardzo łatwo możemy wydobywać dane z bazy, stosując nierzadko bardzo wyrafinowane kryteria. Więcej na ten temat można znaleźć np. w [Bana04].
    - W tej chwili na rynku istnieje wiele różnych rozwiązań mających w nazwie (relacyjna) baza danych. W związku z tym każdy może znaleźć coś co będzie mu odpowiadało: poczynając od darmowego mySQL (http://www.mysql.com/), przez różne produkty Oracle (http://www.oracle.com/index.html), Micorosft (http://msdn.microsoft.com/sql/), a kończąc na dużych systemach IBM (http://www.ibm.com/software/ /data/db2/).

[11] Zgodnie z tym, co twierdzą producenci baz danych, większość dostępnych rozwiązań jest wyposażona w różnego rodzaju konstrukcje obiektowe, np. dziedziczenie. Jednakże, w większości przypadków, są one mało wykorzystywane.

- Korzystając z systemu zarządzania bazą danych, zwykle trzeba się liczyć z dość znaczącym zapotrzebowaniem na zasoby: pamięć RAM, wydajność procesora, wielkość pliku roboczego. Oprócz tego trzeba też wykazać się wiedzą dotyczącą konfiguracji i administrowania serwerem. Z tego względu, szczególnie w przypadku niewielkich projektów warto się zastanowić, czy baza danych jest na pewno optymalnym rozwiązaniem.

- Kolejną zaletą baz danych, oprócz języka zapytań, jest niewątpliwie szybkość działania oraz bezpieczeństwo danych. Jest to szczególnie istotne, gdy tych danych jest dużo i są cenne. Wtedy możemy wykorzystać różne sposoby przyspieszające wykonywanie zapytań, np. indeksowanie. Niestety, prawidłowe skonfigurowanie serwera baz danych nie jest łatwym zadaniem i może wymagać wiedzy, która nie jest typowa dla programistów czy projektantów.

- Wykorzystanie gotowych bibliotek. Innym sposobem uzyskania trwałości danych jest skorzystanie z czyjejś pracy. Istnieje dość dużo bibliotek, które ułatwiają pracę z danymi (nie tylko trwałość). Często, pod warstwą pośredniczącą (którą jest właśnie biblioteka) znajduje się system zarządzania bazą danych. Do najpopularniejszych *framework'ów* można zaliczyć:

  - Hibernate (http://www.hibernate.org/)

  - Java Persistence API (https://glassfish.dev.java.net/)

  - Java Data Objects (http://www.jpox.org/).

3.1.7 Klasa ObjectPlus

Przedstawione sposoby implementacji zarządzania ekstensją będą (prawie) takie same dla każdej biznesowej klasy w systemie. Zakładając implementację w ramach tej samej klasy biznesowej, w każdej z nich musimy umieścić prawie takie same elementy:

- kontener przechowujący referencję do jej wystąpień,

- metody ułatwiające zarządzanie (dodanie, usunięcie itp.).

Czy da się to jakoś zunifikować, abyśmy nie musieli pisać wiele razy (prawie) tego samego kodu?

Na szczęście odpowiedź na powyższe pytanie jest twierdząca. Wykorzystamy dziedziczenie istniejące w języku Java (tak, wiem, że jeszcze go nie omawialiśmy od strony implementacji, ale na razie wystarczą nam informacje z części teoretycznej – podrozdział 2.4.4 na stronie 47).

Stworzymy klasę, z której będą dziedziczyć wszystkie biznesowe klasy w naszej aplikacji. Nazwijmy ją np. `ObjectPlus`[12] i wyposażmy w:

- trwałość,
- zarządzanie ekstensją,
- być może jeszcze jakieś inne elementy.

Zastosujemy pierwsze z omawianych podejść do implementacji ekstensji, czyli w ramach tej samej klasy. A zatem musimy stworzyć kontener będący atrybutem klasowym (aby wszystkie obiekty danej klasy miały do niego dostęp). Czyli na pierwszy rzut oka wygląda to na identyczne rozwiązanie jak dla pojedynczej klasy. Czy na pewno? Zastanówmy się: tworzymy obiekt klasy `Film` dziedziczącej z klasy `ObjectPlus` i dzięki specjalnemu odwołaniu w konstruktorze dodajemy go do ekstensji (która jest przechowywana w atrybucie klasowym klasy `ObjectPlus`). Następnie tworzymy obiekt klasy `Aktor`, która też dziedziczy z klasy `ObjectPlus`. Również dzięki specjalnemu konstruktorowi obiekt dodawany jest do ekstensji przechowywanej w klasie `ObjectPlus`. W efekcie mamy dwa obiekty, klasy `Aktor` oraz `Film`, które znajdują się w tej samej ekstensji (w klasie `ObjectPlus`). Raczej nie o to nam chodziło. Czy ktoś ma pomysł, jak temu zaradzić?

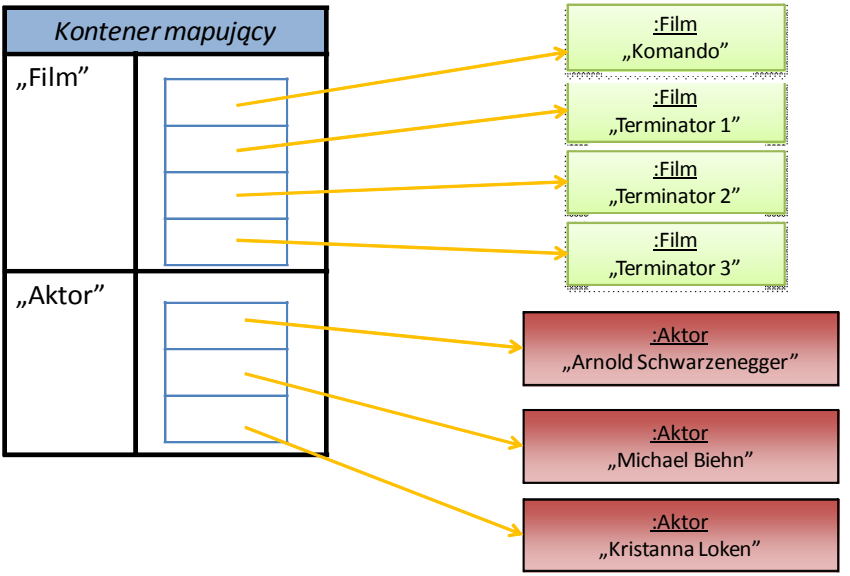
Ponieważ wszystkie biznesowe klasy dziedziczą z jednej nadklasy (`ObjectPlus`), nie możemy zastosować zwykłego kontenera przechowującego referencje. Użyjemy kontenera mapującego przechowującego klucze i wartości:

[12] Jako że wszystkie klasy w języku Java, te dostarczane przez producenta i te stworzone przez programistę dziedziczą z klasy `Object`, to nasza ulepszona klasa może nazywać się `ObjectPlus`. W przypadku MS C# jest zresztą podobnie – również istnieje tam jedna wspólna nadklasa.

- Kluczem będzie nazwa konkretnej biznesowej klasy, np. Aktor lub Film,

- Wartością będzie kontener zawierający referencje do jej wystąpień (właściwa ekstensja).

Innymi słowy, ten nowy kontener będzie zawierał wiele ekstensji, a nie jedną ekstensję. Rysunek 3-71 pokazuje schematycznie zawartość kontenera. Widzimy, że aktualnie znajdują się tam informacje o dwóch ekstensjach: filmów oraz aktorów. Ekstensja aktorów zapamiętana w dedykowanej kolekcji przechowuje referencje do trzech obiektów. Ekstensja filmów zapamiętana w innej kolekcji przechowuje referencje do czterech filmów.



3-71 Wykorzystanie kontenera mapującego do przechowania wielu ekstensji

Spójrzmy na ciekawsze miejsca implementacji klasy `ObjectPlus` pokazane na listingu 3-17:

- (1). Prywatny atrybut klasowy będący jednym z kontenerów mapujących języka Java. Jak już wspomnieliśmy, przechowuje on klucze (nazwy klas konkretnych ekstensji) oraz wartości zawierające kolekcje z referencjami do instancji danej klasy.

- (2). Konstruktor klasy `ObjectPlus`. Zawiera kod, który w odpowiedni i do tego automatyczny sposób dodaje obiekt do określonej ekstensji.

```
    public class ObjectPlus implements Serializable {
(1)         private static Hashtable ekstensje = new Hashtable();

(2)         public ObjectPlus() {
                Vector ekstensja = null;
(3)             Class klasa = this.getClass();

(4)             if(ekstensje.containsKey(klasa)) {
                    // Ekstensja tej klasy istnieje w kolekcji
                    ekstensji
(5)                 ekstensja = (Vector) ekstensje.get(klasa);
                }
                else {
                    // Ekstensji tej klasy jeszcze nie ma ->
                    dodaj ją
(6)                 ekstensja = new Vector();
                    ekstensje.put(klasa, ekstensja);
                }

(7)             ekstensja.add(this);
            }
    }
```

3-17 Implementacja ekstensji klasy przy pomocy `ObjectPlus`

- (3). Dzięki technice zwanej refleksją[13] uzyskujemy informację na temat przynależności klasowej (jakkolwiek dziwnie to brzmi) obiektu, który jest konstruowany. Zwrócona wartość, będąca instancją klasy `Class` będzie wykorzystana jako klucz w naszym `Hashtable`'u. Moglibyśmy wykorzystać jedną z jej metod do ustalenia nazwy klasy, ale tak będzie bardziej wydajnie.

- (4). Sprawdzamy, czy kontener mapujący zawiera już klucz opisujący naszą klasę.

- (5). Jeżeli tak, to na podstawie klucza odzyskujemy wartość, czyli kolekcję zawierającą ekstensję.

[13] Refleksja jest to technologia, która pozwala na uzyskanie, w czasie działania programu, informacji dotyczącej jego budowy, np. przynależność do klasy dla konkretnego obiektu, lista atrybutów, metod itp. Technologia ta jest obsługiwana również przez język MS C#. Nie występuje w podstawowej wersji języka C++, związane jest to między innymi z wydajnością.

- (6). Jeżeli nie, to tworzymy nową pustą kolekcję, która będzie zawierała instancje i dodajemy ją do głównego kontenera mapującego.

- (7). Do ekstensji dodajemy informację o nowej instancji, która właśnie jest tworzona. Warto zwrócić uwagę, że w tym miejscu mamy zawsze prawidłową ekstensję – bo albo ją odzyskaliśmy na podstawie klucza, albo utworzyliśmy nową.

Na pierwszy rzut oka cała ta powyższa technika może się wydawać trochę zagmatwana, ale jestem pewien, że po przeanalizowaniu (być może kilkakrotnym) da się to zrozumieć.

Teraz będzie ta łatwiejsza część – jak możemy z tego korzystać. Przykładowy kod jest pokazany na listingu 3-18. Tak naprawdę warto zwrócić uwagę tylko na dwa elementy, reszta to zwykłe biznesowe zapisy:

- (1). Aby móc używać naszej nowej funkcjonalności, musimy dziedziczyć z klasy `ObjectPlus`.

- (2). W celu automatycznego dodawania do ekstensji należy wywołać konstruktor z nadklasy. Później można umieścić zwykły kod wymagany przez uwarunkowania biznesowe.

```
(1)   public class Film2 extends ObjectPlus implements Serializable {
              private String tytul;
              private float cena;
              private Date dataDodania;

              /**
               * Konstruktor.
               */
              public Film2(String tytul, Date dataDodania, float cena) {
                     // Wywołaj konstruktor z nadklasy
(2)                  super();

                     this.tytul = tytul;
                     this.dataDodania= dataDodania;
                     this.cena = cena;
              }

              // Dalsza implementacja czesci biznesowej
      }
```

3-18 Wykorzystanie klasy `ObjectPlus`

Jak widać, wykorzystywanie tak utworzonej funkcjonalności jest banalnie proste i sprowadza się tylko do dziedziczenia z klasy `ObjectPlus` oraz umieszczenia wywołania konstruktora z nadklasy. Właściwie można z tego korzystać nawet nie wiedząc, jakie „czary" są wykonywane przez klasę `ObjectPlus`. Oczywiście zawsze jest lepiej rozumieć, jak mniej więcej działają nasze biblioteki – dzięki temu możemy je lepiej wykorzystywać.

Jak wcześniej sygnalizowaliśmy, rozszerzymy nasza klasę również o utrwalanie danych. Wykonanie tego jest już bardzo proste – może spróbujesz sam, drogi Czytelniku? a oto jedno z możliwych rozwiązań – listing 3-19.

```
public class ObjectPlus implements Serializable {
        private static Hashtable ekstensje = new Hashtable();

        // ...

        public static void zapiszEkstensje(ObjectOutputStream
        stream) throws IOException {
                stream.writeObject(ekstensje);
        }

        public static void odczytajEkstensje(ObjectInputStream
        stream) throws IOException {
                ekstensje = (Hashtable) stream.readObject();
        }

        // ...
}
```

3-19 Rozszerzenie klasy ObjectPlus o utrwalanie danych

Jak widać, dodaliśmy dwie metody korzystające z serializacji. Zamiast pracowicie zapisywać element po elemencie, po prostu zapamiętujemy cały kontener mapujący. Biblioteki odpowiedzialne za serializację zadbają o właściwe zapisanie całego grafu obiektów (pisaliśmy o tym w podrozdziale 3.1.6.2 na stronie 119).

Warto również wyposażyć naszą klasę w podstawowe metody klasowe, np. wyświetlanie ekstensji (listing 3-20).

Ważniejsze miejsca tej implementacji:

- (1). Jako parametr metody podajemy instancję klasy `Class`, której używamy do identyfikowania przynależności do klasy.

- (2). Sprawdzamy, czy istnieje podany klucz – czyli informacje o ekstensji.

- (3). Jeżeli tak, to zwracamy wartość dla tego klucza, czyli kontener zawierający ekstensję tej konkretnej klasy.

- (4). Jeżeli klucz nie istnieje, to znaczy, że nie mamy informacji o podanej ekstensji. W tej sytuacji rzucamy wyjątek. Można to rozwiązać inaczej, np. wyświetlić komunikat czy (po cichu) zakończyć działanie metody.

- (5). Wyświetlamy nazwę klasy.

- (6). A następnie iterujemy po kolekcji zawierającej ekstensję i wyświetlamy informację o każdym z obiektów. Tak naprawdę, w takiej sytuacji wywoływana jest niejawnie metodą `toSring()` pochodząca z konkretnego obiektu.

```
public class ObjectPlus implements Serializable {
        // ...
(1)     public static void pokazEkstensje(Class klasa) throws
        Exception {
                Vector ekstensja = null;
(2)             if(ekstensje.containsKey(klasa)) {
                        // Ekstensja tej klasy istnieje w kolekcji
                        ekstensji
(3)                     ekstensja = (Vector) ekstensje.get(klasa);
                }
                else {
(4)                     throw new Exception(„Nieznana klasa " +
                        klasa);
                }
(5)             System.out.println("Ekstensja klasy: " +
                klasa.getSimpleName());
(6)             for(Object obiekt : ekstensja) {
                        System.out.println(obiekt);
                }
        }
        // ...
}
```

3-20 Realizacja wyświetlania ekstensji w ramach klasy `ObjectPlus`

No i pozostało nam już tylko pokazać, jak używać metody wyświetlającej ekstensję (listing 3-21) oraz zademonstrować przykładowy efekt jej działania (konsola 3-3).

```
public class Film2 extends ObjectPlus implements Serializable {
        // ...
```

```
        public static void pokazEkstensje() throws Exception {
                ObjectPlus.pokazEkstensje(Film2.class);
        }
}
```

3-21 Wykorzystanie metody wyświetlającej ekstensję

```
Ekstensja klasy: Film2
Film2: Terminator 1, id: mt.mas.Film2@199f91c
Film2: Terminator 2, id: mt.mas.Film2@1b1aa65
```

3-3 Efekt działania metody wyświetlającej ekstensję

Jako krótkie podsumowanie implementacji klasy oraz zagadnień z tym związanych, można napisać, że część pojęć z obiektowości występuje w popularnych językach programowania. Niestety, niektóre z nich istnieją w niepełnym zakresie lub nie ma ich w ogóle. W większości przypadków nieistniejące konstrukcje można zaimplementować samodzielnie na kilka różnych sposobów lub obsłużyć, korzystając z gotowych bibliotek. Całą zaimplementowaną dodatkową funkcjonalność związaną m.in. z zarządzaniem ekstensją klasy warto zgromadzić w specjalnej nadklasie (`ObjectPlus`).

3.2 Asocjacje

Asocjacje od strony teoretycznej były omówione w podrozdziale 2.4.3 (strona 35). Teraz zajmiemy się odniesieniem tych informacji do popularnych języków programowania. Już na wstępie możemy stwierdzić, że w Java, MS C#, C++ asocjacje nie występują bezpośrednio. W związku z tym, nauczeni doświadczeniem, postaramy się je jakoś zaimplementować.

W celu implementacji asocjacji możemy zastosować jedno z dwóch podejść. Zasadnicza różnica pomiędzy nimi polega na sposobie przechowywania informacji o powiązaniu obiektów:

- możemy wykorzystać własne identyfikatory, np. liczbowe,
- lub natywne referencje języka programowania (Java, C#) lub wskaźniki (C++).

Od razu nasuwa się pytanie, które podejście jest lepsze? z którego warto korzystać? Tradycyjnie nie ma jednej uniwersalnej odpowiedzi. W zależności od potrzeb jedno lub drugie rozwiązanie może wydawać się bardziej użyteczne. Chociaż od razu trzeba zauważyć, że w większości przypadków

wykorzystamy natywne referencje języka programowania (lub wskaźniki dla C++). Dlaczego? Na to pytanie dadzą odpowiedź następne rozdziały (przynajmniej taką mam nadzieję).

3.2.1 Implementacja asocjacji za pomocą identyfikatorów

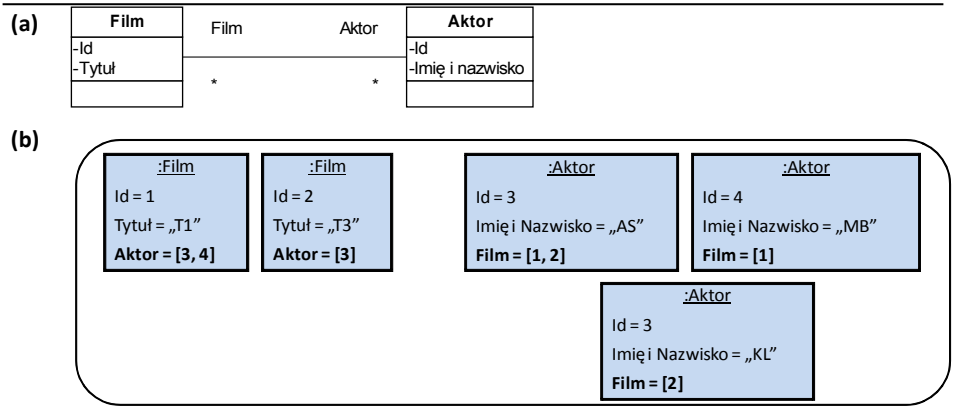
Zanim zastanowimy się, jakie mogą być ewentualne korzyści z zastosowania tego sposobu, zobaczmy, na czym on polega. Ogólną zasadę można streścić w sposób następujący:

- Do każdej klasy biznesowej dodajemy atrybut będący identyfikatorem, np. liczbę `int`. Liczba ta pozwoli na jednoznaczną identyfikację danego obiektu. W zależności od potrzeb biznesowych unikalność może być w ramach pojedynczej ekstensji lub wszystkich obiektów w systemie.

- Informacje o powiązanych obiektach przechowujemy pamiętając ich identyfikatory. W zależności od liczności sposób przechowywania będzie trochę inny.

- Aby móc wykorzystywać asocjacje dwukierunkowe (a mam nadzieję, że we wcześniejszych rozdziałach tej książki przekonałem Ciebie, drogi Czytelniku o korzyściach z tego płynących), musimy wykorzystywać pary identyfikatorów.

Spójrzmy na rysunek 3-72. Część (a) przedstawia prosty diagram klas umożliwiający zapamiętywanie informacji o filmach i grających w nich aktorach (lub innymi słowy: o aktorach i filmach, w których grali). Część (b) zawiera fragment pewnej przykładowej sytuacji, którą zapamiętaliśmy w systemie, korzystając z identyfikatorów. Dlatego w „obiektach" występują atrybuty „Aktor" oraz „Film", których nie ma na diagramie klas. Jest tak dlatego, że są one odpowiednikami ról asocjacji. Jak widać, film „Terminator 1" ma identyfikator równy 1, a jego trzecia część „2". Zwróćmy uwagę na atrybut o nazwie „Aktor". Zawiera on liczby będące identyfikatorami obiektów opisujących aktorów grających w danym filmie, np. „Id=3" oznacza aktora „AS" (ci, którzy oglądali serię „Terminator" na pewno wiedzą, o kogo chodzi). I w drugą stronę: aktor „AS" ma atrybut „Film", który przechowuje identyfikatory filmów, w których grał konkretny aktor, np. dla Id = 3 będzie to „Film = [1, 2]".



3-72 Wykorzystanie identyfikatorów do implementacji asocjacji

Jak można się zorientować, sposób jest dość prosty, ale zarazem skuteczny. Umożliwia bowiem zapamiętywanie dowolnych zależności zachodzących pomiędzy obiektami.

W jaki sposób można zaimplementować takie podejście w języku Java? Przykład można zobaczyć na listingu 3-22.

```
(1)   public class Aktor {
(2)           private int id;
(3)           public String imieNazwisko;// public dla uproszczenia
(4)           public int[] film;
(5)           private static ArrayList<Aktor> ekstensja = new
              ArrayList<Aktor>();

(6)           public Aktor(int id, String imieNazwisko, int[] filmId) {
                      // Dodaj do ekstensji
(7)                   ekstensja.add(this);

(8)                   this.id = id;
                      this.imieNazwisko = imieNazwisko;
                      film = filmId;
              }

(9)           public static Aktor znajdzAktora(int id) throws Exception
(10)          {
                      for(Aktor aktor : ekstensja) {
                              if(aktor.id == id) {
                                      return aktor;
                              }
(11)                  }
                      throw new Exception("Nie znaleziono aktora o id =
                      " + id);
              }
      }
```

```
(12)
      public class Film {
              public int id;
              public String tytul; // public dla uproszczenia
              public int[] aktor;
              private static ArrayList<Film> ekstensja = new
              ArrayList<Film>();

              public Film(int id, String tytul, int[] aktorId) {
                      // Dodaj do ekstensji
                      ekstensja.add(this);

                      this.id = id;
                      this.tytul = tytul;
                      aktor = aktorId;
              }

              public static Film znajdzFilm(int id) throws Exception {
                      for(Film film : ekstensja) {
                              if(film.id == id) {
                                      return film;
                              }
                      }
                      throw new Exception("Nie znaleziono filmu o id =
                      " + id);
              }
      }
```

3-22 Implementacja asocjacji za pomocą identyfikatorów

Ważniejsze miejsca wymagające komentarza:

- (1). Definicja klasy biznesowej Aktor.
- (2). Prywatny atrybut typu `int` pełniący rolę identyfikatora.
- (3). Zwykły atrybut biznesowy.
- (4). Tablica liczb typu `int` przechowująca identyfikatory filmów, w których grał aktor. Warto zwrócić uwagę, że pojedyncza liczba nie wystarczy, ponieważ liczności asocjacji informują nas o możliwości wielu powiązań. Lepiej użyć tu kolekcji (zamiast tablicy), ale chcieliśmy stworzyć najprostszy możliwy przykład.
- (5). Atrybut klasowy przechowujący ekstensję. Tego typu element jest niezbędny w naszym podejściu. Dlaczego? Ponieważ wszelkie zależności opisujemy za pomocą identyfikatorów, musimy mieć spo-

sób otrzymania odpowiadającego mu obiektu. W tym celu przeszukujemy ekstensję dopóki nie znajdziemy obiektu z podanym identyfikatorem. Czy nie dało by się tego zrobić lepiej? Oczywiście, że tak – możemy wykorzystać konstrukcję mapującą, w której kluczem jest id, a wartością referencja do obiektu. Nie zrobiliśmy tak, aby nie komplikować przykładu.

- (6). Konstruktor klasy.

- (7). Wewnątrz konstruktora dodajemy obiekt do ekstensji (analogicznie jak robiliśmy w podrozdziale 3.1.3.1 na stronie 101.

- (8). Dalsza, biznesowa część konstruktora.

- (9). Metoda, która na podstawie identyfikatora (liczba) zwraca referencję do odpowiadającego mu obiektu – patrz uwagi do pkt (5).

- (10). Iterujemy po wszystkich obiektach w ekstensji. Gdy znajdziemy instancję z podanym identyfikatorem, to ją zwracamy.

- (11). Jeżeli przejrzymy całą ekstensję i nie znajdziemy obiektu z podanym identyfikatorem, to rzucamy wyjątek. Zamiast wyjątku można zastosować coś mniej inwazyjnego, np. `null` lub komunikat na konsolę.

- (12). Klasa Film – analogiczna do klasy Aktor.

Kolejny listing (3-23) pokazuje sposób wykorzystania tak zaimplementowanych asocjacji. Warto zwrócić uwagę na:

- (1). Tworzenie instancji klasy `Film`. Podajemy identyfikator, tytuł oraz tablicę liczb typu `int` określających identyfikatory aktorów, którzy w nim grają.

- (2). Powoływanie do życia obiektów opisujących aktorów. Analogicznie jak w przypadku filmów podajemy: identyfikator, inicjały aktora oraz tablicę liczb zawierającą odniesienia do filmów. Zauważ, drogi Czytelniku, że jeżeli w filmie „T1” wystąpił aktor „3”, to przy aktorze „3” musi być też informacja o filmie „1”. Dzięki temu nasze powiązania są dwukierunkowe.

(3). Wyświetlanie informacji o filmach. W pętli wywołujemy metodę `znajdzAktora(...)`, która na podstawie id zwraca referencję do obiektu go opisującego.

```
(1)  Film film1 = new Film(1, "T1", new int[]{3, 4});
     Film film2 = new Film(2, "T3", new int[]{3});

(2)  Aktor aktor1 = new Aktor(3, "AS", new int[]{1, 2});
     Aktor aktor2 = new Aktor(4, "MB", new int[]{1});
     Aktor aktor3 = new Aktor(5, "KL", new int[]{3});

     // Wyswietl info o filmie1
     System.out.println(film1.tytul);
(3)  for(int i = 0; i < film1.aktor.length; i++) {
            System.out.println("   " +
            Aktor.znajdzAktora(film1.aktor[i]).imieNazwisko);
     }

     // Wyswietl info o aktor1
     System.out.println(aktor1.imieNazwisko);
     for(int i = 0; i < aktor1.film.length; i++) {
            System.out.println("   " +
            Film.znajdzFilm(aktor1.film[i]).tytul);
     }
```

3-23 Przykład ilustrujący wykorzystanie asocjacji z identyfikatorami

W efekcie otrzymujemy stan konsoli przedstawiony na rysunku 3-4 Konsola

```
T1
   AS
   MB
AS
   T1
   T3
```

3-4 Konsola ilustrująca wykorzystanie asocjacji

Jak możemy podsumować zaprezentowane rozwiązanie? Zacznijmy od wad, a właściwie jednej – zasadniczej. Jest nią konieczność wyszukiwania obiektu na podstawie jego numeru identyfikacyjnego, co może prowadzić do problemów z wydajnością. Wydajność wyszukiwania można zdecydowanie poprawić, używając kontenerów mapujących zamiast zwykłych. Przykładowo-

wy kod ilustrujący tę modyfikację jest pokazany na listingu 3-24. Jak zwykle spójrzmy na fragmenty kodu:

- (1). Atrybut klasowy będący mapą, gdzie kluczem jest `Integer` (zwróćmy uwagę, że zamiast typu prostego `int` używamy klasy `Integer`. Dlaczego?), a wartością referencja do odpowiadającego mu obiektu.

- (2). Widzimy, że w konstruktorze też dodajemy obiekt do ekstensji, ale ze względu na specyfikę mapy robimy to inaczej niż wcześniej.

- (3). Dzięki wykorzystaniu metody wbudowanej w mapę nasze wyszukiwanie też uległo uproszczeniu.

```
     public class Film {
(1)          private static Map<Integer, Film> ekstensja = new
             TreeMap<Integer, Film>();
             // [...]
             public Film(int id, String tytul, int[] aktorId) {
                    // Dodaj do ekstensji
(2)                 ekstensja.put(new Integer(id), this);
             }
             // [...]
(3)          public static Film znajdzFilm(int id) throws Exception {
                    return ekstensja.get(new Integer(id));
             }
     }
```

3-24 Wykorzystanie pojemnika mapującego w celu przyspieszenia wyszukiwania obiektów

Mimo zastosowania struktury mapującej i tak musimy wyszukiwać w ekstensji. Wyszukiwanie w mapie jest dużo szybsze (myślę, że kilkaset, a może nawet kilka tysięcy razy dla dużych danych) niż iteracyjne przeglądanie całej ekstensji, ale i tak troszkę to może potrwać.

Przejdźmy teraz do zalet, a właściwe też jednej. Można ją streścić w jednym zdaniu: uniezależnienie poszczególnych obiektów od siebie. Dzięki temu, że nie używamy referencji języka Java, maszyna wirtualna nic nie wie o naszych powiązaniach. Jest to bardzo ważne, gdy np. chcemy odczytać jeden obiekt z bazy danych lub przesłać przez sieć. W takim przypadku sytuacja może wyglądać tak:

- Tworzony jest obiekt języka Java na podstawie zawartości BD, ale nie są tworzone obiekty z nim powiązane,

- Całość można tak zaprojektować, aby dopiero przy próbie dostępu do obiektu wskazywanego przez identyfikator pobrać go z BD czy lokalizacji sieciowej.

Dzięki temu nie musimy rekonstruować od razu całego grafu obiektów, który nie musi być nam potrzebny. Gdybyśmy zamiast identyfikatorów użyli natywnych referencji języka Java, to w przypadku np. przesyłania przez sieć obiekt mógłby być serializowany. Jak pamiętamy (a pamiętamy?) z podrozdziału 3.1.6.2 (strona 119), odpowiednie biblioteki dbają, aby cały obiekt był w prawidłowym stanie. Czyli chcąc przesłać jeden obiekt, mogłoby się zdarzyć, że prześlemy wszystkie (gdy są wzajemnie powiązane). W takiej sytuacji wykorzystywanie identyfikatorów jest bardzo użyteczne.

Jako identyfikator możemy zastosować własną klasę, a nie tylko prosty typ int. Dzięki temu, przesłaniając metody `public boolean equals(Object obj)` oraz `public int hashCode()` (metody te zwykle należy przesłaniać parami) możemy uzyskać bardzo dużą elastyczność.

3.2.2 Implementacja asocjacji za pomocą natywnych referencji

Teraz zajmijmy się drugim podejściem do przechowywania informacji o powiązanych obiektach. Rozwiązanie to jest chyba częściej spotykane, a można je streścić w poniższych punktach:

- W celu „pokazywania" na powiązane obiekty wykorzystujemy natywne referencje języka (Java, C#) lub wskaźniki (C++). Dzięki temu nie musimy odszukiwać obiektów.

- Mając referencję (lub wskaźnik) wskazującą na obiekt, mamy do niego natychmiastowy dostęp (szybciej już się nie da). Po prostu maszyna wirtualna czyta spod określonego miejsca w pamięci, odpowiednio interpretując zawarte tam dane.

- Podobnie jak w poprzednim przypadku i tutaj występuje konieczność tworzenia par referencji (jeżeli chcemy nawigować w dwie strony – a przeważnie chcemy).

W zależności od liczności asocjacji wykorzystujemy różne konstrukcje (omawiamy je dla porządku, bo w praktyce często się wykorzystuje „* - *", która umożliwia zapamiętanie dowolnej kombinacji):

- Liczność 1 do 1. Pojedyncza referencja z każdej strony. Przykładowy kod źródłowy na listingu 3-25. Jak widać, w każdej z klas mamy pojedyncze referencje pokazujące na powiązany obiekt. Dla klasy `Aktor` jest to referencja `film` typu `Film`, a dla klasy `Film` odwrotnie: referencja `aktor` typu `Aktor`.

```
public class Aktor {
        public String imieNazwisko;// public dla uproszczenia
        public Film film; // impl. Asocjacji, licznosc 1

        public Aktor(String imieNazwisko, Film film) {
                this.imieNazwisko = imieNazwisko;
                this.film = film;
        }
}
public class Film {
        public String tytul; // public dla uproszczenia
        public Aktor aktor; // impl. Asocjacji, licznosc 1

        public Film(String tytul, Aktor aktor) {
                this.tytul = tytul;
                this.aktor = aktor;
        }
}
```

3-25 Implementacja asocjacji dla liczności 1 - 1

- Liczność 1 do *. Pojedyncza referencja oraz kontener. Przykładowy kod źródłowy na listingu 3-26. W klasie `Aktor` jest sytuacja identyczna jak w poprzednim przypadku. Natomiast w klasie `Film` mamy kontener przechowujący referencje do typu `Aktor` (bo musimy mieć możliwość pokazywania na wiele instancji – liczność „*"). Tutaj akurat wykorzystaliśmy parametryzowany `ArrayList`, ale w zależności od potrzeb może to być też inny pojemnik.

```
public class Aktor {
        public String imieNazwisko;// public dla uproszczenia
        public Film film; // impl. Asocjacji, licznosc 1

        public Aktor(String imieNazwisko, Film film) {
                this.imieNazwisko = imieNazwisko;
                this.film = film;
        }
}
public class Film {
        public String tytul; // public dla uproszczenia
        public ArrayList<Aktor> aktor; // impl. Asocjacji,
        licznosc *
```

```
        public Film(String tytul, Aktor aktor) {
                this.tytul = tytul;
                this.aktor = aktor;
        }
}
```

3-26 Implementacja asocjacji dla liczności 1 - *

- Liczności * do *. Dwa kontenery po każdej ze stron. Przykładowy kod źródłowy na listingu 3-27. W obydwu klasach występują pojemniki – podobnie jak w poprzednim przykładzie dla liczności „*”.

```
public class Aktor {
        public String imieNazwisko;// public dla uproszczenia
        public ArrayList<Film> film; // impl. Asocjacji, licznosc
        *

        public Aktor(String imieNazwisko) {
                this.imieNazwisko = imieNazwisko;
        }
}
public class Film {
        public String tytul; // public dla uproszczenia
        public ArrayList<Aktor> aktor; // impl. Asocjacji,
        licznosc *

        public Film(String tytul) {
                this.tytul = tytul;
        }
}
```

3-27 Implementacja asocjacji dla liczności * - *

Dotychczas zaproponowane podejście wymagało ręcznego dodawania informacji o powiązaniu zwrotnym. Czyli aby połączyć dwa obiekty (np. aktor i film) trzeba oddzielnie dodać referencje pokazująca z filmu na aktora oraz z aktora na film. Warto to jakoś zautomatyzować, tak abyśmy nie musieli robić tego ręcznie. Stworzymy metodę, która doda informacje o powiązaniu:

- w klasie głównej,
- w klasie z nią powiązanej.

Całość musi być tak zaprojektowana, aby nie dochodziło do zapętlenia (więcej na ten temat będzie powiedziane dalej).

```
(1)   public class Aktor {
              public String imieNazwisko;// public dla uproszczenia
(2)           private ArrayList<Film> film = new ArrayList<Film>(); //
              impl. Asocjacji, licznosc *
              public Aktor(String imieNazwisko) {
                     this.imieNazwisko = imieNazwisko;
              }
(3)           public void dodajFilm(Film nowyFilm) {
                     // Sprawdz czy nie mamy juz informacji o tym
(4)                  filmie
(5)                  if(!film.contains(nowyFilm)) {
                            film.add(nowyFilm);

(6)                         // Dodaj informacje zwrotna
                            nowyFilm.dodajAktor(this);
                     }
(7)           }
(8)           public String toString() {
                     String info = "Aktor: " + imieNazwisko + "\n";
(9)                  // Dodaj info o tytulach jego filmow
                     for(Film f : film) {
                            info += "   " + f.tytul + "\n";
                     }
                     return info;
              }
      }

      public class Film {
              public String tytul; // public dla uproszczenia
              private ArrayList<Aktor> aktor = new
              ArrayList<Aktor>(); // impl. Asocjacji, licznosc *
              public Film(String tytul) {
                     this.tytul = tytul;
(10)          }
              public void dodajAktor(Aktor nowyAktor) {
                     // Sprawdz czy nie mamy juz takiej informacji
                     if(!aktor.contains(nowyAktor)) {
                            aktor.add(nowyAktor);
                            // Dodaj informacje zwrotna
                            nowyAktor.dodajFilm(this);
                     }
              }
              public String toString() {
                     String info = "Film: " + tytul + "\n";
                     // Dodaj info o tytulach jego filmow
                     for(Aktor a : aktor) {
                            info += "   " + a.imieNazwisko + "\n";
                     }

                     return info;
              }
```

```
}
```

3-28 Implementacja asocjacji z powiązaniem zwrotnym

Listing 3-28 zawiera kod umożliwiający automatyczne dodawanie informacji o powiązaniu zwrotnym. Innymi słowy, wystarczy np. do filmu dodać informację o aktorze, a powiązanie z aktora do filmu zostanie utworzone automatycznie. Ważniejsze miejsca programu:

- (1). Klasa opisująca aktora.
- (2). Prywatny kontener przechowujący informacje o filmach powiązanych z konkretnym aktorem. Ciekawostka – co by się stało, gdybyśmy zadeklarowali go jako `static`?
- (3). Metoda dodająca informacje o filmie dla konkretnego aktora. To właśnie w niej zaimplementowana jest „automatyka", o której wcześniej mówiliśmy.
- (4). Najpierw sprawdzamy, czy kolekcja filmów zawiera już referencję do podanego obiektu opisującego film. Innymi słowy, chcemy ustalić, czy jest już powiązanie pomiędzy tym aktorem a filmem. Po co to robimy? Wyobraźmy sobie, że nie sprawdzamy tego warunku. Co się wtedy dzieje?
    - Dodajemy informację o tym filmie (5),
    - Następnie wołamy metodę z obiektu po drugiej stronie (6), która doda informację zwrotną (10). Jej kod jest prawie identyczny z metodą, którą właśnie analizujemy. Czyli, w pewnym momencie, zostanie wywołana metoda z obiektu po drugiej stronie; z wnętrza tej metoda znowu zostanie wywołana metoda po drugiej stronie itd. – mamy zapętlenie, którego efektem jest „zawieszenie" się programu.
- Jak widać (mam nadzieję), sprawdzanie tego warunku działa jako bezpiecznik – przerywa łańcuch wzajemnych wywołań.

- (7). Przesłonięcie metody (więcej na ten temat będzie w podrozdziale 3.3.2 na stronie 176) odpowiedzialnej za zwrócenie tekstowego opisu obiektu. Co się tu dzieje? Otóż najpierw konstruujemy pojedynczy string zawierający imię i nazwisko aktora (8). Następnie dodajemy do niego informacje o filmach, w których grał. Robimy to, iterując po kolekcji filmów (9) i odczytując ich tytuły, które rozdzielamy znakiem końca linii.

W jaki sposób możemy używać kodu, który właśnie stworzyliśmy? Przykładowe zastosowanie zawiera listing 3-29. Ważniejsze miejsca:

- (1). Tworzymy obiekty opisujące filmy. Warto zwrócić uwagę, że na razie nie mamy żadnych informacji o powiązaniach z aktorami.

- (2). Tworzymy obiekty opisujące aktorów. Podobnie jak w powyższym przypadku, nie mamy żadnych powiązań.

- (3). Do filmów dodajemy aktorów. Równie dobrze można zrobić odwrotnie: aktorom przypisać filmy. Dzięki automatycznemu tworzeniu powiązań efekt będzie dokładnie taki sam.

- (4). Wyświetlamy informacje o filmach.

- (5). Wyświetlamy informacje o filmach.

```
      // Utworz nowe obiekty (na razie bez informacji o powiazaniach)
(1)   Film film1 = new Film("T1");
      Film film2 = new Film("T3");

(2)   Aktor aktor1 = new Aktor("AS");
      Aktor aktor2 = new Aktor("MB");
      Aktor aktor3 = new Aktor("KL");

      // Dodaj informacje o powiazaniach
(3)   film1.dodajAktor(aktor1);
      film1.dodajAktor(aktor2);
      film2.dodajAktor(aktor1);
      film2.dodajAktor(aktor3);

      // Wyswietl info o filmach
(4)   System.out.println(film1);
      System.out.println(film2);

      // Wyswietl info o aktorach
(5)   System.out.println(aktor1);
      System.out.println(aktor2);
```

```
System.out.println(aktor3);
```

3-29 Przykład ilustrujący wykorzystanie asocjacji z informacją zwrotną

Efekt działania programu można zaobserwować na konsoli 3-5 Konsola.

```
Film: T1
   AS
   MB

Film: T3
   AS
   KL

Aktor: AS
   T1
   T3

Aktor: MB
   T1

Aktor: KL
   T3
```

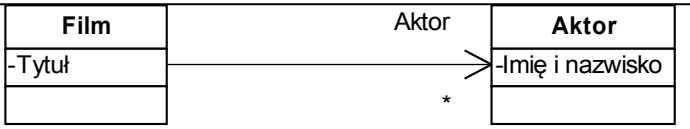
3-5 Konsola ilustrująca wykorzystanie automatycznego dodawania asocjacji zwrotnej

3.2.3 Implementacja różnych rodzajów asocjacji

Poniższe podrozdziały opisują sposób implementacji różnych rodzajów asocjacji oraz agregacji i kompozycji. Implementacja ta może być zrealizowana w oparciu o wykorzystanie identyfikatorów (podrozdział 3.2.1, strona 132) lub natywnych referencji (podrozdział 3.2.2, strona 138). Większość przykładów obejmuje to drugie podejście.

3.2.3.1 Asocjacja skierowana

Spójrzmy na rysunek 3-73. Zawiera on prosty diagram klas wykorzystujący asocjację skierowaną.



3-73 Prosty diagram klas ilustrujący wykorzystanie asocjacji skierowanej

Powyższy diagram oznacza, że dla:

- konkretnego filmu chcemy znać jego aktorów,
- konkretnego aktora nie chcemy znać jego filmów.

Oczywiście użyteczność biznesowa takiego modelowania może być dyskusyjna.

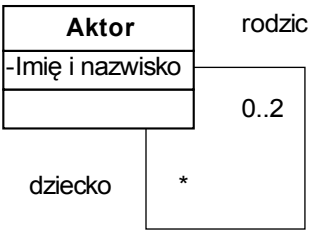
Implementacja jest analogiczna do poprzednich przypadków, z tym, że pamiętamy tylko informacje w jednej z klas:

- W klasie film jest odpowiedni kontener (pokazujący na filmy),
- W klasie aktor brak kontenera przechowującego informacje o filmach.

3.2.3.2 Asocjacja rekurencyjna

Diagram z rysunku 3-74 przedstawia asocjację rekurencyjną. Opisuje ona zależność pomiędzy aktorami oraz ich dziećmi. Jak pamiętamy z podrozdziału 2.4.3.4 (strona 42), w tego typu konstrukcji obowiązkowe są nazwy ról (tutaj: rodzic i dziecko). Zresztą i tak przydają się one na potrzeby implementacji.

Implementacja asocjacji rekurencyjnej jest dość prosta, ale wiąże się z pewnym trikiem, który nie zawsze jest oczywisty. Otóż w poprzednich przykładach mieliśmy po jednym kontenerze w każdej z połączonych klas. Tutaj będą dwa. Jak się nad tym zastanowić, to jest to logiczne: musimy pamiętać powiązania z punktu widzenia każdej z ról. Role są dwie, więc i pojemniki są też dwa.



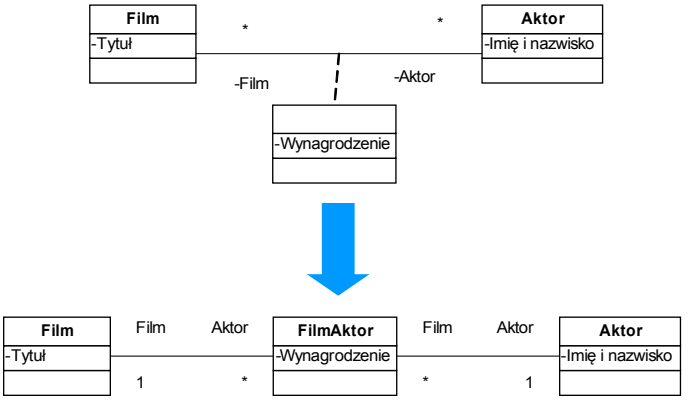
3-74 Przykładowy diagram klas pokazujący zastosowanie asocjacji rekurencyjnej (zwrotnej)

```
     public class Aktor {
            public String imieNazwisko;// public dla uproszczenia
(1)         private ArrayList<Film> film = new ArrayList<Film>(); //
            impl. Asocjacji, licznosc

(2)         private ArrayList<Aktor> rodzic = new ArrayList<Aktor>();
(3)         private ArrayList<Aktor> dziecko = new
            ArrayList<Aktor>();
            // [...]
     }
```

3-30 Implementacja asocjacji rekurencyjnej

Listing 3-30 przedstawia przykładową implementację. Zawiera kolekcję (1) przechowującą powiązania do filmów (niezwiązaną z asocjacją rekurencyjną) oraz dwa kontenery pamiętające referencje do dzieci oraz rodziców (2, 3). Przy czym trzeba jasno podkreślić, że w takim kształcie jak tu widać i dzieci, i rodzice muszą być aktorami (takie aktorskie rodziny).

3.2.3.3 *Asocjacja z atrybutem*



3-75 Przekształcenie asocjacji z atrybutem

Jednym z częściej wykorzystywanych rodzajów asocjacji jest asocjacja z atrybutem – czasami też zwana klasą asocjacji. Spójrzmy na przykład z rysunku 3-75. Dokładniejsze (teoretyczne) omówienie tego pojęcia znajduje się w podrozdziale 2.4.3.5 na stronie 43.

Podejście do implementacji klasy asocjacji polega na zamianie:

- jednej konstrukcji UML (asocjacja z atrybutem)
- na inną konstrukcję UML (asocjacja z klasą pośredniczącą).

W efekcie, dzięki zastąpieniu atrybutu asocjacji klasą pośredniczącą otrzymaliśmy dwie „zwykłe" asocjacje. Te „nowe" asocjacje implementujemy na jeden ze znanych, omówionych wcześniej sposobów. Niestety, po takim przekształceniu możemy oczekiwać drobnych problemów związanych z semantyką tej nowej klasy:

- jaką nazwę jej nadać? Jeżeli nie przychodzi nam do głowy żadna sensowna biznesowa nazwa, to możemy stworzyć nową jako zlepek nazw sąsiadujących klas. Niektórzy twierdzą, że tak się powinno zawsze robić – dzięki temu od razu widać, że jest to nowa klasa.
- Kolejnym problemem mogą być nazwy ról: „starych" oraz „nowych". Wcześniej mieliśmy dwie, dobrze zdefiniowane, biznesowe role. Po przekształceniu otrzymaliśmy dwie kolejne. Dobra wiadomość jest

taka, że zwykle nowe nazwy tworzymy przez powielenie starych. Oczywiście tak nie musi być.

Wadą takiego podejścia jest utrudniony dostęp do obiektów docelowych. Wcześniej z obiektu klasy Film poprzez rolę aktor dostawaliśmy się do instancji aktora. Po zmianie musimy „przejść" poprzez obiekt klasy pośredniczącej. Ewentualnie można zaimplementować jakieś metody pomocnicze, które po wywołaniu z klasy pośredniczącej zwrócą nam obiekt docelowy.

3.2.3.4 Asocjacja kwalifikowana

Asocjacja kwalifikowana (rysunek 3-76) jest często wykorzystywana w praktyce, chociaż czasami ci, co jej używają, nawet o tym nie wiedzą. Przypomnijmy, że polega na tym, iż dostęp do obiektu docelowego odbywa się na podstawie unikatowego kwalifikatora. Jej teoretyczny opis można znaleźć w podrozdziale 2.4.3.3 na stronie 40. W tym podrozdziale opowiemy jak sobie z nią radzić podczas implementacji.

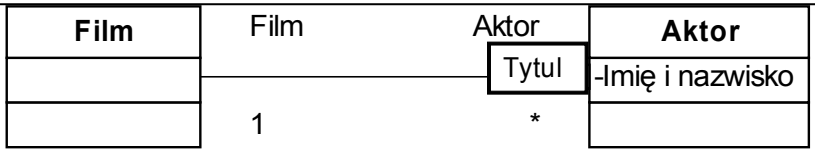
Najprostsza implementacja asocjacji kwalifikowanej (a właściwe jej funkcjonalności) może być wykonana w następujący sposób:

- Korzystamy z istniejącego podejścia dotyczącego realizacji asocjacji (bez różnicy, czy używamy identyfikatorów czy referencji),
- Dodajemy metodę, która na podstawie tytułu zwróci nam obiekt klasy Film.

Zauważmy, że takie podejście charakteryzuje się słabą wydajnością, ponieważ aby znaleźć określony tytuł, musimy „ręcznie" przeglądać filmy.

Lepsze rozwiązanie (i to jest zalecane) można scharakteryzować w sposób następujący:

- Nie korzystamy ze standardowego kontenera (takiego jak np. `ArrayList`),
- Stosujemy kontener mapujący, gdzie kluczem jest kwalifikator (np. tytuł), a wartością referencja do obiektu docelowego (np. obiektu opisującego film).
- Informacja zwrotna jest przechowywana w dotychczasowy sposób (np. kontener typu `ArrayList`).



3-76 Asocjacja kwalifikowana

Ponownie spójrzmy na rysunek 3-76 przedstawiający przykładową asocjację kwalifikowaną. Odpowiadającą jej implementację umieszczono na listingu 3-31 (jako kod klasy film można wykorzystać ten z listingu 3-28, strona 142). Miejsca wymagające komentarza:

- (1). Klasa przechowująca informacje o aktorze.
- (2). Mapa będąca głównym elementem implementacji asocjacji kwalifikowanej. Kluczem jest kwalifikator (w naszym przypadku tytuł filmu), a wartością referencja do obiektu docelowego (filmu).
- (3). Metoda tworząca połączenie w ramach asocjacji kwalifikowanej. Obsługuje również połączenie zwrotne.
- (4). Sprawdzamy, czy nie ma już informacji o filmie z takim tytułem (zauważ, drogi Czytelniku, że nie piszę o filmie, a tylko o jego tytule. W takim podejściu tytuły muszą być unikatowe.)
- (5). Jeżeli nie, to dodajemy nowy zapis do mapy (kluczem jest tytuł, a wartością referencja do obiektu filmu).
- (6). Następnie tworzymy połączenie zwrotne, wywołując odpowiednią metodę z klasy `Film`.
- (7). Metoda, która odszukuje obiekt docelowy (film) na podstawie kwalifikatora (tytuł filmu).
- (8). Jeżeli mapa nie zawiera podanego klucza, to znaczy, że nie mamy informacji o powiązaniu. W takiej sytuacji rzucamy wyjątek. Jak już wspominaliśmy, można to obsłużyć inaczej, np. zwracając `null`.

- (9). Korzystamy z metody mapy, która zwraca wartość na podstawie odpowiadającej jej wartości.

```
(1)  public class Aktor {
            // [...]

(2)         private TreeMap<String, Film> filmKwalif = new
            TreeMap<String, Film>();

(3)         public void dodajFilmKwalif(Film nowyFilm) {
                    // Sprawdz czy nie mamy juz informacji o tym filmie
(4)                 if(!filmKwalif.containsKey(nowyFilm.tytul)) {
(5)                         filmKwalif.put(nowyFilm.tytul, nowyFilm);
                            // Dodaj informację zwrotną
(6)                         nowyFilm.dodajAktor(this);
                    }
            }
(7)         public Film znajdzFilmKwalif(String tytul) throws
            Exception {
                    // Sprawdz czy nie mamy juz informacji o tym filmie
(8)                 if(!filmKwalif.containsKey(tytul)) {
                            throw new Exception("Nie odnaleziono filmu
                            o tytule: " + tytul);
                    }
(9)                 return filmKwalif.get(tytul);
            }
            // [...]
    }
```

3-31Implementacja asocjacji kwalifikowanej

I jeszcze pozostało nam zademonstrować sposób użycia tak opracowanego kodu. Odpowiedni przykład znajduje się na listingu 3-32. I tradycyjnie komentarz:

- (1), (2). Tworzymy obiekty opisujące filmy oraz aktorów (na razie bez informacji o powiązaniach).

- (3). Dodajemy informacje o powiązaniach, z uwzględnieniem kwalifikatora.

- (4). Wyświetlamy informacje o aktorach.

- (5). Odszukujemy film powiązany z konkretnym aktorem na podstawie jego tytułu.

```
// Utworz nowe obiekty (na razie bez informacji o powiazaniach)
```

```
(1)   Film film1 = new Film("T1");
      Film film2 = new Film("T3");

(2)   Aktor aktor1 = new Aktor("AS");
      Aktor aktor2 = new Aktor("MB");
      Aktor aktor3 = new Aktor("KL");

      // Dodaj informacje o powiazaniach
(3)   aktor1.dodajFilmKwalif(film1);
      aktor1.dodajFilmKwalif(film2);
      aktor2.dodajFilmKwalif(film1);
      aktor3.dodajFilmKwalif(film2);

      // Wyswietl info o aktorach
(4)   System.out.println(aktor1);
      System.out.println(aktor2);
      System.out.println(aktor3);

      // Pobierz info o filmie "T1" dla aktora aktor1
(5)   Film f = aktor1.znajdzFilmKwalif("T1");
      System.out.println(f);
```

3-32 Przykład użycia asocjacji kwalifikowanej

Efekty działania powyższego programu przedstawia konsola 3-6.

```
Aktor: AS
   T1
   T3

Aktor: MB
   T1

Aktor: KL
   T3

Film: T1
   AS
   MB
```

3-6 Efekty działania programu testującego asocjację kwalifikowaną

3.2.3.5 Asocjacja n-arna

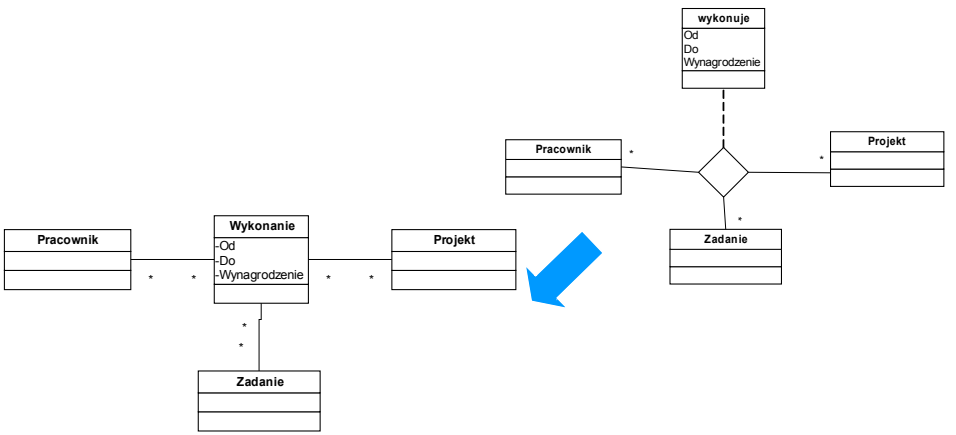
Jak wspomnieliśmy w podrozdziale 2.4.3.2 (strona 39), asocjacja n-arna nie cieszy się wielką popularnością wśród projektantów. Powody tej sytuacji oraz jej dokładniejszy opis można też tam znaleźć. Teraz interesuje nas sposób jej implementacji (bo może kiedyś jednak będzie taka konieczność).

Podobnie jak dla asocjacji z atrybutem, także tutaj musimy zamienić:

- jedną konstrukcję UML (asocjacja n-arną)
- na inną konstrukcję UML (n asocjacji binarnych oraz klasę pośredniczącą).

Sposób tej zamiany ilustruje rysunek 3-77. Dzięki zastąpieniu asocjacji n-arnej asocjacjami binarnymi oraz klasą pośredniczącą otrzymaliśmy n zwykłych asocjacji. „Nowe" asocjacje implementujemy na jeden ze znanych nam już sposobów. Niestety, przy okazji zamiany napotykamy pewne problemy z semantyką:

- Nazwa nowej klasy. Zwykle tworzymy ją na podstawie nazwy zamienianej asocjacji. Tworzenie nie jest „automatyczne", ponieważ inne są kryteria nazywania asocjacji, a inne klas.
- Musimy określić nazwy dla nowych ról i być może zmodyfikować istniejące.
- Należy określić liczności nowych asocjacji. Czasami te stare mogą być inne. No i trzeba „wymyślić" nowe.



3-77 Zamiana asocjacji kwalifikowanej na asocjacje binarne

Z powodu wprowadzenia nowej klasy mamy też utrudniony dostęp do obiektów docelowych. Przed zamianą mogliśmy bezpośrednio (po asocjacji)

przejść np. z projektu do pracownika. Po zamianie trzeba nawigować przez klasę pośredniczącą.

3.2.3.6 Implementacja agregacji

To będzie chyba najkrótszy podrozdział w tej książce.

Zanim zaczniemy rozważać sposoby implementacji agregacji (teoretyczne wyjaśnienie pojęcia było w podrozdziale 2.4.3.6 na stronie 45), zastanówmy się, czy zastosowanie agregacji niesie jakieś konsekwencje dla zaangażowanych obiektów. Odpowiedź jest tylko jedna: nie! W związku z tym agregacje implementujemy dokładnie tak samo jak klasyczne asocjacje[14].

3.2.3.7 Implementacja kompozycji

Sprawa z implementacją kompozycji nie będzie taka szybka i łatwa jak w przypadku agregacji. Jest to związane ze specjalnymi cechami tej konstrukcji - omawialiśmy je w podrozdziale 2.4.3.6 na stronie 45.

Aby nie utrudniać sobie życia, część „asocjacyjną" kompozycji zrealizujemy na dotychczasowych zasadach. Problemy do rozwiązania:

- Blokowanie samodzielnego tworzenia części – ponieważ część nie może istnieć bez całości,
- Zakazanie współdzielenia części,
- Usuwanie części przy usuwaniu całości – ponownie: ponieważ część nie może istnieć bez całości.

Wydaje się, że mamy co najmniej dwa generalne podejścia:

- Zmodyfikowanie istniejącego rozwiązania zaimplementowanego w oparciu o identyfikatory lub natywne referencje,
- Skorzystanie z całkiem nowej konstrukcji, do tej pory nieomawianej - klas wewnętrznych.

Na razie zajmijmy się tym pierwszym przypadkiem, czyli modyfikacją rozwiązania z natywnymi referencjami.

W jaki sposób zablokujemy samodzielne tworzenie części? Na początek spróbujmy odpowiedzieć na następujące pytanie: co zrobić, aby w ogóle nie dało się utworzyć obiektu jakiejś klasy? (ktoś wie?) Trzeba jej konstruktor uczynić prywatnym. W tej sytuacji nie będzie dostępny dla kodu znajdu-

[14] Krótko i treściwie – prawda?

jącego się poza klasą. Po co nam klasa, dla której nie da się stworzyć obiektów (równie dobrze możemy zrobić ją jako abstrakcyjną). Stworzymy – tylko jak? Oczywiście dodamy metodę – klasową (bo chcemy jej używać do tworzenia obiektów). Jako metoda znajdująca się w tej samej klasie ma dostęp do prywatnego konstruktora. Powinna charakteryzować się następującymi cechami:

- Pobierać referencję do całości. Oprócz tego musi też sprawdzać, czy jest ona prawidłowa. Ktoś mógłby przekazać np. `null`.

- Tworzyć obiekt części i ewentualnie go zwracać. W pewnych sytuacjach udostępnianie referencji do obiektu-części może nie być wskazane, ponieważ mając taką referencję, programista może zrobić z obiektem, co mu się podoba (np. połączyć go z inną całością, a to kłóci się z semantyką kompozycji). W takiej sytuacji tworzymy metody operujące na części z poziomu całości.

- Dodawać informacje o powiązaniu zwrotnym – aby zachować cechy asocjacji.

Oto jak może wyglądać przykładowy kod (listing 3-33) realizujący taką funkcjonalność. W tym przykładzie zdecydowaliśmy się używać nazw typu „Część" oraz „Całość", aby ułatwić zrozumienie. Ważniejsze miejsca prezentowanego programu:

- (1). Klasa opisująca część.

- (2). Ponieważ część może być połączona tylko z jedną całością, używamy pojedynczej referencji zamiast kontenera.

- (3). Konstruktor klasy. Zwróćmy uwagę na to, że jest prywatny. Dzięki temu tylko metody z tej samej klasy mogą tworzyć jej instancje.

- (4). Metoda tworząca część w oparciu o przekazaną całość.

- (5). Najpierw sprawdzamy, czy całość istniej. Jeżeli nie, to rzucamy wyjątek. Zamiast tego można np. zwracać `null` lub wyświetlać komunikat.

- (6). Całość jest prawidłowa, więc tworzymy część.

- (7). Dodajemy część do całości. W tym celu wywołujemy specjalną metodę z klasy `Całość`.

```
(1)   public class Czesc {
              public String nazwa;// public - dla uproszczenia
(2)           private Calosc calosc;
(3)           private Czesc(Calosc calosc, String nazwa) {
                      this.nazwa = nazwa;
                      this.calosc =calosc;
              }

(4)           public static Czesc utworzCzesc(Calosc calosc, String
              nazwa) throws Exception {
(5)                   if(calosc == null) {
                              throw new Exception("Calosc nie
                              istnieje!");
                      }

(6)                   // Utwocz nowa czesc
                      Czesc cz = new Czesc(calosc, nazwa);

(7)                   // Dodaj do calosci
                      calosc.dodajCzesc(cz);
(8)
                      return cz;
              }
      }
(9)
(10)  public class Calosc {
              private Vector<Czesc> czesci = new Vector<Czesc>();
              private String nazwa;

              public Calosc(String nazwa) {
                      this.nazwa = nazwa;
              }
(11)
              public void dodajCzesc(Czesc czesc) {
                      if(!czesci.contains(czesc)) {
                              czesci.add(czesc);
                      }
              }
(12)
              public String toString() {
                      String info = "Calosc: " + nazwa + "\n";
                      for(Czesc cz : czesci) {
                              info += "    " + cz.nazwa + "\n";
                      }

                      return info;
              }
      }
```

3-33 Implementacja kompozycji – część 1

- (8). Zwracamy referencję do nowo utworzonej części.
- (9). Klasa opisująca całość.
- (10). Kontener przechowujący powiązania (referencje do) z częściami. Tutaj nie możemy użyć pojedynczej referencji, ponieważ części może być wiele.
- (11). Metoda dodająca część do całości. Sprawdza, czy dana część nie jest już połączona z całością. Jeżeli nie, to ją dodaje; jeżeli istnieje powiązanie, to nic nie robi.
- (12). Specjalna wersja metody służącej do wyświetlania tekstowego opisu obiektu.

Powyższy kod uniemożliwia samodzielne stworzenie części (bez całości). Kolejnym problemem, który musimy rozwiązać przy okazji implementacji kompozycji, jest zakazanie współdzielenia części. Jak to zrobimy? Zmodyfikujemy metodę dodającą część:

- Będzie ona sprawdzała, czy dana część nie jest już gdzieś dodana,
- Oprócz dodawania informacji o powiązaniu z podaną częścią zapamiętuje (globalnie – czyli dla całej ekstensji) fakt, że dana część jest już powiązana z całością. Do tego będzie nam potrzebny atrybut klasowy przechowujący informacje o wszystkich częściach powiązanych z całościami.

Kod realizujący tę funkcjonalność jest pokazany na listingu 3-34. Pokazuje on tylko te fragmenty, które się różnią w stosunku do listingu 3-33. Poniżej wymieniamy ważniejsze miejsca programu:

- (1). Kontener przechowujące referencje do obiektów-części dla konkretnego obiektu-całości.
- (2). Atrybut klasowy, będący pojemnikiem pamiętającym referencje do wszystkich części (połączonych ze wszystkim obiektami-całościami). Dzięki niemu jesteśmy w stanie sprawdzić, czy jakaś część jest już połączona z którymkolwiek obiektem-całością.

- (3). Zmodyfikowana wersja metody dodającej część.
- (4). Najpierw sprawdzamy, czy podana część nie jest już z nami połączona. Jeżeli tak, to metoda kończy swoje działanie.
- (5). Jeżeli nie, to sprawdzamy, czy część jest połączona z którymkolwiek obiektem całością. Jeżeli tak, to rzucamy wyjątek z odpowiednim komunikatem.
- (6). Dodajemy część do naszej całości.
- (7). Zapamiętujemy fakt połączenia danej części na globalnej liście wszystkich części.

```
     public class Calosc {
(1)          private Vector<Czesc> czesci = new Vector<Czesc>();

(2)          private static HashSet<Czesc> wszystkieCzesci = new
             HashSet<Czesc>();

             // [...]

(3)          public void dodajCzesc(Czesc czesc) throws Exception {
(4)                  if(!czesci.contains(czesc)) {
                             // Sprawdz czy ta czesc nie zostala dodana
                             do jakiejs calosci
(5)                          if(wszystkieCzesci.contains(czesc)) {
                                     throw new Exception("Ta czesc jest
                                     juz powiazana z caloscia!");
                             }

(6)                          czesci.add(czesc);

                             // Zapamietaj na liscie wszystkich czesci
                             (przeciwdziala wspoldzielniu czesci)
(7)                          wszystkieCzesci.add(czesc);
                     }
             }
```

```
        // [...]
}
```

3-34 Implementacja kompozycji – część 2

Ostatnim problemem do rozwiązania jest usuwanie części przy usuwaniu całości. Tutaj mała dygresja: w językach typu Java oraz C# nie ma możliwości ręcznego usuwania obiektu. Obiekt jest oznaczany jako gotowy do usunięcia, gdy nie jest osiągalny, czyli gdy nie istnieją żadne referencje wskazujące na niego. Następnie, gdy maszyna wirtualna uzna, że np. brakuje pamięci, to usuwa obiekty. Innymi słowy: nie istnieje żadne specjalne słowo kluczowe języka, które usunie wskazany przez nas obiekt[15]. Inaczej jest w C++. Tam mamy specjalne słowo kluczowe, które powoduje ręczne usunięcie obiektu. Programując w tym języku, usunięcia części warto umieścić w destruktorze (specjalna metoda, analogiczna do konstruktora, ale wywoływana w momencie usuwania obiektu). Dzięki temu całość jest w miarę zautomatyzowana.

Wracając do języków zarządzanych (czyli właśnie Java czy C#) – co możemy zrobić? w przypadku naszej implementacji warto:

- stworzyć metodę klasową usuwającą całość z ekstensji (czyli usuwającą referencję do obiektu),
- powyższa metoda powinna również zadbać o usunięcie informacji z globalnej listy części (przeciwdziałającej współdzieleniu).

Warto również pamiętać, że obiekt nie zostanie usunięty gdy jakakolwiek referencja w systemie na niego pokazuje. Z tego powodu trzeba również sprawdzić wszystkie inne powiązania. Oczywiście można to zrobić sprytniej: ponieważ wszystkie powiązania są dwukierunkowe (namawiałem na takie rozwiązanie), obiekt wie, kto na niego pokazuje (bo on też to robi). Dzięki temu nie trzeba przeglądać wszystkich istniejących powiązań, a tylko

[15] Na pierwszy rzut oka wygląda to na poważne ograniczenie. W rzeczywistości jest dużym ułatwieniem dla programistów. Dlaczego? Wyobraźmy sobie, co się stanie, gdybyśmy usunęli obiekt, a następnie odwołali się do niego (jest to tzw. „problem zwisających wskaźników"). Zawieszenie się programu prawie murowane. Piszę „prawie" bo w rzeczywistości, w zależności od konkretnego przypadku efekty byłyby bardziej losowe (a przez to trudniejsze do zidentyfikowania): natychmiastowe zakończenie się programu, późniejsze zakończenie się programu czy wreszcie, czasami, nic by się nie działo.

te wychodzące z danego obiektu. Innym sposobem (dość radykalnym) jest nieudostępnianie referencji do obiektu-części w ogóle. W takiej sytuacji wszelkie operacje na nim, wykonuje się za pomocą metod obiektu-całości.

Innym podejściem do implementacji kompozycji jest wykorzystanie klas wewnętrznych języka Java. Mechanizm ten charakteryzuje się następującymi cechami:

- Obiekt klasy wewnętrznej nie może istnieć bez (otaczającego go) obiektu klasy zewnętrznej.

- Obiekt klasy wewnętrznej ma bezpośredni dostęp do inwariantów obiektu klasy zewnętrznej.

Spójrzmy na kod z listingu 3-35. Co prawda nie wywołuje błędu, ale jego efekt nie jest taki jaki byśmy chcieli. Obiekt klasy wewnętrznej (`Czesc`) ma dostęp do obiektu klasy zewnętrznej (`Calosc`) – to dobrze. Niestety obiekt klasy zewnętrznej (`Calosc`) nic nie wie, że utworzono obiekt klasy wewnętrznej (`Czesc`) – a to już bardzo źle. Jakoś sobie z tym poradzimy, tyle że trzeba będzie odpowiednią funkcjonalność „ręcznie" dorobić.

```
// Bledny efekt: utworzenie nowej czesci w kontekscie istniejacej
// calosci, ale bez informowania o tym calosci.

Calosc.Czesc cz = c1.new Czesc("Czesc 02");
```

3-35 Przykład użycia klas wewnętrznych

Analogicznie jak w poprzednim przypadku należy zadbać, aby:

- klasa wewnętrzna miała prywatny konstruktor,

- klasa zewnętrzna dostarczała dedykowaną metodę zapewniającą właściwe utworzenie obiektów-części.

Jak wcześniej wspomnieliśmy, nie wszystkie pożądane właściwości możemy uzyskać dzięki zastosowaniu klas wewnętrznych. W związku z tym ręcznie musimy zapewnić:

- Blokowanie współdzielenia części. Co prawda część jest zawsze połączona tylko z jednym obiektem-całością, ale może się zdarzyć, że różne całości byłyby połączone (pokazywałyby na) z tą samą częścią (a to nam się nie podoba).

- Usuwanie części przy usuwaniu całości (podobnie jak w przypadku poprzedniego sposobu implementacji kompozycji).

Spójrzmy na listing 3-36 przedstawiający przykładową implementację kompozycji przy pomocy klas wewnętrznych. I tradycyjne komentarze dotyczące kodu:

```
      public class Calosc {
              private String nazwaCalosci;
(1)           private Vector<Czesc> czesci = new Vector<Czesc>();
              public Calosc(String nazwa) {
                      nazwaCalosci = nazwa;
              }
(2)           public Czesc utworzCzesc(String nazwa) {
(3)                   Czesc czesc = new Czesc(nazwa);
(4)                   czesci.add(czesc);

                      return czesc;
              }

              // Klasa wewnetrzna – czesc.
(5)           public class Czesc {
                      private String nazwaCzesci;
                      // Ze wzgledu na specyfike klas wewnetrznych, nie
                      potrzebujemy referencji pokazujacej na calosc.

                      public Czesc(String nazwa) {
                              nazwaCzesci = nazwa;
                      }
              }
      }
```

3-36 Implementacja kompozycji przy pomocy klas wewnętrznych

- (1). Kontener przechowujący referencje do obiektów-części. Musimy go dodać, ponieważ połączenie w tę stronę nie jest zapewniane przez semantykę klas wewnętrznych.

- (2). Metoda tworzącą część i zwracająca referencję do niej. Warto zwrócić uwagę, że nie jest klasowa i znajduje się w całości, a nie jak poprzednio, w części.

- (3). Fizyczne utworzenie obiektu-części jako instancji klasy wewnętrznej.

- (4). Zapamiętanie referencji do niego.

- (5). Klasa wewnętrzna (warto zwrócić uwagę, że faktycznie znajduje się wewnątrz głównej klasy – patrz nawiasy). Dzięki temu, że każda instancja klasa wewnętrznej ma automatycznie dostęp do obiektu ją otaczającego, nie musimy tworzyć ręcznego powiązania (tak jak robiliśmy to poprzednio).

I jeszcze kilka słów podsumowania dotyczącego implementacji kompozycji. Zasadnicza kwestia jest następująca: które podejście wybrać? Czy to z klasą wewnętrzną czy też „klasyczne"? Tradycyjnie, nie ma jednoznacznej odpowiedzi. Mimo wszystko wydaje się, że to podejście z klasą wewnętrzną jest użyteczniejsze (dzięki cechom klas wewnętrznych jako takich). Czasami może to być też utrudnieniem – szczególnie gdy chcemy mieć większą swobodę w zarządzaniu obiektem-częścią. No, ale wtedy pewnie nie powinniśmy używać kompozycji...

3.2.4 Klasa ObjectPlusPlus

Podobnie jak w poprzednim podrozdziale (3.1.7, strona 124) tak i tutaj pozwolimy sobie na pewne spostrzeżenie. Otóż przedstawione sposoby implementacji zarządzania asocjacjami będą (prawie) takie same dla każdej biznesowej klasy w systemie. Co więcej, będą (prawie) takie same dla każdej asocjacji nawet w tej samej klasie. Innymi słowy, wielokrotnie będziemy musieli powielać ten sam kod. Żadnemu dobremu programiście to się nie może podobać. W związku z tym powstaje pytanie: czy da się to jakoś zunifikować?

Naturalnie – podobnie jak przy okazji zarządzania ekstensją stworzymy specjalną klasę grupująca funkcjonalność służącą do zarządzania asocjacjami. Co więcej, ta nasza nowa klasa, nazwijmy ją `ObjectPlusPlus`, będzie miała „za darmo" całą funkcjonalność dotyczącą obsługi ekstensji, trwałości itp. Jak to osiągniemy? Klasa `ObjectPlusPlus` będzie dziedziczyć z klasy `ObjectPlus`. Zwróć uwagę, drogi Czytelniku, jak użyteczne jest dziedziczenie – dzięki dosłownie dodaniu dwóch słów („`extends ObjectPlus`") do kodu nowej klasy mocno wzbogacamy jej funkcjonalność.

Założenia tej nowej klasy (`ObjectPlusPlus`) można podsumować w następujący sposób:

- Stworzymy klasę, z której będą dziedziczyć wszystkie biznesowe klasy w naszej aplikacji.

- Ta nowa klasa będzie dziedziczyła z istniejącej klasy `ObjectPlus`. Dzięki temu ułatwi też nam zarządzanie ekstensją, zapewnianie trwałości itp.

- Wyposażymy ją w funkcjonalność ułatwiającą zarządzanie:

    - zwykłymi asocjacjami binarnymi,

    - asocjacjami kwalifikowanymi,

    - kompozycjami (częściowo – tylko warunek nr 2).

- Zastosujemy drugie z omawianych podejść do implementacji asocjacji: w oparciu o referencje.

Aby wcześniejsze rozwiązanie uogólnić, musimy je trochę rozbudować (skomplikować). Ponieważ wszystkie asocjacje w ramach jednego obiektu będą przechowywane w jednej kolekcji, nie możemy zastosować zwykłego pojemnika typu `Vector` czy `ArrayList`. Użyjemy kontenera przechowującego klucze i wartości:

- Kluczem będzie nazwa roli asocjacji. Czyli w jednym obiekcie będzie tyle kluczy, ile jest z nią połączonych ról.

- Wartością będzie mapa zawierająca:

    - Klucz będący kwalifikatorem. Gdy nie chcemy użyć asocjacji kwalifikowanej, kwalifikator będzie tożsamy z obiektem docelowym.

    - Wartość będącą referencją do konkretnego powiązania.

Powyższe założenie wymaga krótkiego komentarza. Otóż gdybyśmy nie chcieli implementować asocjacji kwalifikowanej, to zamiast powyższej mapy moglibyśmy użyć zwykłego pojemnika, np. `ArrayList`. Ponieważ w przypadku asocjacji kwalifikowanej chcemy móc szybko dotrzeć do obiektu docelowego, wykorzystujemy mapę. Ktoś może zarzucić nam, że w przypadku przechowywania informacji o klasycznych powiązaniach klucz i wartość się dublują. To jest prawda, ale aby temu przeciwdziałać, trzeba by było zdecydować się na jedno z poniższych rozwiązań:

- zrezygnować z wydajnej implementacji asocjacji kwalifikowanej i zaimplementować wyszukiwanie obiektu na podstawie kwalifikatora za pomocą ręcznego przeglądania całej (klasycznej) kolekcji,

- stworzyć dedykowany pojemnik dla powiązań z asocjacji kwalifikowanej (co skomplikuje implementację),

- w ogóle zrezygnować z umieszczania w uniwersalnej klasie Obsługi asocjacji kwalifikowanych.

Jak widać, żadna z powyższych propozycji nie jest idealna. Wydaje się, że możemy sobie pozwolić na niewielkie „marnotrawstwo" pamięci co znacząco ułatwi nam implementację.

Wracając do proponowanego rozwiązania: w klasie zarządzającej umieścimy również atrybut klasowy przechowujący referencje do wszystkich obiektów dodanych jako części, co umożliwi pilnowanie warunku nr 2 dotyczącego kompozycji (brak współdzielenia). Innymi słowy, ten nowy kontener będzie zawierał powiązania istniejące w ramach wielu asocjacji.

Ponieważ cały ten wywód może się wydawać nieco skomplikowany, spójrzmy na rysunek 3-78, który ma ułatwić jego zrozumienie.

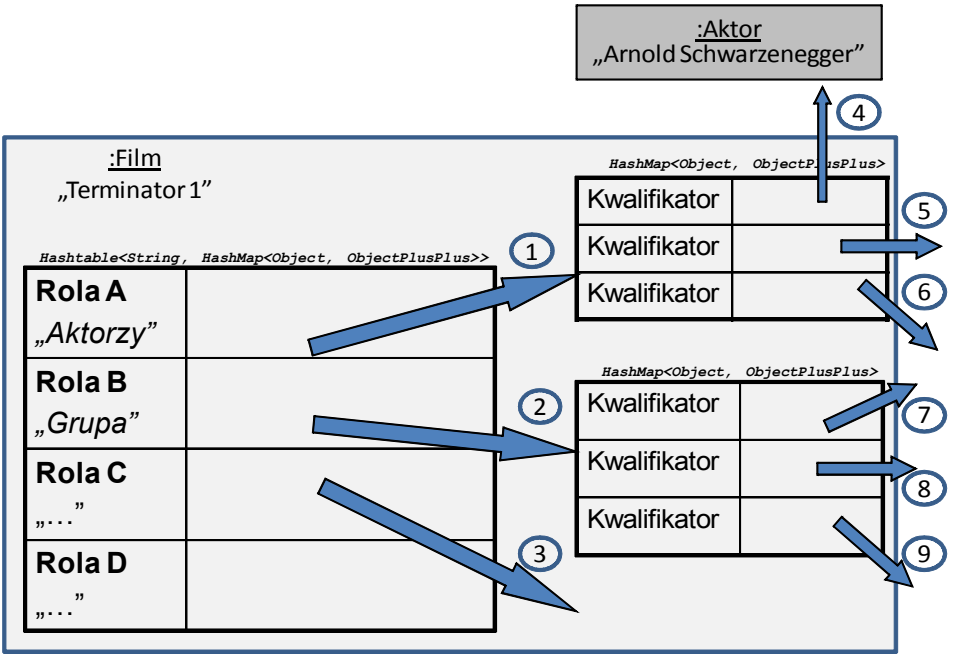
Główną część rysunku zajmuje konkretny obiekt klasy `Film` zatytułowany „Terminator 1". Wewnątrz niego widzimy kolekcję mapującą (parametryzowany `Hashtable`) zawierającą:

- klucze z nazwami ról asocjacji („Aktorzy", „Grupa")

- oraz odpowiadające im wartości w postaci referencji (1, 2, 3) do kolejnych pojemników mapujących (parametryzowane `HashMap`'y) – po jednym dla każdej roli:

    - kluczem jest kwalifikator,

    - wartością jest referencja pokazująca na konkretny obiekt docelowy, np. dla roli „Aktorzy", poprzez kolejne połączenia (1, 4) dochodzimy do instancji klasy `Aktor` o nazwie „Arnold Schwarzenegger". Kolejne referencje (5, 6) prowadzą do innych aktorów (niepokazani z braku miejsca) grających w tym filmie. Inne referencje (7, 8 , 9) pokazują na obiekty docelowe w ramach roli „Grupa" (opisującej przynależność filmu do jakichś grup).



3-78 ilustracja wyjaśniająca sposób przechowywania informacji dotyczących asocjacji w klasie ObjectPlusPlus

Skoro już teraz mamy dobrze opracowaną koncepcję, to pozostaje nam tylko wprowadzić ją w czyn – czyli napisać odpowiedni program. Ale uwaga – ze względu na objętość kod tej klasy przedstawiam w trzech częściach: listingi 3-37, 3-38 oraz 3-39. Również niektóre linie tekstu nie mieściły się w jednym wierszu i dlatego są przeniesione do następnego (z niewielkim wcięciem na początku przeniesionego wiersza). Poniższe akapity zawierają dokładny opis wymienionych listingów.

Pierwsza część przykładowej implementacji zunifikowanego zarządzania asocjacjami jest zamieszczona na listingu 3-37. Jak zwykle poniżej skomentowałem ważniejsze miejsca programu:

- (1). Fragment klasy `ObjectPlusPlus` implementującej zarządzanie asocjacjami.

- (2). Główna kolekcja mapująca zadeklarowana jako parametryzowana `Hashtable`. Kluczem jest nazwa roli asocjacji, a wartością kolejna mapa (parametryzowany `HashMap`) przechowująca:
    - Klucz będący kwalifikatorem,
    - Wartość będącą referencją do obiektu docelowego.
- (3). Kolekcja (parametryzowany `HashSet`) przechowująca referencje do wszystkich obiektów części połączonych z którymkolwiek obiektem całością. Ten element potrzebny jest nam ze względu na implementację warunku nr 2 (część może należeć tylko do jednej całości) dotyczącego kompozycji.
- (4). Konstruktor klasy. Wszystkie obiekty korzystające (dziedziczące) z klasy `ObjectPlusPlus` muszą go wywoływać. Analogicznie, w jego ciele jest wołanie konstruktora z `ObjectPlus` (m.in. dodającego obiekt do ekstensji). Bez tego wołania nasze klasy nie będą działały prawidłowo.
- (5). Podstawowa wersja metody dodającej powiązanie. Warto zwrócić uwagę, że jest zadeklarowana jako prywatna. Jest tak dlatego, że wykorzystujemy ją tylko wewnętrznie. Dla programistów korzystających z naszej klasy istnieją jej przeciążone i bardziej przyjazne (prostsze) wersje.
- (6). Licznik jest wykorzystywany jako zabezpieczenie przeciw zapętleniu. Jak może pamiętasz, drogi Czytelniku, wcześniej wykorzystywaliśmy do tego celu sprawdzenie, czy dany obiekt jest już przechowywany w pojemniku (patrz opis listingu 3-28 na stronie 142). Teraz wprowadzimy inne rozwiązanie, które pewnie jest też wydajniejsze. Trzeba go też użyć, jeżeli chcemy pamiętać wiele powiązań (w ramach tej samej roli) do tego samego obiektu. Działa to w ten sposób, że przy wołaniu tej samej metody dodającej powiązanie z punktu widzenia obiektu docelowego zmniejszamy jego wartość. W efekcie, przy następnym wołaniu, dochodzi do przerwania wykonywania się metody i braku zapętlenia.

- (7). Sprawdzamy, czy istnieje już kolekcja dla podanej, konkretnej nazwy roli.
- (8). Jeżeli tak, to ją zwracamy.
- (9). Jeżeli nie to tworzymy odpowiednią kolekcję (parametryzowany `HashMap`) i dodajemy do głównej mapy (10).
- (11). Jeżeli nie, mamy jeszcze takiego kwalifikatora (bo muszą być unikatowe), to dodajemy informacje o powiązaniu (12).
- (13). Wołamy tę samą metodę, ale na rzecz obiektu docelowego, tak aby utworzyć powiązanie zwrotne. Podajemy zmniejszoną wartość licznika, tak aby przy kolejnym wołaniu nie doszło do zapętlenia.

```
 (1)  public class ObjectPlusPlus extends ObjectPlus implements
      Serializable {
             /**
              * Przechowuje informacje o wszystkich powiazaniach tego
             obiektu.
              */
(2)          private Hashtable<String, HashMap<Object, ObjectPlusPlus>>
             powiazania = new Hashtable<String, HashMap<Object,
                ObjectPlusPlus>>();

             /**
              * Przechowuje informacje o wszystkich czesciach powiazanych
             z ktorymkolwiek z obiektow.
              */
(3)          private static HashSet<ObjectPlusPlus> wszystkieCzesci = new
                HashSet<ObjectPlusPlus>();

             /**
              * Konstruktor.
              *
              */
(4)          public ObjectPlusPlus() {
                     super();
             }

(5)          private void dodajPowiazanie(String nazwaRoli, String
                odwrotnaNazwaRoli, ObjectPlusPlus obiektDocelowy, Object
                kwalifikator, int licznik) {
                     HashMap<Object, ObjectPlusPlus> powiazaniaObiektu;

(6)                  if(licznik < 1) {
                            return;
                     }
```

```
(7)                     if(powiazania.containsKey(nazwaRoli)) {
                                // Pobierz te powiazania
(8)                             powiazaniaObiektu =
                                powiazania.get(nazwaRoli);
                        }
                        else {
(9)                             // Brak powiazan dla takiej roli ==> utworz
                                powiazaniaObiektu = new HashMap<Object,
(10)                            ObjectPlusPlus>();
                                powiazania.put(nazwaRoli, powiazaniaObiektu);
(11)                    }
                        if(!powiazaniaObiektu.containsKey(kwalifikator)) {
(12)                            // Dodaj powiazanie dla tego obiektu
                                powiazaniaObiektu.put(kwalifikator,
                                obiektDocelowy);
(13)                            // Dodaj powiazanie zwrotne
                                obiektDocelowy.dodajPowiazanie(odwrotnaNazwaR
                                oli, nazwaRoli, this, this, licznik - 1);
                        }
                }
                // [...]
```

3-37 Implementacja klasy ObjectPlusPlus - część pierwsza

Teraz warto chwilę się zastanowić i przemyśleć, czy wszystko jest zrozumiałe. Jeżeli tak, to zabieramy się do analizy następnej części implementacji klasy `ObjectPlusPlus`, która jest przedstawiona na listingu 3-38. I znowu nasza „świecka tradycja" – opis ważniejszych miejsc implementacji:

- (1). Publiczna wersja metody dodającej asocjację kwalifikowaną. Jak widać, nie ma już parametru dotyczącego licznika. To są nasze wewnętrzne sprawy implementacyjne i nie należy obciążać nimi programistów korzystających z naszej biblioteki.

- (2). Metoda ta (1) korzysta z wewnętrznej implementacji, przekazując odpowiednie parametry. Tworzenie jednej (lub kilku) parametryzowanej metody i wielu jej publicznych (przeciążonych) wersji jest częstą praktyką. Dzięki temu zasadniczy kod mamy w jednym miejscu, a programiści korzystający z naszej pracy mają ułatwione życie. Nie bez znaczenia jest też kwestia wprowadzania poprawek – robimy to tylko w jednym miejscu.

- (3). Publiczna wersja metody dodającej powiązanie bez kwalifikatora.

- (4). Tym razem wołamy metodę (1), podając jako kwalifikator obiekt docelowy (bo coś musimy podać).

- (5). Metoda dodająca część w ramach kompozycji.

- (6). Sprawdzamy, czy podany obiekt-część nie występuje już gdzieś jako część. Jeżeli tak, to rzucamy wyjątek.

- (7). Wołamy standardową metodę tworzącą asocjację. Czyli mamy kolejny przykład na ponowne użycie. Jest to nawet zgodne z semantyką kompozycji – przecież jest też w końcu asocjacją.

- (8). Zapamiętujemy obiekt-część na liście wszystkich części (aby zapobiegać współdzieleniu części).

- (9). Metoda zwracająca tablicę powiązań dla podanej nazwy roli. Oczywiście zamiast tablicy możemy zwracać inny pojemnik, ale tablice w są wykorzystywane przez wiele innych metod.

- (10). Sprawdzamy, czy mamy powiązania dla tej roli. Jeżeli nie, to rzucamy wyjątek.

- (11). Odczytujemy powiązania z mapy (na podstawie klucza).

- (12). Zwracamy tablicę, używając metody wbudowanej w mapę.

```
public class ObjectPlusPlus extends ObjectPlus implements
   Serializable {
        // [...]
(1)     public void dodajPowiazanie(String nazwaRoli, String
           odwrotnaNazwaRoli, ObjectPlusPlus obiektDocelowy,
        Object
           kwalifikator) {
(2)
                dodajPowiazanie(nazwaRoli, odwrotnaNazwaRoli,
                obiektDocelowy, kwalifikator, 2);
        }
(3)
        public void dodajPowiazanie(String nazwaRoli, String
           odwrotnaNazwaRoli, ObjectPlusPlus obiektDocelowy) {
(4)
                dodajPowiazanie(nazwaRoli, odwrotnaNazwaRoli,
                obiektDocelowy, obiektDocelowy);
```

```
            }
(5)
            public void dodajCzesc(String nazwaRoli, String
               odwrotnaNazwaRoli, ObjectPlusPlus obiektCzesc) throws
               Exception {

(6)                 // Sprawdz czy ta czesc juz gdzies nie wystepuje
                    if(wszystkieCzesci.contains(obiektCzesc)) {
                            throw new Exception("Ta czesc jest już
                            powiazana z jakas caloscia!");
                    }
(7)
                    dodajPowiazanie(nazwaRoli, odwrotnaNazwaRoli,
                    obiektCzesc);

(8)                 // Zapamietaj dodanie obiektu jako czesci
                    wszystkieCzesci.add(obiektCzesc);
            }
(9)
            public ObjectPlusPlus[] dajPowiazania(String nazwaRoli)
               throws Exception {
                    HashMap<Object, ObjectPlusPlus> powiazaniaObiektu;
(10)
                    if(!powiazania.containsKey(nazwaRoli)) {
                            // Brak powiazan dla tej roli
                            throw new Exception("Brak powiazan dla
                            roli: " + nazwaRoli);
(11)                }
(12)                powiazaniaObiektu = powiazania.get(nazwaRoli);
                    return (ObjectPlusPlus[])
                    powiazaniaObiektu.values().toArray(new
                    ObjectPlusPlus[0]);
            }
(13)
            public void wyswietlPowiazania(String nazwaRoli,
               PrintStream stream) throws Exception {
                    HashMap<Object, ObjectPlusPlus> powiazaniaObiektu;
(14)
                    if(!powiazania.containsKey(nazwaRoli)) {
                            // Brak powiazan dla tej roli
                            throw new Exception("Brak powiazan dla
                            roli: " + nazwaRoli);
(15)                }
(16)                powiazaniaObiektu = powiazania.get(nazwaRoli);
(17)                Collection col = powiazaniaObiektu.values();
                    stream.println(this.getClass().getSimpleName() + "
(18)                powiazania w roli " +  nazwaRoli + ":");
                    for(Object obj : col) {
                            stream.println("   " + obj);
                    }
            }
            // [...]
```

3-38 Implementacja klasy ObjectPlusPlus - część druga

- (13). Metoda wyświetlająca powiązania dla podanej roli na podanym strumieniu. Dzięki temu możemy je wyświetlić na np. konsoli, ale również skierować do np. pliku.

- (14). Sprawdzamy, czy mamy powiązania dla tej roli. Jeżeli nie, to rzucamy wyjątek.

- (15). Odzyskujemy powiązania dla tej roli i pobieramy je jako wartości (16).

- (17). Wyświetlamy nazwę klasy, korzystając z refleksji.

- (18). Iterujemy po wszystkich obiektach docelowych, wysyłając ich opisy do strumienia.

Sporo tego było, ale na szczęście już dochodzimy do ostatniej części implementacji klasy `ObjectPlusPlus`. Ważniejsze miejsca listingu 3-39:

- (1). Metoda zwracająca obiekt docelowy na podstawie kwalifikatora. Wykorzystywana jest tylko w przypadku asocjacji kwalifikowanej. Dla asocjacji „klasycznych" mamy metodę `dajPowiazania(...)`.

- (2). Sprawdzamy, czy mamy powiązania dla tej roli. Jeżeli nie, to rzucamy wyjątek.

- (3). Odzyskujemy mapę zawierającą powiązania tej roli.

- (4). Sprawdzamy, czy ta mapa zawiera podany kwalifikator. Jeżeli nie, to rzucamy wyjątek (5).

- (6). Zwracamy obiekt na podstawie klucza (kwalifikatora).

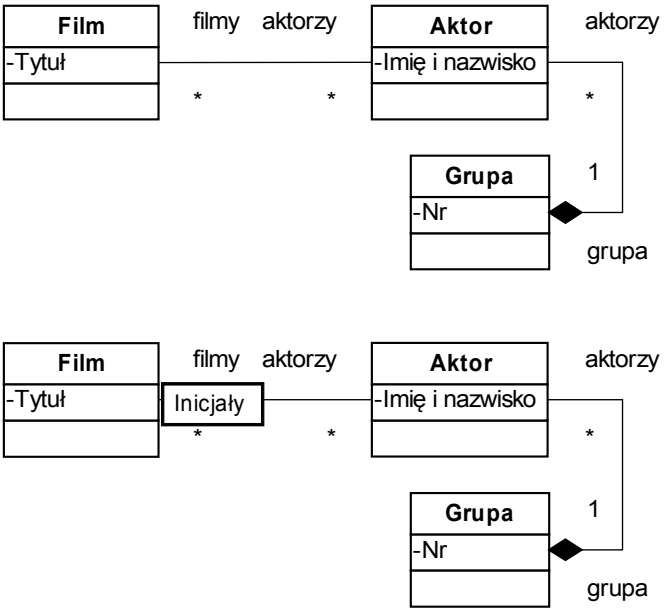
```
    public class ObjectPlusPlus extends ObjectPlus implements
       Serializable {
            // [...]
(1)         public ObjectPlusPlus dajPowiazanyObiekt(String nazwaRoli,
               Object kwalifikator) throws Exception {
                    HashMap<Object, ObjectPlusPlus> powiazaniaObiektu;

(2)                 if(!powiazania.containsKey(nazwaRoli)) {
                           // Brak powiazan dla tej roli
                           throw new Exception("Brak powiazan dla
```

```
                        roli: " + nazwaRoli);
                }

(3)             powiazaniaObiektu = powiazania.get(nazwaRoli);
(4)             if(!powiazaniaObiektu.containsKey(kwalifikator)) {
                        // Brak powiazan dla tej roli
(5)                     throw new Exception("Brak powiazania dla
                        kwalifikatora: " + kwalifikator);
                }

(6)             return powiazaniaObiektu.get(kwalifikator);
            }
    }
```

3-39 Implementacja klasy ObjectPlusPlus - część trzecia

Po tak dużej dawce wyjaśnień (ponad 30 punktów w trzech listingach) należy nam się coś lżejszego. Jako zasłużony relaks zobaczymy, jak korzystać z tej naszej nowej funkcjonalności. Na początku spójrzmy na rysunek 3-79 przedstawiający dwie podobne sytuacje biznesowe. Różni je tylko to, że w drugim przypadku wykorzystujemy asocjację kwalifikowaną. Naszym zadaniem jest stworzyć program, który będzie w stanie zapamiętać te dane (i do tego ma to zrobić trwale).

Odpowiedni kod opisujący poszczególne klasy biznesowe pokazany jest na listingu 3-40. Jest on tak (zaskakująco?) prosty, że nie będziemy umieszczali naszego tradycyjnego wypunktowania. Zwrócimy uwagę tylko na fakt, że dziedziczymy z klasy `ObjectPlusPlus` i w każdej z nich wołamy konstruktor z nadklasy.



3-79 Sytuacja biznesowa do zaimplementowania w oparciu o klasę ObjectPlusPlus

```
public class Aktor extends ObjectPlusPlus {
        private String imieNazwisko;

        public Aktor(String imieNazwisko) {
                super();// wolanie konstruktora z nadklasy -
                obowiazkowe!
                this.imieNazwisko = imieNazwisko;
        }

        public String toString() {
                return "Aktor: " + imieNazwisko;
        }
}

public class Film extends ObjectPlusPlus {
        private String tytul;

        public Film(String tytul) {
                super();// wolanie konstruktora z nadklasy -
                obowiazkowe!
                this.tytul = tytul;
        }
```

```
        public String toString() {
                return "Film: " + tytul;
        }
}

public class Grupa extends ObjectPlusPlus {
        private int nr;

        public Grupa(int nr) {
                super();// wolanie konstruktora z nadklasy -
                obowiazkowe!
                this.nr = nr;
        }

        public String toString() {
                return "Grupa: " + nr;
        }
}
```

3-40 Klasy służące do zaimplementowania sytuacji biznesowej z rysunku 3-79

Spójrzmy teraz na nasze klasy biznesowe w działaniu - listing 3-41. I krótkie omówienie:

- (1, 2, 3). Tworzymy klasy biznesowe – na razie bez powiązań.
- (4). Tworzymy powiązanie w ramach asocjacji pomiędzy aktorami a filmami. Automatycznie tworzone jest też powiązanie zwrotne.
- (5). Utworzenie powiązania kwalifikowanego z kwalifikatorem będącym inicjałami aktora.
- (6). Tworzenie powiązań część – całość (kompozycja).
- (7). Po usunięciu komentarza z tego kodu otrzymamy wyjątek. Powodem jest to, iż dodawany aktor (część) należy już do grupy (całość).
- (8). Wyświetlamy powiązania.
- (9). Odzyskujemy obiekt docelowy (aktor) na podstawie jego kwalifikatora (inicjały).

```
     // Utworz nowe obiekty (na razie bez informacji o powiazaniach)
(1)  Aktor a1 = new Aktor("Arnold Schwarzenegger");
     Aktor a2 = new Aktor("Michael Biehn");
     Aktor a3 = new Aktor("Kristanna Loken");
```

```
(2)   Film f1 = new Film("Terminator 1");
      Film f3 = new Film("Terminator 3");

(3)   Grupa g1 = new Grupa(1);
      Grupa g2 = new Grupa(2);

      // Dodaj informacje o powiazaniach
(4)   f1.dodajPowiazanie("aktorzy", "filmy", a1);
      // f1.dodajPowiazanie("aktorzy", "filmy", a2);
(5)   f1.dodajPowiazanie("aktorzy", "filmy", a2, "MB");// wykorzystanie
      asocjacji kwalifikowanej
      f3.dodajPowiazanie("aktorzy", "filmy", a1);
      f3.dodajPowiazanie("aktorzy", "filmy", a3);

(6)   g1.dodajCzesc("czesc", "calosc", a1);
      g1.dodajCzesc("czesc", "calosc", a2);
      g2.dodajCzesc("czesc", "calosc", a3);
(7)   // g2.dodajCzesc("czesc", "calosc", a1); // wyjatek poniewaz
      dodawana czesc (aktor) nalezy
      juz do innej calosci (grupy)

      // Wyswietl informacje
(8)   f1.wyswietlPowiazania("aktorzy", System.out);
      f3.wyswietlPowiazania("aktorzy", System.out);

      a1.wyswietlPowiazania("filmy", System.out);

      g1.wyswietlPowiazania("czesc", System.out);

      // test asocjacji kwalifikowanej
(9)   System.out.println(f1.dajPowiazanyObiekt("aktorzy", "MB"));
```

3-41 Wykorzystanie klas biznesowych do implementacji zadania z rysunku 3-79

I jeszcze, na koniec, przyjrzyjmy się konsoli wyświetlającej wyniki działania naszego programu z listingu 3-41.

```
Film powiazania w roli aktorzy:
   Aktor: Arnold Schwarzenegger
   Aktor: Michael Biehn
Film powiazania w roli aktorzy:
   Aktor: Arnold Schwarzenegger
   Aktor: Kristanna Loken
Aktor powiazania w roli filmy:
   Film: Terminator 1
   Film: Terminator 3
Grupa powiazania w roli czesc:
   Aktor: Arnold Schwarzenegger
   Aktor: Michael Biehn
Aktor: Michael Biehn
```

3-42 Wyniki działania programu z listingu 3-41

Tytułem podsumowania rozdziału o implementacji asocjacji możemy napisać, że mamy dwa generalne podejścia do implementacji asocjacji: za pomocą identyfikatorów oraz natywnych referencji.

Niektóre rodzaje asocjacji przed ich implementacją należy zamienić na konstrukcje równoważne. Dzięki temu implementujemy je korzystając ze znanych już sposobów.

Całą funkcjonalność związaną z zarządzaniem asocjacjami warto zgromadzić w specjalnej nadklasie (`ObjectPlusPlus`). Dzięki temu, że nowa klasa (`ObjectPlusPlus`) dziedziczy z wcześniej zaimplementowanej klasy `ObjectPlus`, obsługuje również zarządzanie ekstensjami i trwałość danych. W efekcie, pisząc zaledwie kilka linijek kodu, możemy korzystać z bardzo użytecznej funkcjonalności znacząco upraszczającej tworzenie programów.

3.3 Dziedziczenie

Od strony teoretycznej pojęcie dziedziczenia omówiliśmy w podrozdziale 2.4.4 na stronie 47. Teraz zajmiemy się odniesieniem tych informacji do popularnych języków programowania takich jak Java, C# czy C++.

Niestety, większość z wymienionych języków zapewnia tylko najprostszy rodzaj dziedziczenia: rozłączne (*disjoint*). Jedynie C++ umożliwia wykorzystywanie dziedziczenia wielokrotnego. Co w takim razie możemy zrobić, gdy jako rezultat fazy analizy pojawią się konstrukcje nieobsługiwane? Jak zwykle w tej książce – bierzemy się do pracy i przekształcamy lub samodzielnie implementujemy to co jest nam potrzebne. Następne podrozdziały zawierają propozycje radzenia sobie z problemami dla poszczególnych rodzajów dziedziczeń.

3.3.1 Dziedziczenie rozłączne

W tym przypadku mam dobre wiadomości. Ten rodzaj dziedziczenia występuje chyba we wszystkich obiektowych językach programowania i oczywiście jest też w języku Java. Jedyna rzecz, która się trochę różni w zależności od konkretnego języka, dotyczy dostępności inwariantów w podklasach. Np. w języku Java atrybuty oraz metody zadeklarowane jako `private` nie są dostępne w podklasie. W innych językach może być trochę inaczej. Po prostu trzeba o tym pamiętać, projektując klasy. Generalnie, zgodnie z zasadami hermetyzacji oraz zdrowym rozsądkiem, ukrywamy wszystkie elementy, których programista nie potrzebuje do pracy.

```
Public class Osoba {
        private String imie;
        private String nazwisko;
        private Date dataUrodzenia;
}

public class Pracownik extends Osoba {
        private boolean badaniaLekarskie;
}

public class Student extends Osoba {
        private int numerIndeksu;
}

public class Emeryt extends Osoba {
}
```

3-43 Przykładowa implementacja dziedziczenia rozłącznego

Składnia dla dziedziczenia rozłącznego różni się też w zależności od języka. W C++ oraz C# używa się jednego znaku „:”, a w Javie słowa `extends`. Listing 3-43 pokazuje przykładowy kod ilustrujący dziedziczenie rozłączne w języku Java. Odpowiada on prostemu diagramowi klas z 2-22 (strona 47). Jest on na tyle prosty, że tym razem zrezygnujemy z komentarzy.

3.3.2 Polimorficzne wołanie metod

Ten podrozdział nie dotyczy bezpośrednio dziedziczenia, ale zagadnienia, które jest kluczowe dla obiektowych języków programowania, a bez dziedziczenia nie miałoby racji bytu. Chodzi o polimorficzne wołanie metod (naturalnie związane z przesłanianiem). Pisaliśmy trochę o tym w podroz-

dziale 2.4.4.2 na stronie 49 i obiecaliśmy, że wrócimy do tego przy okazji implementacji.

Aby dobrze wyjaśnić tę kwestię, odwołamy się do przykładu przedstawionego w podrozdziale 2.4.4 (strona 47 i dalsze).

Na początek jednak wyjaśnijmy, czym różnią się pojęcia przesłaniania oraz polimorficznego wołania metod (bo często są one mylone czy też utożsamiane):

- Przesłanianie umożliwia umieszczenie w podklasie (czyli musi występować dziedziczenie) metody o takiej samej nazwie oraz liczbie i typie argumentów jak metoda w którejś z nadklas. Gdyby nie było przesłaniania, to mogłoby dojść do następującej sytuacji:

    - W którejś z nadklas hierarchii dziedziczenia jest jakaś metoda, np. `getZarobki()`,

    - W podklasie tworzymy metodę np. `getZarobki(int stawka)`. To nie stanowi problemu, ponieważ takiej metody[16] jeszcze nie ma. Co prawda doszło do przeciążenia metod, ale to nie ma w tej chwili dla nas znaczenia.

    - W podklasie (tej samej albo innej – to nie jest ważne) tworzymy metodę `getZarobki()`. W czasie kompilacji programu dostajemy informację o błędzie, ponieważ taka metoda już tam jest (została odziedziczona z nadklasy).

  Jeżeli dany język programowania dopuszcza przesłanianie, to rozpozna powyższą sytuację i na nią zezwoli.

- Polimorficzne wołanie metod oznacza, że w zależności od rodzaju obiektu, bez względu na to, jakiego typu referencją na niego pokazujemy, wywoła się metoda znajdująca się najbliżej w hierarchii dziedziczenia.

Zaraz wyjaśnimy powyższe sformułowanie na zapowiadanym przykładzie. Dodajmy tylko, że w C++, C# oraz Java przesłanianie metod oraz ich polimorficzne wołanie jak najbardziej występują. Z tą różnicą, że w C# oraz Javie wszystkie odwołania metod są polimorficzne, a w C++ trzeba je „włą-

[16] Mówiąc precyzyjniej: z taką sygnaturą, czyli nazwą, liczbą oraz typem argumentów.

czyć" za pomocą specjalnego słowa kluczowego (przy okazji deklarowania metody) `virtual`.

I oto zapowiadany przykład – listing 3-44. Ważniejsze miejsca wymagające komentarza:

```
public abstract class Osoba {
        // [...]
        public Osoba(String imie, String nazwisko, Date
        dataUrodzenia) {
                super();
                this.imie = imie;
                this.nazwisko = nazwisko;
                this.dataUrodzenia = dataUrodzenia;
        }

(1)     public abstract float getDochody();
}

public class Pracownik extends Osoba {
        // [...]
(2)     public float getDochody() {
                return getPensja();
        }
        public float getPensja() {
                return pensja;
        }
}

public class Student extends Osoba {
        private int numerIndeksu;
        private float stypendium;
        // [...]
(3)     public float getDochody() {
                return getStypendium();
        }


        public float getStypendium() {
                return stypendium;
        }
}
```

3-44 Kod ilustrujący przesłanianie metod

- (1). Deklaracja metody abstrakcyjnej (bez ciała) zwracającej dochody danej osoby.

- (2). Przesłonięta wersja metody zwracającej dochody. Dla pracownika podaje po prostu pensję.

- (3). Przesłonięta wersja metody zwracającej dochody. Dla studenta podaje stypendium.

Oczywiście, przesłanianie i polimorficzne wołanie metod działa nie tylko, gdy w nadklasie jest metoda abstrakcyjna. Równie dobrze może tam być metoda konkretna zwracająca dowolne wartości.

Spójrzmy, jak z tego korzystamy – na czym tak naprawdę polega to polimorficzne wołanie metod – listing 3-45. Zobaczmy, co tam się dzieje:

```
(1)   Osoba o1 = new Pracownik("Jan", "Kowalski", new Date(), true,
      4000.0f);
(2)   Osoba o2 = new Student("Adam", "Abacki", new Date(), 1212,
      2000.0f);

(3)   System.out.println(o1.getDochody());
(4)   System.out.println(o2.getDochody());
```

3-45 Kod ilustrujący polimorficzne wołanie metod

- (1). Tworzymy pracownika. Zwróćmy uwagę, że pokazujemy na niego referencją typu `Osoba`[17], czyli po prostu traktujemy go jak osobę.

- (2). Tworzymy studenta. Podobnie jak powyżej, pokazujemy na niego referencją typu `Osoba`, czyli ponownie traktujemy go jak osobę.

- (3). Teraz będą „czary". Wyświetlamy rezultat wołania metody podającej dochody osoby. Ale co się stało? – przecież w osobie jako takiej jest tylko metoda abstrakcyjna, która nie ma ciała. Teraz właśnie odbywa się polimorficzne wołanie metody: zanim tak naprawdę maszyna wirtualna wywoła metodę, sprawdza, z jakim obiektem ma do czynienia (a nie tylko czym na niego pokazujemy). Jeżeli nie znajdzie metody o podanej sygnaturze w tej samej klasie, to szuka w nadklasach („idąc" coraz wyżej). Dzięki temu możemy pozostawać na stosunkowo wysokim poziomie abstrakcji – nie wnikamy, z kim mamy do czynienia (studentem, pracownikiem czy emerytem) – po prostu pytamy daną osobę o dochody. A „czarną robotę" wykonuje komputer – tak to lubimy, nieprawdaż?

- (4). Sytuacja analogiczna jak w (3), ale dotyczy studenta.

[17] Typem z nadklasy zawsze można pokazywać na wszystkie typy, które z niej dziedziczą. Jest to zgodne z zasadą zamienialności, bo przecież wszystkie podklasy są też w pewnym stopniu nadklasami, np. studenci są osobami.

W przykładzie ilustrującym polimorficzne wołanie ważne jest, aby pokazywać typem z nadklasy na podklasę. Gdybyśmy używali referencji typu `Student`, wołając metodę z klasy `Student`, to gdzie tu jakaś „magia”?

```
public class Film {
        // Implementacja czesci biznesowej

        public String toString() {
                return "Film: " + tytul;
        }
}
```

3-46 Przesłanianie metod na przykładzie systemowej metody `toString()`

Warto jeszcze przytoczyć przykład metody `toString()`, która jest bardzo często przesłaniana w celu poprawy czytelności wyświetlanych informacji. Dzięki temu, stosując dość prosty kod (listing 3-46) możemy uzyskać dużo czytelniejsze efekty (konsola 3-7).

```
Ekstensja klasy Film - standardowy toString():
mt.mas.Film@126804e
mt.mas.Film@b1b4c3

Ekstensja klasy Film – przesłonięty toString():
Film: Terminator 1
Film: Terminator 2
```

3-7 Efekt działania przesłoniętej wersji metody toString()

I jeszcze na koniec dwie zagadki:

- Wspomnieliśmy, że w C++ trzeba „włączyć” polimorficzne wołanie za pomocą specjalnego słowa kluczowego. Dlaczego? Skoro jest to taka pożyteczna właściwość, a mam nadzieję, że udało mi się o tym Was przekonać. Ktoś wie? Jak nie wiadomo, o co chodzi, to chodzi o pieniądze. No, może nie do końca, bo raczej o wydajność. Język C++ był projektowany dość dawno (gdy komputery były dużo wolniejsze) i do tego z myślą o wydajności. Otóż, gdy komputer wykonuje polimorficzne wołanie metody, to w czasie działania programu musi poświęcić troszkę zasobów na sprawdzenie, z jakim obiektem ma naprawdę do czynienia, odnalezienie odpowiedniej metody w hierarchii dziedziczenia i dopiero wtedy może ją wywołać. Zajmuje to troszkę czasu, a czas to pieniądz. Dla współczesnych maszyn i nowoczesnych kompilatorów nie jest to wielkie wyzwanie – być może nawet niemie-

rzalne. Może ktoś z Czytelników zechce zaprojektować i przeprowadzić odpowiednie testy i przyśle mi wyniki?

- Czy może występować przesłanianie bez polimorficznego wołania metody? Jak najbardziej - tak jest w C++ dopóki nie użyjemy słowa `virtual`. A odwrotnie – czy może być polimorficzne wołanie metody bez przesłaniania? Gdyby nie było przesłaniania, to nie mielibyśmy wielu wersji metody i nie byłoby z czego wybierać. Czyli odpowiedź brzmi: nie.

3.3.3 Dziedziczenie typu *overlapping*

Niestety, ten typ dziedziczenia (pisaliśmy o nim w podrozdziale 2.4.4.4 na stronie 55) nie występuje bezpośrednio w żadnym z popularnych języków programowania. W związku z tym musimy go jakoś ręcznie zaimplementować. Właściwie do wyboru mamy następujące sposoby obejścia tego braku:

- Zastąpienie całej hierarchii dziedziczenia jedną klasą,
- Wykorzystanie agregacji lub kompozycji,
- Rozwiązania łączące powyższe metody.

3.3.3.1 Obejście dziedziczenia overlapping za pomocą grupowania

Rozwiązanie to jest bardzo proste i to zarówno koncepcyjnie, jak i implementacyjnie. Polega na specyficznym zgrupowaniu wszystkich klas z hierarchii *overlapping* w ramach jednej klasy. Zwykle ma ona nazwę nadklasy. Innymi słowy:

- Wszystkie inwarianty (atrybuty, metody, asocjacje) umieszczamy w jednej nadklasie,
- Dodajemy dyskryminator (specjalny atrybut), który informuje nas o rodzaju obiektu. Tu mała dygresja: w dawnych językach programowania do rozróżniania stanu atrybutów używano stałych, co niosło ze sobą liczne problemy, m.in. możliwość podstawienia zamiast stałej biznesowej będącej liczbą po prostu liczby. Dla kompilatora było to dokładnie to samo. Na szczęście nowsze języki programowania oferują typ wyliczeniowy, który jest nie tylko bezpieczniejszy, ale również wygodniejszy w użyciu. W naszym konkretnym przypadku

chcemy móc zapamiętać kilka wartości naraz. Dlatego użyjemy specjalnej kolekcji oferowanej przez Javę o nazwie `EnumSet`.

Odpowiedni kod źródłowy, odpowiadający przykładowi z 2-27 (strona 55), pokazany jest na listingu 3-47. Zwróćmy uwagę, że zgodnie z naszymi założeniami:

```
(1)   enum OsobaRodzaj {Osoba, Pracownik, Student, Emeryt};

      public class Osoba {
(2)           private String imie;
              private String nazwisko;
              private Date dataUrodzenia;

(3)           private boolean badaniaLekarskie;

(4)           private int numerIndeksu;

              // Musimy uzyc EnumSet zamiast rodzajOsoby poniewaz chcemy
              miec
              // mozliwosc przechowywania kombinacji osob, np. Pracownik
              + Student
(5)           private EnumSet<OsobaRodzaj> rodzajOsoby =
              EnumSet.<OsobaRodzaj>of(OsobaRodzaj.Osoba);
      }
```

3-47 Implementacja dziedziczenia overlapping za pomocą grupowania

- (1). Wprowadziliśmy typ wyliczeniowy umożliwiający zapamiętanie, z jakim rodzajem osoby mamy do czynienia.

- (2). W jednej klasie zgrupowaliśmy atrybuty osoby (2), pracownika (3) oraz studenta (4).

- (5). Specjalny rodzaj kolekcji pełniący rolę dyskryminatora. Wstawiamy do niej odpowiednie typy wyliczeniowe, np. `OsobaRodzaj.Pracownik`, `OsobaRodzaj.Student`. Taka konfiguracja oznacza, że mamy do czynienia z pracującym studentem.

Jak możemy podsumować takie rozwiązanie? Na pewno odznacza się ono dużą prostotą realizacji oraz łatwością używania[18]. Równocześnie ma też pewne wady:

- Brak możliwości korzystania z konstrukcji związanych z dziedziczeniem, np. przesłanianie metod, polimorficzne wołanie metod itd.

- Niewykorzystywanie inwariantów należących do innej specjalizacji - mimo tego, że zajmują miejsce. Przykładowo: jeżeli obiekt naszej nowej klasy Osoba będzie opisywał pracującego studenta, to pozostałe atrybuty przechowujące pozostałe cechy nie będą wykorzystywane.

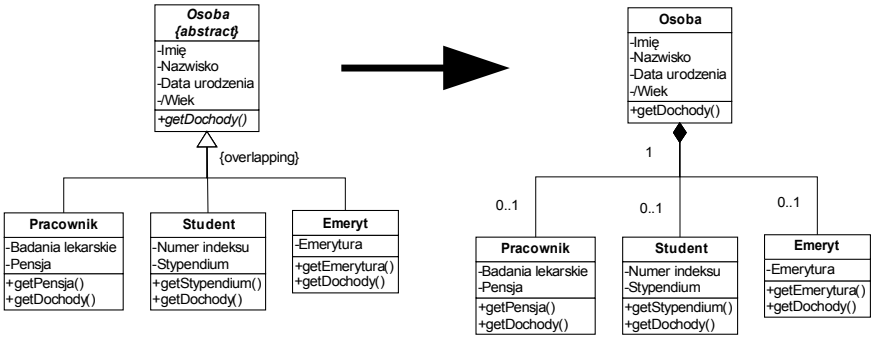
- Znaczne zagmatwanie diagramu klas. Wszystkie atrybuty i metody znajdą się w jednej klasie (może stać się ona dość duża), a asocjacje dotychczas dochodzące do łączonych klas zostaną podłączone do tej jednej.

3.3.3.2 Obejście dziedziczenia overlapping za pomocą agregacji lub kompozycji

Kolejnym sposobem obejścia braku dziedziczenia typu overlapping w popularnych językach programowania jest wykorzystanie agregacji lub kompozycji. Zwykle wybieramy kompozycję, ale w jakichś szczególnych przypadkach biznesowych może agregacja będzie lepsza.

Rysunek 3-80 przedstawia przekształcenie naszego poprzedniego przykładu (2-27 na stronie 55) do postaci pozbawionej dziedziczenia *overlapping*.

[18] Oczywiście te dwie cechy nie są tożsame, np. stworzyliśmy jakąś bibliotekę, która wymagała sporo pracy, czyli nie była prosta w realizacji. Ale miejmy nadzieję, że jej używanie będzie łatwe, proste, a może nawet przyjemne.



3-80 Obejście dziedziczenia overlapping za pomocą kompozycji

Jak widać, sprawa jest dość prosta. Zamiast dziedziczenia wstawiamy kompozycję, gdzie nadklasa staje się całością, a podklasy częściami. w efekcie pozbyliśmy się dziedziczenia dość radykalnie, bo nie występuje już ono pod żadną postacią.

Przy takim podejściu trzeba zdecydować, w jaki sposób potraktujemy asocjacje pierwotnie istniejące w podklasach. Problem polega na tym, że przy dziedziczeniu pokazywanie na np. instancję klasy `Pracownik` daje nam bezpośredni dostęp do inwariantów (atrybutów, metod, asocjacji) z nadklasy. W momencie gdy dziedziczenie zastępujemy kompozycją, takie możliwości znikają i trzeba je jakoś „symulować". Mamy dwa sposoby, co z nimi zrobić po przekształceniu. Asocjacje z podklas mogą pokazywać na:

- Całość. W takiej sytuacji możemy dodać odpowiednie metody, które ułatwią nam dostęp do inwariantów znajdujących się w klasach-częściach (podklasach). Będą to metody propagujące zapytanie od klasy-całości do klasy-części. Dzięki temu programista, który będzie z tego korzystał, nie będzie musiał nawigować przez klasę-całość.

- Część. W tym przypadku obiekty-części nie mogą być ukryte. Musi być do nich dostęp bezpośredni (nie przez obiekt-całość), co wiąże się z określonym sposobem kompozycji (patrz podrozdział 3.2.3.7 na stronie 153).

Oczywiście, kompozycję możemy zaimplementować na jeden z wcześniej omawianych sposobów, najlepiej korzystając z klasy `ObjectPlusPlus`. Dzięki temu zaoszczędzimy sobie sporo pracy.

Listing zawiera przykładową implementację dziedziczenia `overlapping` przy pomocy kompozycji oraz klasy `ObjectPlusPlus`. Ważniejsze miejsca kodu:

- (1). Standardowe dane biznesowe charakterystyczne dla osoby.
- (2). Konstruktor tworzący instancję klasy osoba.
- (3). Ponieważ korzystamy z klasy `ObjectPlusPlus`, obowiązkowo musimy wywołać konstruktor z nadklasy.
- (4). Podstawienie przekazanych danych biznesowych.
- (5). Konstruktor tworzący osobę będącą pracownikiem.
- (6). Wywołanie metody (7) tworzącej obiekt-część opisujący pracownika oraz powiązanie w ramach kompozycji.
- (8). Tworzenie obiektu-części opisującego pracownika.
- (9). Dodanie go jako części do istniejącej całości. Korzysta z metody udostępnianej przez `ObjectPlusPlus`.
- (10). Metoda analogiczna do (7), ale tworząca emeryta.
- (11). Nazwy ról wykorzystywane przy tworzeniu powiązań w ramach kompozycji.
- (12). Metoda sprawdzająca, czy pracownik ma badania lekarskie.
- (13). Zasadnicza część metody jest wykonywana wewnątrz bloku wyłapującego wyjątki.
- (14). Na podstawie nazwy roli próbujemy odzyskać powiązane obiekty. Tak naprawdę powinien być tylko jeden.
- (15). W przypadku gdy nie istnieje dane powiązanie, co dla nas znaczy, że dana osoba nie jest pracownikiem, otrzymujemy wyjątek. W prawdziwej implementacji należałoby jakoś rozróżniać różne rodzaje błędów, np. poprzez wprowadzenie różnych klas wyjątków.

- (16). Metoda analogiczna do (12), ale zwracająca informacje specyficzne dla studenta.

```
     public class Osoba extends ObjectPlusPlus {
(1)         private String imie;
            private String nazwisko;
            private Date dataUrodzenia;

(2)         public Osoba(String imie, String nazwisko, Date
               dataUrodzenia) {
(3)              super();// Wymagane przez ObjectPlusPlus

(4)              this.imie = imie;
                 this.nazwisko = nazwisko;
                 this.dataUrodzenia = dataUrodzenia;
            }

(5)         public Osoba(String imie, String nazwisko, Date
               dataUrodzenia, boolean badaniaLekarskie) {
                 super();// Wymagane przez ObjectPlusPlus

                 this.imie = imie;
                 this.nazwisko = nazwisko;
                 this.dataUrodzenia = dataUrodzenia;

                 // "Zmienia" osobe w pracownika
                 dodajPracownika(badaniaLekarskie);
(6)          }

            public void dodajPracownika(boolean badaniaLekarskie) {
(7)              // Tworzymy czesc opisujaca Pracownika
                 Pracownik p = new Pracownik(badaniaLekarskie);
(8)
                 // Dodanie pracownika jako powiazania
                 // (nie korzystamy z dodawania jako czesci
                 w agregacji aby uniknac wyjatku)
                 // Korzystamy z metody dostarczanej przez
                 ObjectPlusPlus
                 this.dodajPowiazanie(nazwaRoliPracownik,
(9)                 "generalizacja", p);
            }

            public void dodajEmeryta() throws Exception {
(10)             // Tworzymy czesc opisujaca Pracownika
                 Emeryt e = new Emeryt();

                 // Dodanie emeryta jako powiazania
                 // (nie korzystamy z dodawania jako czesci
                 w agregacji aby uniknac wyjatku)
                 // Korzystamy z metody dostarczanej przez
                 ObjectPlusPlus
                 this.dodajPowiazanie(nazwaRoliEmeryt,
                    "generalizacja", e);
            }
```

```
            private static String nazwaRoliPracownik =
(11)           "specjalizacjaPracownik";
            private static String nazwaRoliEmeryt =
               "specjalizacjaEmeryt";

            public boolean czyMaBadaniaLekarskie() throws Exception {
(12)               // daj obiekt opisujacy pracownika
                   try {
(13)                      ObjectPlusPlus[] obj =
(14)                         this.dajPowiazania(nazwaRoliPracownik);
                          return ((Pracownik)
                             obj[0]).isBadaniaLekarskie();
                   } catch (Exception e) {
(15)                      // Prawdopodobnie dostalismy wyjatek
                          mowiacy, ze taka rola nie istnieje
                          // (docelowo powinny to byc rozne klasy
                          wyjatkow)
                          throw new Exception("Obiekt nie jest
                             Pracownikiem!");
                   }
            }
(16)
            public int dajNumerIndeksu() throws Exception {
                   // daj obiekt opisujacy pracownika
                   try {
                          ObjectPlusPlus[] obj =
                          this.dajPowiazania(nazwaRoliStudent);
                          return ((Student)
                          obj[0]).getNumerIndeksu();
                   } catch (Exception e) {
                          // Prawdopodobnie dostalismy wyjatek
                          mowiacy, ze taka rola nie istnieje
                          // (docelowo powinny to byc rozne klasy
                          wyjatkow)
                          throw new Exception("Obiekt nie jest
                             Studentem!");
                   }
            }
      }
```

3-48 Implementacja dziedziczenia overlapping za pomocą kompozycji

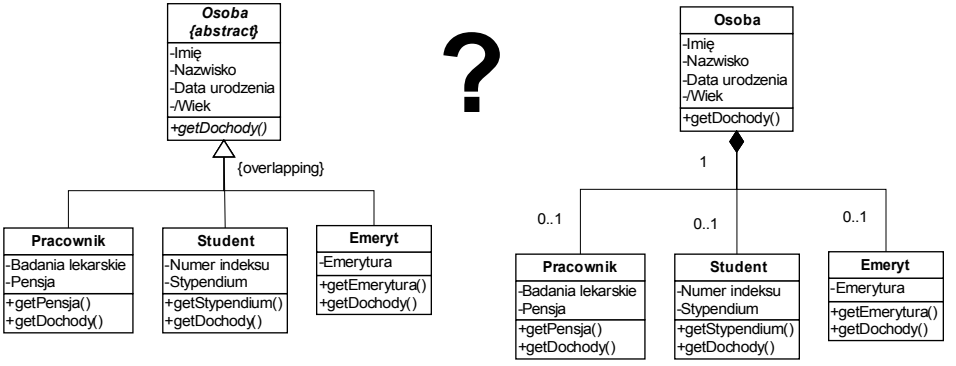
Niewątpliwą zaletą tej metody jest fakt, iż dzięki rozbiciu funkcjonalności na różne klasy korzystamy tylko z tych inwariantów, które są nam rzeczywiście potrzebne. W porównaniu z poprzednim sposobem, gdzie w jednej klasie były zawsze wszystkie atrybutu czy metody, jest to na pewno użyteczna cecha. W zależności od zaimplementowanych metod pomocniczych można też mówić o ewentualnej łatwości użycia.

Zasadniczą wadą tego sposobu, która zresztą występowała też w poprzednim podejściu, jest brak konstrukcji związanych z dziedziczeniem, ta-

kich jak przesłanianie metod czy ich polimorficzne wołanie. No i oczywiście, większy nieporządek na diagramie. Ktoś, kto pierwszy raz spojrzy i zobaczymy kompozycje, to nie będzie kojarzył tego związku z zależnościami opisującymi zwykle dziedziczenie.

3.3.3.3 Polimorfizm w dziedziczeniu overlapping

Przy okazji dziedziczenia typu overlapping może wystąpić pewien problem związany z wołaniem metod. Spójrzmy na rysunek 3-81.



3-81 Ilustracja problemu związanego z wywoływaniem metod w przypadku dziedziczenia typu overlapping

Która wersja metody `getDochody()` (z której klasy) powinna być wywołana tak, aby dobrze policzyła dochody osoby będącej i pracownikiem, i studentem? Chyba żadna… Rozwiązaniem jest stworzenie nowej metody, która w zależności od rodzajów obiektów uwzględni odpowiednie dochody. Przykładowy kod pokazany jest na listingu 3-49. Jej kod jest dość prosty:

- (1). Na początku dochody osoby wynosza 0.
- (2). Sprawdzamy, czy dana osoba jest pracownikiem. Jeżeli tak, to pobieramy obiekt-część, który go opisuje (3).
- (4). Dochody z „części pracowniczej" dodajemy do ogólnych dochodów osoby. Podobnie postępujemy z pozostałymi obiektami-częściami opisującymi studenta oraz emeryta.

```
public float getDochody() throws Exception {
```

```
(1)             float dochody = 0.0f;

(2)             if(this.czySaPowiazania(nazwaRoliPracownik)) {
                        // Jest pracownikiem. Znajdz obiekt opisujacy
                        pracownika.
(3)                     ObjectPlusPlus[] obj =
                           this.dajPowiazania(nazwaRoliPracownik);
                        // ==> dolicz dochody pracownika
(4)                     dochody += ((Pracownik) obj[0]).getDochody();
                }

                if(this.czySaPowiazania(nazwaRoliStudent)) {
                        // Jest studentem. Znajdz obiekt opisujacy studenta.
                        ObjectPlusPlus[] obj =
                           this.dajPowiazania(nazwaRoliStudent);
                        // ==> dolicz dochody studenta
                        dochody += ((Student) obj[0]).getDochody();
                }

                if(this.czySaPowiazania(nazwaRoliEmeryt)) {
                        // Jest emerytem. Znajdz obiekt opisujacy emeryta.
                        // [...]
                }
                return dochody;
        }
```

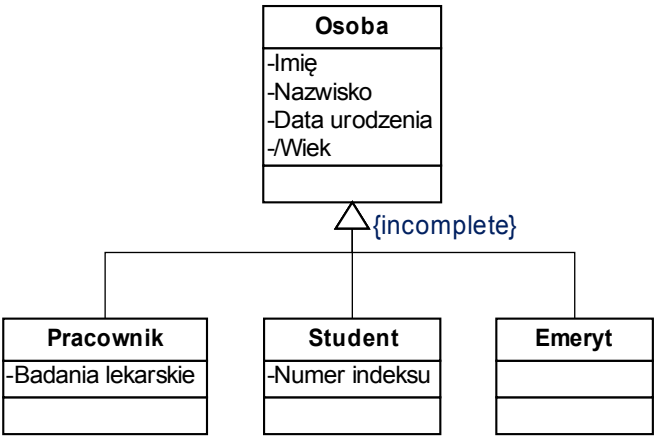
3-49 Kod przykładowej metody prawidłowo zwracającej dochody osób w przypadku dziedziczenia overlapping

Jak widać, powyższy sposób postępowania ma niewiele wspólnego z wysokim poziomem abstrakcji, który tak lubimy w obiektowości. Chociaż z drugiej strony, ktoś po prostu pyta o dochody i nie musi sprawdzać, jak one są liczone.

3.3.4 Dziedziczenie kompletne oraz niekompletne

Dziedziczenie niekompletne (*incomplete*) było omówione przy okazji diagramu klas wypożyczalni, a dokładniej rysunek 2-49 (strona 76). Spójrzmy na kolejny przykład – rysunek 3-82. Zastanówmy się - co ten rodzaj dziedziczenia znaczy dla diagramu? Czy fakt, że być może w przyszłości dodamy nowe klasy, znaczy coś dla implementacji? Raczej nie. Może tylko, że musimy zapewnić możliwość dalszego dziedziczenia, np. nie powinniśmy używać słowa `final`, które w języku Java nie pozwala na dziedziczenie z klasy zadeklarowanej z użyciem tego wyrażenia. Ale ta sytuacja zdarza się dość rzadko – ilu z Was go używało?

W związku z tym, w przypadku implementacji raczej ignorujemy te oznaczenia.



3-82 Diagram ilustrujący dziedziczenie niekompletne

3.3.5 Dziedziczenie wielokrotne (wielodziedziczenie)

Dziedziczenie wielokrotne, zwane także wielodziedziczeniem, omówiliśmy od strony koncepcyjnej w podrozdziale 2.4.4.3 na stronie 53. Teraz zajmiemy się jego przedyskutowaniem od strony języków programowania.

Już na wstępie możemy stwierdzić, że dziedziczenie wielokrotne:

- występuje w języku C++. W przypadku konfliktu nazw, o którym pisaliśmy przy okazji teoretycznego omówienia, używamy operatora zakresu. Dzięki temu kompilator wie, którego inwariantu użyć.
- nie występuje w języku Java ani w MS C#.

Jak możemy je zaimplementować? Tradycyjnie sposobów zmierzenia się z tym problemem jest kilka:

- Możemy wykorzystać podejścia, które zastosowaliśmy przy okazji dziedziczenia *overlapping* (podrozdział 3.3.3 na stronie 181):
    - Pozostawiamy jedną hierarchię dziedziczenia, a pozostałe grupujemy w nadklasie.
    - Analogicznie jak powyżej, ale pozostałe hierarchie implementujemy za pomocą kompozycji lub agregacji. Podstawo-

we pytanie, jakie się nasuwa, dotyczy kryteriów wyboru: którą hierarchię dziedziczenia zostawić, a którą zastąpić? Odpowiedź jest dość intuicyjna: gałąź, gdzie więcej wykorzystujemy konstrukcji związanych z dziedziczeniem (przesłanianie, polimorficzne wołanie metod itp.), zostawiamy, pozostałe „konwertujemy".

- Nowym podejściem, którego jak dotąd nie używaliśmy, jest zastosowanie interfejsów dostępnych zarówno w języku Java, jak i MS C#. Dalszą część tego rozdziału poświęcimy właśnie opisowi tego podejścia.

Dobra wiadomość jest taka, że klasa może implementować dowolną liczbę interfejsów. Natomiast, ze względu na ich ograniczenia, nie mamy możliwości umieszczenia w nich atrybutów[19], więc korzystamy tylko z metod. Częściowo ten problem możemy rozwiązać używając tzw. getterów i setterów, czyli metod z przedrostkiem get/set, a służących do pobierania wartości atrybutu oraz zmieniania jego wartości.

Niestety dość poważną wadą wykorzystywania interfejsów jest konieczność wielokrotnego implementowania takich samych metod i do tego w ten sam sposób. Taka sytuacja zachodzi, gdy w różnych klasach mamy np. metodę `getNazwisko()`, która po prostu zwraca jego wartość. Częściowo ten problem rozwiązują nowoczesne środowiska programistyczne (IDE), które potrafią automatycznie wygenerować odpowiednie zestawy setterów i getterów dla wskazanych atrybutów. Mimo wszystko jest to tylko częściowe rozwiązanie, ponieważ:

- Dotyczy specyficznego rodzaju metod i nie pomaga w „normalnych" metodach,

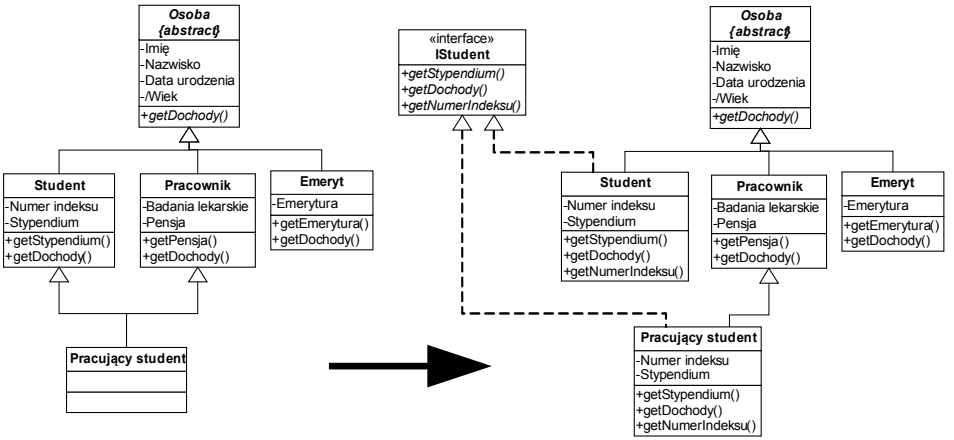
- W przypadku dokonywania modyfikacji musimy ręcznie poprawiać każdą implementację.

Innym podejściem do rozwiązania problemu wielokrotnej implementacji metod mogą być rozwiązania wykorzystujące wzorce projektowe, np. delegacji czy polecenia. W skrócie polega to na tym, że implementując taką samą metodę w różnych klasach, zamiast bezpośrednio powielać kod w jej

[19] Teoretycznie w interfejsie można umieścić atrybuty, ale z definicji stają się one `public` oraz `final`. Czyli tak naprawdę pełnią rolę stałych służących np. do zdywersyfikowania rodzaju jakiegoś elementu.

ciele, delegujemy jej wykonanie (z jej wnętrza w wielu klasach) do jakiegoś (tego samego) obiektu.

Spójrzmy na rysunek 3-83 przedstawiający przekształcenie wielodziedziczenia z wykorzystaniem interfejsów (w UML interfejsy zaznacza się podobnie do dziedziczenia, ale z przerywaną linią). Po zamianie klasa `Pracujący Student` dziedziczy z klasy `Pracownik` oraz implementuje interfejs `IStudent`[20]. Interfejs ten jest również implementowany przez klasę `Student`. Mamy tu też do czynienia z sytuacją, którą opisywaliśmy poprzednio. Metody `getStypendium()` oraz `getNumerIndeksu()` będą prawdopodobnie miały identyczne ciała w obydwóch implementacjach. Natomiast `getDochody()` już nie – w przypadku studenta po prostu zwróci wartość stypendium, ale dla pracownika uwzględni jeszcze jego pensję.



3-83 Przekształcenie wielodziedziczenia z wykorzystaniem interfejsów

Odpowiedni kod implementujący klasę `PracujacyStudent` jest pokazany na listingu 3-50. Komentarz do ważniejszych miejsc zamieszczonego programu:

- (1). Deklaracja interfejsu `IStudent`. Jak widzimy, zawiera tylko nagłówki metod. Ich ciała zostaną dodane w klasach, które go implementują.

[20] w niektórych językach programowania, a może raczej niektórzy programiści, zaczynają nazwy interfejsów od litery „I”. Dzięki temu już na pierwszy rzut oka widać, z czym mamy do czynienia oraz nie występuje problem konfliktu nazw (gdybyśmy chcieli mieć klasę oraz interfejs o nazwie `Student`).

- (2). Klasa `PracujacyStudent` dziedzicząca z `Pracownik` oraz implementująca interfejs `IStudent` (słowo kluczowe `implements`).
- (3). Konstruktor klasy pobierający odpowiednie parametry biznesowe.
- (4). Wywołanie konstruktora z nadklasy.
- (5). Implementacja metody z interfejsu. Prawdopodobnie identyczna z tą, która jest w klasie `Student`.
- (6). Implementacja metody z interfejsu, ale w innej wersji niż występująca w klasie `Student`.

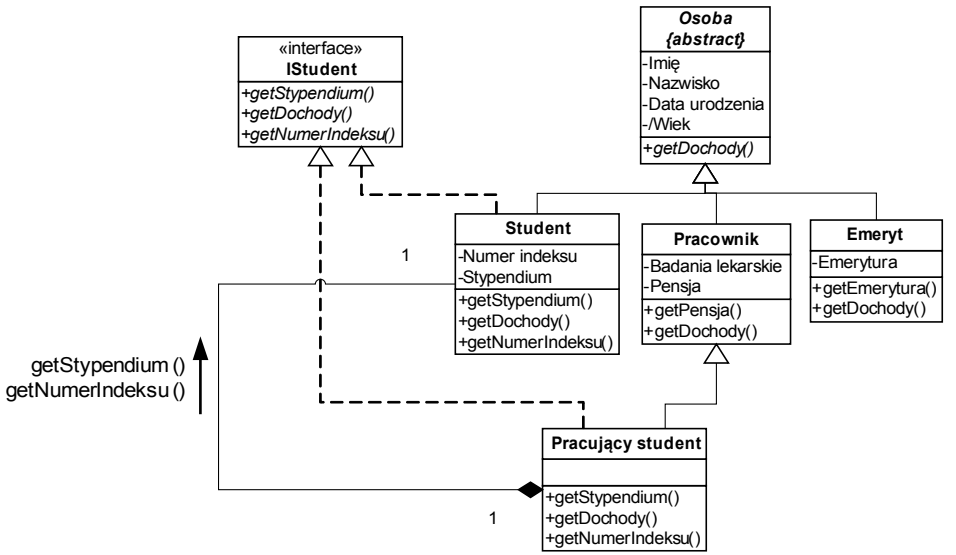
```
(1)   public interface IStudent {
              public abstract float getDochody();
              public abstract float getStypendium();
              public abstract void setStypendium(float stypendium);
              public abstract int getNumerIndeksu();
      }

(2)   public class PracujacyStudent extends Pracownik implements
         IStudent {
              private int numerIndeksu;
              private float stypendium;

(3)           public PracujacyStudent(String imie, String nazwisko,
              Date
                 dataUrodzenia, boolean badaniaLekarskie, float pensja,
                 int numerIndeksu, float stypendium) {
(4)                   super(imie, nazwisko, dataUrodzenia,
                         badaniaLekarskie, pensja);

                      this.numerIndeksu = numerIndeksu;
                      this.stypendium = stypendium;
              }
              public float getStypendium() {
(5)                   return stypendium;
              }
              public float getDochody() {
(6)                   return super.getDochody() + getStypendium();
              }
              public int getNumerIndeksu() {
                      return numerIndeksu;
              }
      }
```

3-50 Implementacja wielodziedziczenia za pomocą interfejsów

Wspominaliśmy o problemie związanym z wielokrotnym powielaniem funkcjonalności w klasach implementujących ten sam interfejs. Czy da się temu zaradzić? Oczywiście – też o tym już pisaliśmy. Częściowe rozwiązanie problemu wielokrotnej implementacji tych samych metod może wyglądać np. tak (diagram klas z rysunku 3-84):

- Klasa `PracujacyStudent` dziedziczy funkcjonalność pracownika i deleguje funkcjonalność studenta do podłączonego obiektu.

- Innymi słowy: opakowuje funkcjonalność klasy Student. Jeżeli PracujacyStudent otrzyma nakaz wywołania metody `getStypendium()` lub `getNumerIndeksu()`, to propaguje go do podłączonego obiektu klasy `Student`.



3-84 Delegowanie funkcjonalności do innej klasy

Jedna z możliwych implementacji dla tego diagramu jest przedstawiona na listingu 3-51. Komentarze:

```
      public class PracujacyStudent extends Pracownik implements
      IStudent {
(1)           Student student;
```

```
(2)          public PracujacyStudent(String imie, String nazwisko,
             Date
                dataUrodzenia, boolean badaniaLekarskie, float pensja,
(3)             int numerIndeksu, float stypendium) {
                     super(imie, nazwisko, dataUrodzenia,
                        badaniaLekarskie, pensja);
(4)
                     student = new Student(imie, nazwisko,
                        dataUrodzenia, numerIndeksu, stypendium);
             }
             public float getStypendium() {
(5)                  return student.getStypendium();
             }
             public void setStypendium(float stypendium) {
                     student.setStypendium(stypendium);
             }
             public float getDochody() {
                     return super.getDochody() + getStypendium();
(6)          }
             public int getNumerIndeksu() {
                     return student.getNumerIndeksu();
             }
     }
```

3-51 Implementacja propagacji operacji w ramach wykorzystania interfejsów

- (1). Referencja do obiektu, do którego będziemy przekierowywali odwołania związane z funkcjonalnością studenta.
- (2). Konstruktor.
- (3). Wywołanie konstruktora z nadklasy.
- (4). Utworzenie obiektu studenta, który będzie wykorzystywany do zapewnienia funkcjonalności „studenckiej”.
- (5). Metoda z interfejsu. Jak widać, wywołujemy ją z obiektu studenta i zwracamy wynik.
- (6). Metoda z interfejsu zwracająca dochody. Najpierw wołamy metodę z nadklasy (aby podała nam dochody pracownika), a następnie dodajemy wysokość stypendium.

Pewien niepokój może budzić pamiętanie niektórych atrybutów dwa razy (np. imię, nazwisko): raz w klasie `PracujacyStudent`, a drugi raz w podłączonym obiekcie klasy `Student`. Jak możemy temu zaradzić? Wyjścia są właściwie dwa: modyfikacja klasy `Student` (i/lub hierarchii dziedzi-

czenia) lub przekazywanie do obiektu klasy `Student` jakichś specjalnych wartości, np. `null`'i.

3.3.6 Dziedziczenie wieloaspektowe

Dziedziczenie wieloaspektowe, omawialiśmy od strony koncepcyjnej w podrozdziale 2.4.4.5 na stronie 55 (m.in. rysunek 2-28). Ten podrozdział będzie poświęcony przeanalizowaniu tej koncepcji w kontekście popularnych języków programowania.

Na wstępnie mam do przekazania złą wiadomość polegającą na tym, że ten rodzaj generalizacji nie występuje w żadnym z języków programowania, które omawiamy w tej książce. W związku z tym musimy je jakoś zaimplementować, korzystając np. z poniższych sugestii:

- Jeden aspekt dziedziczymy używając wbudowanych (prostych) mechanizmów dziedziczenia danego języka programowania.

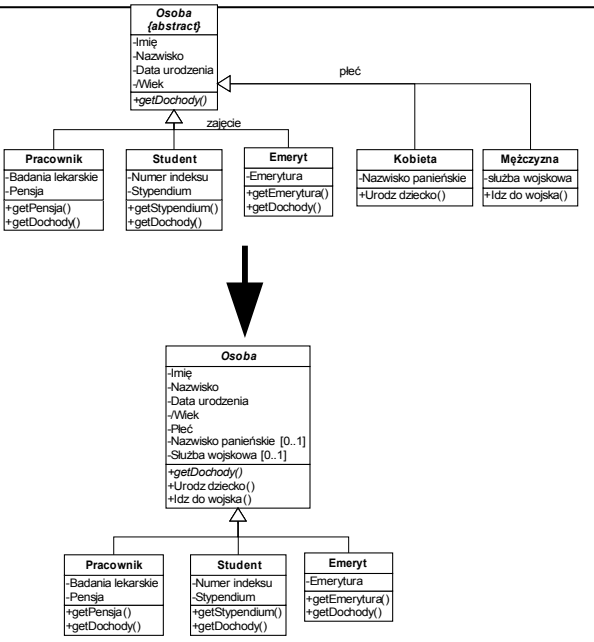
- Pozostałe aspekty implementujemy za pomocą jednego z wcześniej omawianych sposobów.

- Możemy również spróbować usunąć część hierarchii lub inaczej mówiąc, zgrupować w jednej klasie, np. dodając flagi do głównej klasy. Tak możemy postąpić, gdy nie potrzebujemy przechowywać różnych informacji w zależności od wartości tejże flagi. Innymi słowy, flaga ta jest po prostu swego rodzaju dyskryminatorem.

Od razu nasuwa się pytanie: który aspekt powinniśmy dziedziczyć, a który zamodelować w inny sposób? Podobnie jak w przypadku dziedziczenia wielokrotnego pozostawiamy ten aspekt, gdzie:

- występuje przesłanianie metod, polimorficzne wołanie,

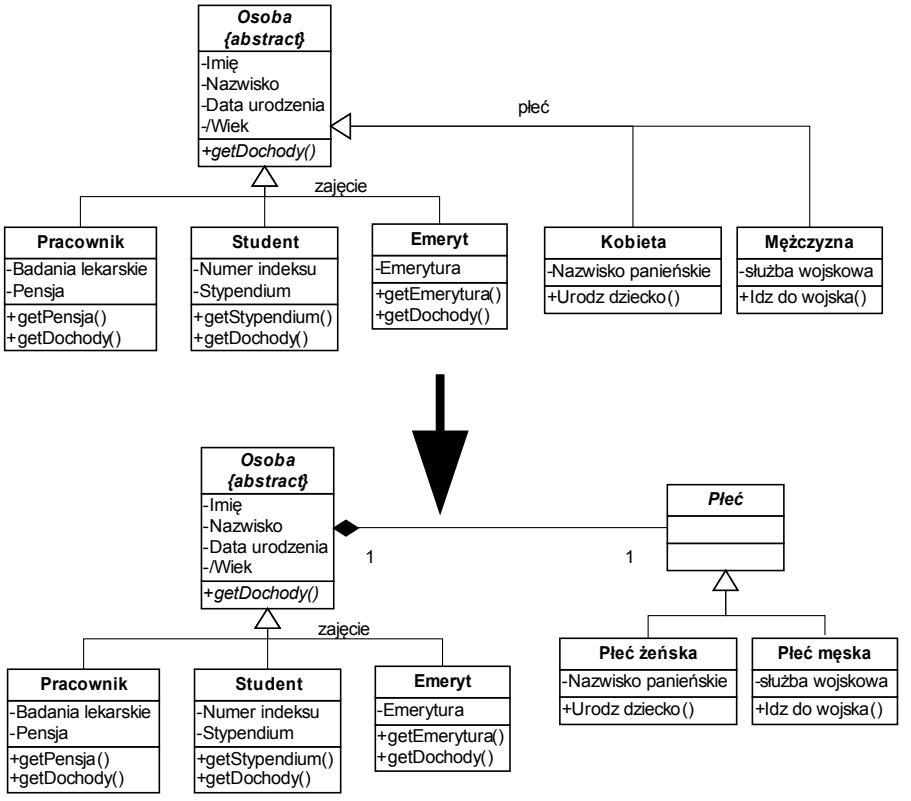
- jest większe zróżnicowanie atrybutów w poszczególnych podklasach. Innymi słowy – najbardziej skomplikowaną/rozbudowaną hierarchię.



3-85 Obejście problemu dziedziczenia wieloaspektowego z pomocą grupowania atrybutów

W niektórych sytuacjach, gdy nie przechowujemy informacji specyficznych dla danego aspektu, a tylko informację o rodzaju obiektu, możemy dziedziczenie zastąpić np. flagą umieszczoną w nadklasie. Podobnie możemy postąpić, gdy specyficznych informacji jest mało. W takiej sytuacji może lepiej jest umieścić jeden lub dwa atrybuty w nadklasie, nawet gdy one nie będą zawsze wykorzystywane, niż komplikować model za pomocą np. kompozycji.

Spójrzmy na przykładowe rozwiązania. Rysunek 3-85 przedstawia jeden z możliwych sposób rozwiązania problemu z dziedziczeniem wieloaspektowym. Pomysł polega na umieszczeniu atrybutów i metod ze zlikwidowanego aspektu w nadklasie. Warto zwrócić uwagę na oznaczenia nowych atrybutów. I nazwisko panieńskie, i stosunek do służby wojskowej są oznaczone jako opcjonalne. Jest tak dlatego, że w zależności od płci danej osoby któryś z nich nie będzie miał wartości.



3-86 Obejście problemu dziedziczenia wieloaspektowego z pomocą kompozycji

Inne rozwiązanie dla tego samego diagramu początkowego zostało przedstawione na rysunku 3-86. Zamiast podklas kobieta oraz mężczyzna mamy kompozycję oraz klasy płeć żeńska oraz męska. Zwróćmy uwagę, że zmianie uległa również semantyka klas. Płeć nie jest osobą (jak klasa poprzednio), a tylko pewnym zestawem informacji o niej. Oczywiście, można dyskutować, czy ta zmiana była potrzebna. Moim zdaniem tak, ale gdyby ktoś tego nie zrobił, to nie uważałbym tego za jakiś błąd. Po prostu, w tego typu przypadkach, ilu projektantów tyle rozwiązań. Warto też zwrócić uwagę, że dzięki liczności kompozycji równiej 1 zawsze będziemy połączeni z dokładnie jednym obiektem przechowującym tego typu informacje. I w przeciwieństwie do poprzedniego podejścia, tym razem żadne atrybutu niepotrzebnie nie zajmują miejsca. Ceną za to jest pewnie trochę bardziej

skomplikowany diagram, a co przeważnie za tym idzie, trudniejsza implementacja.

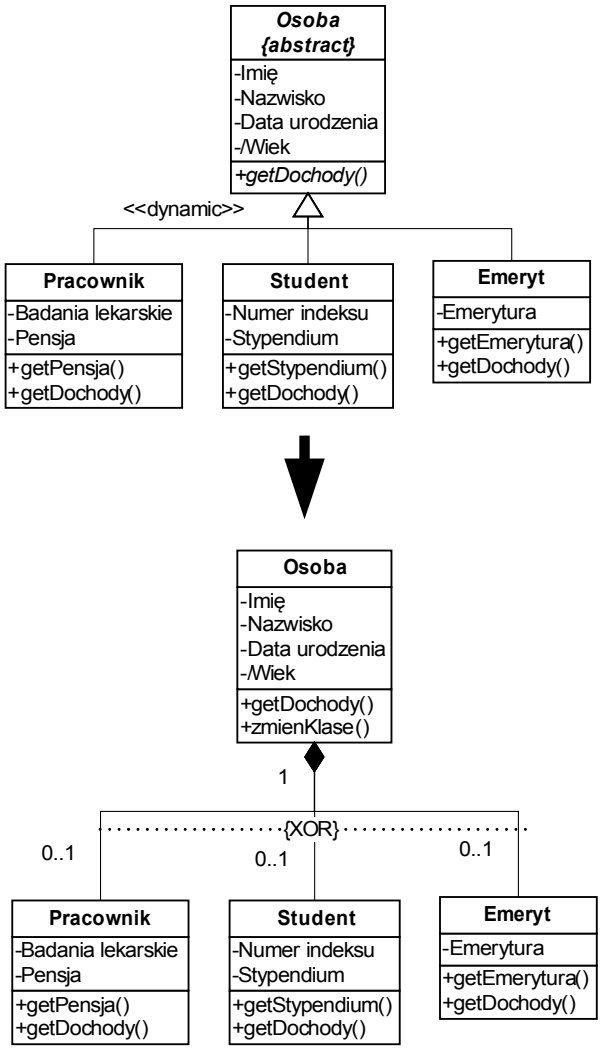
3.3.7 Dziedziczenie dynamiczne

Dziedziczenie dynamiczne (opis w podrozdziale 2.4.4.6 na stronie 57) zamyka nasz podrozdział poświęcony implementacji różnych związków generalizacji/specjalizacji. Podobnie jak większość swoich poprzedników nie występuje ani w C++, ani w C#, ani też w języku Java. Tradycyjnie więc będziemy musieli je jakoś zaimplementować ręcznie, stosując różne techniki. Zaliczają się do nich:

- Umieszczenie wszystkich inwariantów w nadklasie i dodanie dyskryminatora. Sposób ten stosowaliśmy przy okazji innych rodzajów dziedziczenia, więc nie będziemy go tutaj szczegółowo omawiać. Właściwie to podejście ma tylko jedną zaletę: łatwość implementacji. Reszta to już chyba same wady.

- Wykorzystanie agregacji/kompozycji z ograniczeniem *{xor}* (przykład jest przedstawiony na rysunku 3-87). Wykorzystujemy kod stworzony przy okazji dziedziczenia *overlapping*. Dodatkowo umieszczamy metody ułatwiające „zmianę klasy". Ich zadaniem jest usunięcie dotychczasowego obiektu-części (odpowiadającego za „dynamiczny aspekt" danego obiektu) i stworzenie nowego. Dla omawianego przykładu mogłoby to być przekształcenie pracownika w emeryta czy studenta w pracownika. Metody te umieszczamy w obiekcie-całości (klasa `Osoba`).

- „Sprytne" kopiowanie obiektów. Takiego rozwiązania jeszcze nie prezentowaliśmy, więc omówimy je trochę dokładniej poczynając od następnego akapitu.



3-87 Obejście dziedziczenia dynamicznego za pomocą kompozycji

Pomysł polega na zastąpieniu starego obiektu należącego do pierwotnej klasy nowym, który jest instancją nowej klasy – tej „po przemianie". W tym celu w każdej z podklas tworzymy dodatkowe konstruktory. Każdy z nich przyjmuje jako parametr referencję do obiektu nadklasy (plus ewentualnie dodatkowe informacje specyficzne dla określonej klasy). Informacje wspólne dla wszystkich obiektów znajdujących się na danym poziomie hierarchii

są kopiowane z wnętrza otrzymanego obiektu do wnętrza nowo tworzonego obiektu. Czyli dla naszego przykładu z 3-87, zmieniając „stary" obiekt klasy Student w „nowy" obiekt klasy Pracownik, skopiujemy imię, nazwisko oraz datę urodzenia. Pozostałe informacje specyficzne dla studenta (nr indeksu, stypendium) giną. Jeżeli są potrzebne, to można je jakoś zachować, ale wtedy to nie będzie „czyste" dziedziczenie dynamiczne.

Na pierwszy rzut oka to rozwiązanie wygląda bardzo dobrze. Jest dość łatwe w implementacji, no i „zadziała". Niestety nie do końca. Problemem może być uaktualnienie odpowiednich referencji prowadzących do „starego" obiektu (czyli powiązań) tak, aby pokazywały na nowy obiekt. „Odpowiednie" referencje oznaczają te, które są wspólne dla „starej" i „nowej" klasy. Pozostałe referencje (te specyficzne dla „starej" klasy) „przepadają" – podobnie jak wartości atrybutów.

W przypadku korzystania z klasy `ObjectPlusPlus` rozwiązanie tego problemu jest dużo łatwiejsze. Jest tak dlatego, że posiadamy informacje o obiektach, które na „nas" pokazują – bo wszystkie powiązania w `ObjectPlusPlus` są dwustronne. Trzeba również pamiętać o zadbaniu o ekstensję, czyli usunięciu z niej „starych" obiektów i dodaniu „nowych".

Przykładowy kod implementujący część tej funkcjonalności jest pokazany na listingu 3-52. Jak widzimy, w klasie `Pracownik` mamy konstruktor (1), który pobiera instancję klasy `Osoba` (a właściwie jej podklasę, bo `Osoba` jest klasą abstrakcyjną) oraz parametry specyficzne dla pracownika. Ktoś mógłby zapytać: a gdzie odbywa się to kopiowanie „starych" parametrów, o którym pisaliśmy? Dzięki wołaniu konstruktora z nadklasy (2) nie musimy robić tego ręcznie. Następnie zapamiętywane są informacje charakterystyczna dla pracownika (3).

```
public abstract class Osoba {
        protected String imie;
        protected String nazwisko;
        protected Date dataUrodzenia;

        public Osoba(String imie, String nazwisko, Date
           dataUrodzenia) {
                super();
                this.imie = imie;
                this.nazwisko = nazwisko;
                this.dataUrodzenia = dataUrodzenia;
        }

        // […]
        public String toString() {
                return this.getClass().getSimpleName() + ": " +
                imie + " " + nazwisko;
```

```
        }
    }

    public class Pracownik extends Osoba {
        private boolean badaniaLekarskie;
        private float pensja;
        // […]

(1)     public Pracownik(Osoba poprzedniaOsoba, boolean
        badaniaLekarskie, float pensja) {
            // Skopiowanie "starych" danych
(2)         super(poprzedniaOsoba.imie,
            poprzedniaOsoba.nazwisko,
                poprzedniaOsoba.dataUrodzenia);

            // Zapamietanie nowych
(3)         this.badaniaLekarskie = badaniaLekarskie;
            this.pensja = pensja;
        }
```

3-52 Implementacja „sprytnego" kopiowania obiektów jako sposobu obejścia dziedziczenia dynamicznego

A w jaki sposób można tego używać? Przykład jest na listingu 3-53. Najpierw tworzymy obiekt opisujący studenta (1), następnie zamieniamy go w pracownika (2), a później w emeryta (3). Takie życie w pigułce. Efekt działania programu jest na konsoli 3-8.

```
(1)  // Tworzymy studenta
     Osoba o1  = new Student("Jan", "Kowalski", new Date(), 1212,
     2000.0f);
     System.out.println(o1);

(2)  // Tworzymy pracownika na podstawie studenta
     o1 = new Pracownik(o1, true, 4000.0f);
     System.out.println(o1);

     // Tworzymy emeryta na podstawie pracownika
(3)  o1 = new Emeryt(o1, 3000.0f);
     System.out.println(o1);
```

3-53 Przykład wykorzystania obejścia dziedziczenia dynamicznego

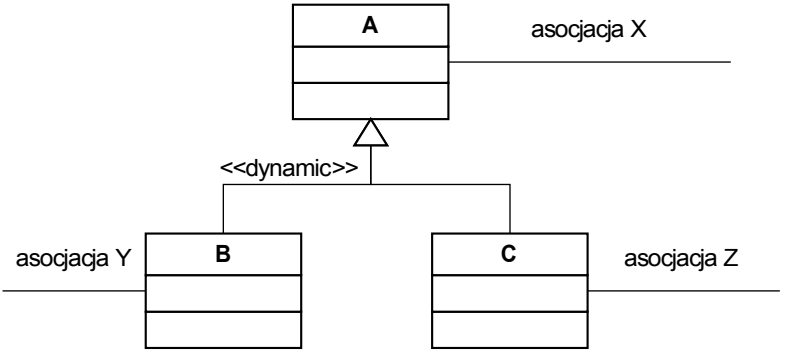
```
Student: Jan Kowalski
Pracownik: Jan Kowalski
Emeryt: Jan Kowalski
```

3-8 Wynik działania programu z listingu 3-53

A co z uaktualnianiem powiązań pokazujących na „stary” obiekt? Zwykle tę część daje studentom jako pracę domową. Nie powinno to być trudne, ponieważ w przypadku korzystania z klasy `ObjectPlusPlus` większość wymaganych informacji jest już w systemie. Oprócz samego skopiowania powiązań trzeba zadbać o kilka spraw:

- Ze „starego” obiektu powinniśmy skopiować tylko powiązania należące do „nadklasy” (np. A) – rysunek 3-88. Nie kopiujemy powiązań należących do jego asocjacji (np. B, C).

- Aby poprawnie uaktualnić powiązania pokazujące na „nowy” obiekt, potrzebujemy informacji o wzajemnych powiązaniach ról. Czyli która *nazwaRoli* odpowiada *odwrotnaNazwaRoli*. Teoretycznie możemy zastąpić wszystkie wystąpienia (we wszystkich rolach) `obiektDocelowy` pokazujące na „stary” obiekt, ale część z tych uaktualnień jest niepotrzebna, ponieważ część z powiązań „przepada” (patrz poprzednia strona). Innymi słowy, gdy jest kilka powiązań (należących do różnych asocjacji/ról) pomiędzy tymi samymi obiektami, nie jesteśmy w stanie stwierdzić, które z nich tworzą pary.



3-88 Ilustracja dla potrzeb „zadania domowego”

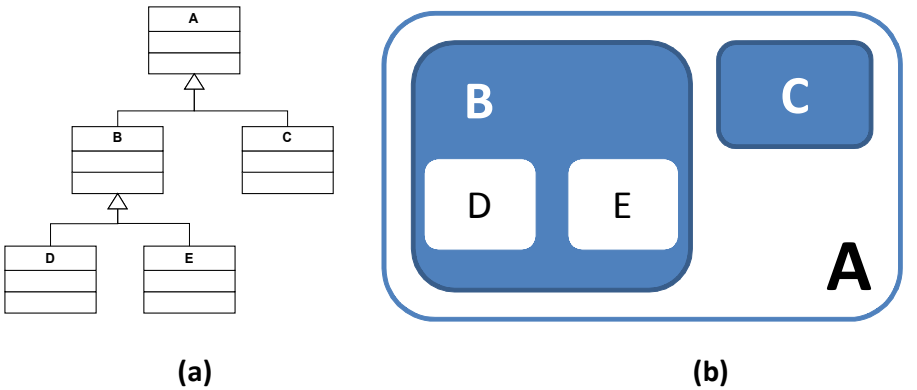
Tak więc proponuję samodzielne zaprojektowanie oraz opracowanie rozwiązania. Dzięki temu każdy będzie mógł sprawdzić, na ile zrozumiał omawiany materiał.

3.3.8 Dziedziczenie a ekstensja klasy

Pod koniec podrozdziału 2.4.4.2 (strona 49) zastanawialiśmy się, czy i w jaki sposób powiązać ze sobą ekstensje klas występujących w tej samej hierarchii dziedziczenia. Jak wspomnieliśmy, zdania naukowców zajmujących się obiektowością są podzielone. Myślę, że najbardziej przydatnym podejściem będzie włączanie ekstensji podklas do nadklas (ale nie odwrotnie).

Spójrzmy na rysunek 3-89. Część (a) przedstawia przykładową hierarchię klas. Część (b) pokazuje zawieranie się poszczególnych ekstensji. Widzimy, że ekstensja klasy a (najwyżej w hierarchii) zawiera wszystkie ekstensje jej podklas, a ekstensja klasy B zawiera ekstensje klas C oraz D. Innymi słowy, w ekstensji:

- klasy a możemy znaleźć obiekty należące do klas: A, B, C, D, E,
- klasy B instancje klas B, D, E.



3-89 Hierarchia klas (a) oraz zawieranie się jej ekstensji (b)

Po co nam to jest potrzebne? Mówiliśmy już kiedyś, ale na wszelki wypadek powtórzmy na przykładzie naszej wypożyczalni. Mamy klasę Klient, z której dziedziczy klient firma oraz klient osoba. Co zrobić, aby dostać listę wszystkich klientów, bez względu na ich rodzaj? Gdybyśmy nie mieli takiego specjalnego zarządzania ekstensjami, musielibyśmy stosować jakieś „sklejanie" dwóch (lub więcej) kolekcji. A tak mamy to zapewniane przez nasz system.

Właśnie, ale jak to zaimplementujemy? Oczywiste wydaje się zmodyfikowanie konstruktora dodającego obiekty do ekstensji konkretnej klasy. W tym celu musimy:

```
public ObjectPlus() {
        super();

        Class superClass = this.getClass();
        while((superClass=superClass.getSuperclass()) != ObjectPlus.class)
{
                Vector ekstensja = null;

                if(ekstensje.containsKey(superClass)) {
                        // Ekstensja tej klasy istnieje w kolekcji ekst
                        ekstensja = (Vector) ekstensje.get(superClass);
                }
                else {
                        // Ekstensji tej klasy jeszcze nie ma -> dodaj ja
                        ekstensja = new Vector();
                        ekstensje.put(superClass, ekstensja);
                }

                ekstensja.add(this);
        }
}
```

3-54 Kod obsługujący zawieranie się ekstensji

- uzyskać listę wszystkich nadklas, które nas interesują. Mówimy tu o klasach biznesowych, czyli nie uwzględniamy samego `ObjectPlus` i ewentualnych kolejnych nadklas,

- dla każdej klasy z listy dodajemy do jej ekstensji aktualną instancję.

Jak widać, da się to zrobić, co dowodzi kod z listingu 3-54. Jest on dokładną implementacją powyższej koncepcji, więc chyba nie wymaga szczegółowego komentarza.

Sposób użycia naszej nowej funkcjonalności jest dokładnie taki sam jak poprzednio. Różnica jest tylko taka, że w momencie pobrania kolekcji przechowującej ekstensję jakiejś klasy będzie ona zawierała również wszystkie instancje podklas.

3.3.9 Podsumowanie

W popularnych językach programowania występuje tylko najprostszy rodzaj dziedziczenia. Tak naprawdę, wszystkie inne rodzaje dziedziczenia można obejść za pomocą jednej lub kilku poniższych technik:

- Zastąpienie hierarchii za pomocą jednej klasy. Niewątpliwą zaletą tego rozwiązania jest duża łatwość implementacji. Czasami może być

ona pozorna, np. gdy trzeba zastąpić przesłanianie metod (i polimorficzne wołanie) za pomocą wielu instrukcji warunkowych (np. `case`-'ów lub `if`'ów). Inną wadą jest nieoptymalne wykorzystanie zasobów.

- Wykorzystanie agregacji/kompozycji. Do zalet zaliczymy na pewno optymalne wykorzystanie zasobów. W zależności od przyjętych szczegółów implementacji, szczególnie gdy nie korzystamy z gotowych bibliotek (np. `ObjectPlusPlus`) może nas czekać dość pracochłonna implementacja (m.in. metody „opakowujące"), chociaż zwykle nie jest ona zbyt trudna.

- Zastosowanie interfejsów. Wadą jest dość duża pracochłonność, nawet gdy zastosujemy któryś ze wzorców projektowych. Zaletą są za to szerokie możliwości oraz fakt, iż jest to technika bardzo często stosowana (a przez to dość znana większości programistów).

W przeciwieństwie do asocjacji, gdzie właściwie musimy tylko rozstrzygnąć, czy korzystamy z identyfikatorów czy z referencji, w przypadku dziedziczenia nie ma jednego idealnego rozwiązania. Każdy przypadek powinien być traktowany indywidualnie.

Pamiętajmy, że wszystkie omówione sposoby są obejściem dziedziczenia, a nie konstrukcjami równoważnymi. W związku z powyższym, tam gdzie się tylko da należy korzystać z dziedziczenia, a nie jego substytutów.

3.4 Ograniczenia i inne konstrukcje

Podrozdział 2.4.5 (strona 57) poświęciliśmy krótkiemu omówieniu najpopularniejszych rodzajów ograniczeń definiowanych przez UML. Teraz zajmiemy się sprawdzeniem, w jaki sposób możemy je przełożyć na język programowania.

Tak jak pewnie większość Czytelników przypuszcza, w popularnych językach programowania (Java, C#, C++) ograniczenia nie występują bezpośrednio. Jednym ze sposobów ich uwzględnienia jest ręczna implementacja w postaci metod. Stopień skomplikowania takich implementacji zależy od rodzaju ograniczeń. Istnieją co prawda częściowe implementacje języka OCL, ale zwykle współpracują one z bibliotekami dotyczącymi modeli (np. EMF) i dlatego nie będziemy się nimi zajmowali. W związku z tym, kolejne

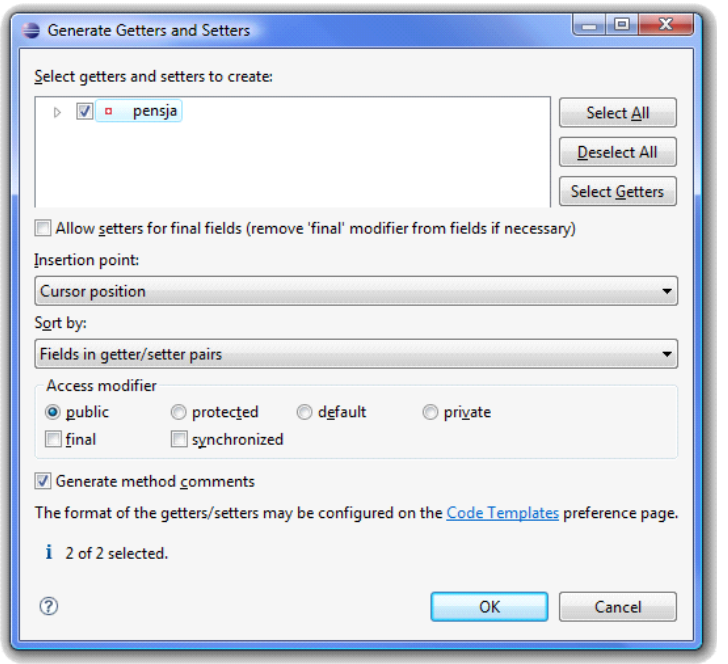
podrozdziały zawierają przykładowe rozwiązania dotyczące implementacji poszczególnych rodzajów ograniczeń.

3.4.1 Implementacja ograniczeń dotyczących atrybutów

Pisząc o ograniczeniach dotyczących atrybutów, mamy na myśli wszystkie ich rodzaje: zarówno dynamiczne, jak i statyczne. Po prostu każda akcja, która wykorzystuje atrybut. Abyśmy w ogóle byli w stanie zaimplementować jakiekolwiek ograniczenia dotyczące atrybutów, musimy poczynić pewne założenia:

- Atrybuty w klasie są ukryte (najlepiej jako `private`),
- Wszelka aktywność związana z atrybutami odbywa się przez dedykowane metody, np. tzw. gettery i settery.
- Powyższa reguła obowiązuje również wewnątrz klasy, w której są atrybuty. Czyli mając jakąś metodę w danej klasie, odwołującą się do atrybutu w tej samej klasie, powinniśmy to robić za pomocą odpowiedniego getttera lub settera, a nie bezpośrednio. Niestety, tego założenia nie da się kontrolować. Nawet jak atrybut jest prywatny, to i tak inwarianty w jego klasie mają do niego dostęp.

Jak widać, takie podejście wymaga zastosowania hermetyzacji ortodoksyjnej (pisaliśmy o tym w przypisie Błąd: Nie znaleziono źródła odwołania na stronie 108). Konsekwencją tej decyzji jest konieczność pisania odpowiednich metod dla każdego z atrybutów w klasie. Na szczęście współczesne środowiska programistyczne (IDE) wspierają proces tworzenia metod służących do operacji na „ukrytych" atrybutach. Na przykład w Eclipse jest do tego dedykowana opcja w menu kontekstowym: *Source – Generate Getters and Setters* (rysunek 3-90).



3-90 Generowanie metod w środowisku Eclipse

Spójrzmy na trzy przykłady ograniczeń dotyczących atrybutu pensja:

- Pensja nie może zmaleć (listing 3-55).

```
public class Pracownik {
        private float pensja;

        public void setPensja(float pensja) throws Exception {
                // Sprawdzenie ograniczenia
                if(pensja < this.pensja) {
                        throw new Exception("Pensja nie moze
                        zmalec!");
                }

                this.pensja = pensja;
        }
}
```

3-55 Implementacja ograniczenia dla atrybutu pensja – przykład nr 1

- Wzrost pensji nie może być większy niż 10% (listing 3-56).

```
public void setPensja(float pensja) throws Exception {
        // Sprawdzenie ograniczenia
        if(this.pensja * 1.1f < pensja) {
                throw new Exception("Wzrost pensji nie moze byc
                wiekszy niz 10%!");
        }

        this.pensja = pensja;
}
```

3-56 Implementacja ograniczenia dla atrybutu pensja – przykład nr 2

- Pensja > 2000 (listing 3-57).

```
public class Pracownik {
        private float pensja;

        public void setPensja(float pensja) throws Exception {
                // Sprawdzenie ograniczenia
                if(pensja < minimalnaPensja) {
                        throw new Exception("Pensja musi wynosic
                        co najmniej " +        minimalnaPensja);
                }

                this.pensja = pensja;
        }

        final static float minimalnaPensja = 2000;
        final static float maksymalnaPensja = 5000;
}
```

3-57 Implementacja ograniczenia dla atrybutu pensja – przykład nr 3

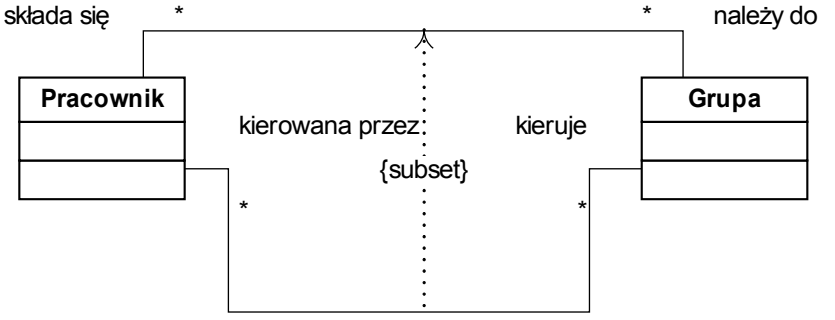
Myślę, że powyższe kody źródłowe są dość jasne, a przez to zrozumiałe. Jedyna dyskusyjna sprawa to odpowiedź na pytanie: co zrobić, gdy ograniczenie zostanie złamane? Ja zdecydowałem się zastosować wyjątki, ale oczywiście nie zawsze będzie to najlepsze rozwiązanie. Czasami lepiej jest wykorzystać mechanizm asercji (dostępny w niektórych językach programowania) czy też wyświetlać odpowiedni komunikat.

3.4.2 Implementacja ograniczenia {subset}

Implementację tego ograniczenia najłatwiej będzie objaśnić na przykładzie. Spójrzmy na rysunek 3-91.

Powiązania w ramach asocjacji opisanej przez role „składa się”, „należy do” dodajemy „normalnie”. Natomiast przed dodaniem powiązania w ramach asocjacji opisanej przez role „kierowana przez”, „kieruje”, sprawdzamy, czy istnieje powiązanie do dodawanego obiektu w ramach „nadrzędnej”

asocjacji. Aby cały proces zautomatyzować, dodamy nową funkcjonalność do klasy `ObjectPlusPlus`. Stworzymy metodę, która sprawdzi, czy istnieje powiązanie do podanego obiektu w ramach danej roli. Całą funkcjonalność związaną z ograniczeniami umieścimy w nowej klasie o nazwie `ObjectPlus4`. Aby nie tracić naszych dotychczasowych „udogodnień", będzie ona dziedziczyła z `ObjectPlusPlus`.



3-91 Ograniczenie typu {subset}

Listing 3-58 przedstawia kod metody w klasie `ObjectPlusPlus`, która sprawdza, czy istnieje powiązanie do podanego obiektu w ramach danej roli. Ważniejsze miejsca:

- (1). Sprawdzamy, czy w ogóle mamy informacje o podanej (w parametrze metody) nazwie roli.
- (2). Jeżeli nie, to rzucamy wyjątek.
- (3). W tym miejscu programu mamy pewność, że istnieje zapis dotyczący powiązań dla podanej nazwy roli. Dzięki temu odzyskujemy odpowiednią kolekcję.
- (4). Badamy, czy kontener mapujący przechowujący informacje o powiązaniach zawiera wartość pokazująca na sprawdzany obiekt. Rezultat zwracamy.

```
public boolean czyIstniejePowiazanie(String nazwaRoli,
   ObjectPlusPlus obiektDocelowy) throws Exception {
        HashMap<Object, ObjectPlusPlus> powiazaniaObiektu;
```

```
(1)          if(!powiazania.containsKey(nazwaRoli)) {
                    // Brak powiazan dla tej roli
(2)                 throw new Exception("Brak powiazan dla roli: " +
                       nazwaRoli);
             }
(3)          powiazaniaObiektu = powiazania.get(nazwaRoli);

(4)          if(powiazaniaObiektu.containsValue(obiektDocelowy)) {
                    // Obiekt docelowy istnieje
                    return true;
             }
             // brak obiektu
             return false;
     }
```

3-58 Implementacja ograniczenia {subset} - krok 1

Kolejnym elementem, który musimy zaimplementować, jest metoda dodająca powiązanie razem ze sprawdzaniem warunku {subset}. Jedno z możliwych rozwiązań pokazane jest na listingu. Fragmenty wymagające komentarza:

- (1). Konstruktor, który w swoim ciele odwołuje się do konstruktora nadklasy (przypominamy, że jest to niezbędny warunek poprawnego działania obiektów należących do klas dziedziczących z `ObjectPlus`).

- (2). Wywołujemy metodę sprawdzającą istnienie powiązania dla podanej roli (omawianą wcześniej).

- (3). Dodajemy powiązanie, korzystając z już istniejących mechanizmów klasy `ObjectPlus`.

- (4). Jeżeli nie istnieje powiązanie w ramach asocjacji nadrzędnej (złamanie ograniczenia), to rzucamy wyjątek.

```
     public class ObjectPlus4 extends ObjectPlusPlus {
(1)          public ObjectPlus4() {
                    super();
             }
             public void dodajPowiazanie_subset(String nazwaRoli,
             String
                odwrotnaNazwaRoli,  String nazwaRolinadrzednej,
(2)             ObjectPlusPlus obiektDocelowy) throws Exception {
                    if(czyIstniejePowiazanie(nazwaRoliNadrzednej,
                    obiektDocelowy)) {
                           // Istnieje powiazanie do dodawanego
```

```
(3)                         obiektu w ramach roli nadrzednej
                            // Mozemy utworzyc nowe powiazanie
                            dodajPowiazanie(nazwaRoli,
                            odwrotnaNazwaRoli, obiektDocelowy);
                    }
                    else {
(4)                         // Brak powiazania do dodawanego obiektu
                            w ramach roli nadrzednej ==> wyjatek
                            throw new Exception("Brak powiazania do
                            dodawanego obiektu '"
                                    + obiektDocelowy + "'w ramach roli
                                    nadrzednej '" + nazwaRoliNadrzednej
                                    + "'!");
                    }
            }
    }
```

3-59 Implementacja ograniczenia {subset} - krok 2

A jak możemy z tego korzystać? Na dwa sposoby:

- Automatycznie – listing 3-60. Tworzymy pracownika (1) oraz grupę (2). Następnie dodajemy powiązanie nadrzędne (3) oraz podrzędne, korzystając ze specjalnej metody (4). I w celu sprawdzenia poprawności wyświetlamy informacje (5).

```
(1)   Pracownik p = new Pracownik("Jan", "Kowalski");
(2)   Grupa g = new Grupa("Grupa nr 1");
      try {
              // Dodaj "zwykle" powiazanie
(3)           p.dodajPowiazanie(Pracownik.rolaNalezyDo,
              Grupa.rolaSkladaSie, g);

(4)           p.dodajPowiazanie_subset(Pracownik.rolaKieruje,
              Grupa.rolaKierowanaPrzez, Pracownik.rolaNalezyDo, g);

              // Wyswietl informacje o wszystkich powiazaniach tego
(5)           obiektu
              p.wyswietlPowiazania(System.out);
      } catch (Exception e) {
              e.printStackTrace();
      }
```

3-60 Wykorzystanie ograniczenia {ordered} – sposób automatyczny

- Ręcznie – samodzielnie sprawdzając spełnienie warunku (listing 3-61). Takie podejście oznacza nieco więcej pracy, ale zapewnia większą elastyczność.

```
Pracownik p = new Pracownik("Jan", "Kowalski");
Grupa g = new Grupa("Grupa nr 1");

try {
        // Dodaj "zwykle" powiazanie
        p.dodajPowiazanie(Pracownik.rolaNalezyDo,
        Grupa.rolaSkladaSie, g);

        // Sprawdz czy istnieje "nadrzedne" powiazanie
        if(p.czyIstniejePowiazanie(Pracownik.rolaNalezyDo, g)) {
                // Powiazanie nadrzedne istnieje => mozna dodac
                nowe powiazanie
                p.dodajPowiazanie(Pracownik.rolaKieruje,
                Grupa.rolaKierowanaPrzez, g);
        }
        else {
                // Brak powiazania nadrzednego
                System.out.println("Brak powiazania nadrzednego
                w roli: " +    Pracownik.rolaNalezyDo);
        }

        // Wyswietl informacje o wszystkich powiazaniach tego
        obiektu
        p.wyswietlPowiazania(System.out);
} catch (Exception e) {
        // Blad
        e.printStackTrace();
}
```

3-61 Wykorzystanie ograniczenia {ordered} – sposób ręczny

3.4.3 Implementacja ograniczenia {ordered}

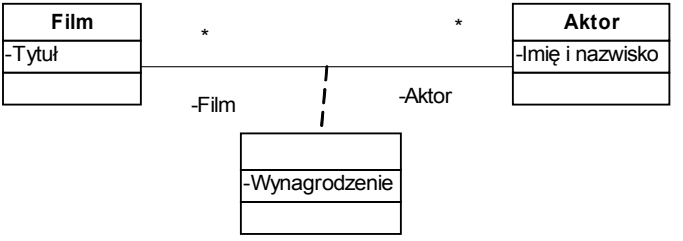
Implementacja ograniczenia {ordered} jest stosunkowo prosta. Jedyne co musimy zrobić to zapewnić, aby kolejność przechowywanych elementów nie ulegała zmianie. W tym celu trzeba użyć określonego rodzaju kontenera. Jeżeli chcemy, aby elementy były przechowywane w kolejności dodawania, to wykorzystujemy np. `List`. Natomiast, gdy potrzebujemy specyficznych kryteriów sortowania, to warto zastosować pojemnik umożliwiający zdefiniowanie komparatora (np. `TreeSet`). Komparator jest to specjalny obiekt, którego zadaniem jest określenie kolejności porównywanych elementów (implementuje on interfejs `Comparator`). Dzięki temu metoda sortująca wie, w jakiej kolejności umieszczać obiekty w określonym pojemniku. Więcej na ten temat można znaleźć w [Ecke06].

Klasa `ObjectPlusPlus` domyślnie używa klasy `Vector` dla ekstensji (kolejność gwarantowana) oraz `HashMap` dla powiązań.

Reasumując – implementacja ograniczenia {ordered} jest dość prosta i w praktyce sprowadza się do zastosowania określonego typu pojemnika (gwarantującego kolejność elementów).

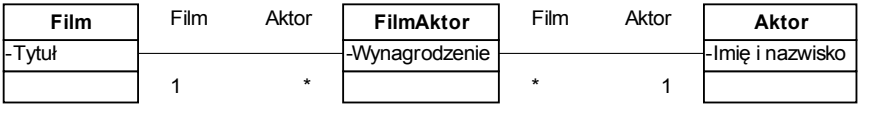
3.4.4 Implementacja ograniczenia {bag} oraz {history}

Klasyczna asocjacja, beż żadnych dodatkowych oznaczeń, nie pozwala na przechowywanie więcej niż jednego powiązania pomiędzy tymi samymi obiektami. Jeżeli z jakichś powodów jest nam to potrzebne, musimy zastosować ograniczenie {bag}. Zwykle taka sytuacja zachodzi w przypadku asocjacji z atrybutem, tak jak pokazano na rysunku 3-92.



3-92 Asocjacja z atrybutem

Ale jak pamiętamy z podrozdziału 3.2.3.3 (strona 147), implementacja takiej asocjacji polega na jej przekształceniu na dwie asocjacje binarne oraz klasę pośredniczącą – tak jak na rysunku 3-93.



3-93 Efekt przekształcenia asocjacji z atrybutem pokazanej na rysunku 3-92

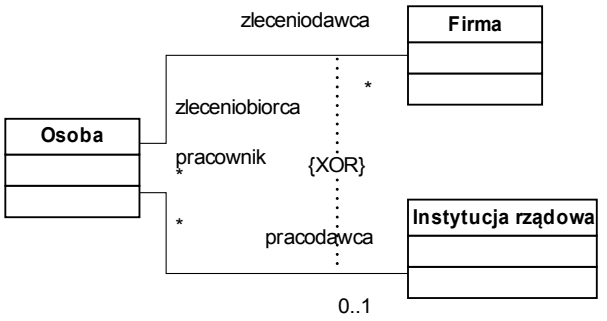
Spójrzmy, co się stało – problem powiązań pomiędzy tymi samymi obiektami po prostu zniknął. Zamiast powtarzającego się połączenia pomiędzy aktorem a filmem, mamy powiązania pomiędzy aktorem a klasą `FilmAktor` oraz `FilmAktor` a aktorem. Czyli nic specjalnego nie musimy robić. W takiej sytuacji ograniczenie {bag} po prostu ignorujemy i normalnie implementujemy asocjację.

Może się jednak tak zdarzyć, że ktoś chciałby przechowywać duplikaty, np. aby zliczać, ile razy i jaki napój ktoś wypił (dla klas `Osoba` oraz `Napój`).

w takiej sytuacji trzeba wybrać kontener, który pozwala zapamiętać wiele razy tę samą referencję, np. `Vector`. Klasa `ObjectPlusPlus` domyślnie używa klasy `HashMap` dla powiązań, która pilnuje unikalności kluczy. Można dodawać duplikowane wartości, ale z unikalnymi kluczami. Lub trzeba zmodyfikować przyjęte założenia projektowe dla klasy `ObjectPlusPlus`.

W przypadku ograniczenia {history} sytuacja jest identyczna. Powiedzieliśmy wcześniej, że oznacza ono mniej więcej to samo co {bag}, ale podkreśla aspekt czasowy informacji. W związku z tym, z punktu widzenia implementacji postępujemy dokładnie tak samo jak dla ograniczenia {bag}.

3.4.5 Implementacja ograniczenia {XOR}



3-94 Przykład ograniczenia typu {xor}

Ograniczenie {xor} dba o to, aby w ramach grupy asocjacji występowało tylko jedno powiązanie naraz (przykład na rysunku 3-94). Innymi słowy, nawiązując do przykładu, konkretna osoba może być połączona albo z firmą, albo z instytucją rządową. Oczywiście, ograniczenie może dotyczyć więcej niż dwóch asocjacji.

Co musimy zrobić, aby zaimplementować taki rodzaj ograniczenia? Sprawa jest dość oczywista – należy zadbać, aby w zdefiniowanej grupie asocjacji (ról) występowało tylko jedno powiązanie. Pokażemy przykładowe rozwiązanie dla klasy `ObjectPlus4` – listing 3-62.

```
     public class ObjectPlus4 extends ObjectPlusPlus {
(1)          private List<String> roleXOR = new LinkedList<String>();

             // [...]

(2)          public void dodajRoleXOR(String nazwaRoliXor) {
                     roleXOR.add(nazwaRoliXor);
```

```
        }

(3)     public void dodajPowiazanie_xor(String nazwaRoli, String
           odwrotnaNazwaRoli, ObjectPlusPlus obiektDocelowy)
        throws
(4)        Exception {
             if(roleXOR.contains(nazwaRoli)) {
                    // Aktualnie dodawana rola jest objeta
                    ograniczeniem XOR

(5)                 // Sprawdz czy jest juz jakies powiazanie
                    w ramach rol objetych ograniczeniem
                    if(czyIstniejePowiazanie()) {
                           throw new Exception("Istnieja juz
                              powiazanie w ramach rol objetych
                              ograniczeniem {XOR}!");
                    }

             // Nie ma powiazan, wiec ponizej dodajemy
             powiazanie korzystajac z "normalnej" metody
             z nadklasy
             }

             // Dodaj powiazanie
(6)          super.dodajPowiazanie(nazwaRoli,
             odwrotnaNazwaRoli, obiektDocelowy);
        }

        private boolean czyIstniejePowiazanie() {
             for(String rola : roleXOR) {
                    if(this.czySaPowiazania(rola)) {
                           return true;
                    }
             }

             return false;
        }
        // [...]
```

3-62 Implementacja ograniczenia {xor} dla klasy ObjectPlus4

Modyfikację zaczynamy od dodania pojemnika (1) przechowującego nazwy ról, których dotyczy ograniczenie. Następnie tworzymy metodę, która do ww. pojemnika doda podaną nazwę roli (2). Takie dodanie oznacza, iż asocjacja w ramach tej roli jest objęta ograniczeniem. Aby nie mnożyć metod o różnych nazwach, przesłaniamy już istniejąca metodę, tak aby uwzględniała implementowany xor (3). W jej ciele wykonujemy następujące czynności:

- (4). Sprawdzamy, czy dodawana rola jest objęta ograniczeniem {xor}.

- (5). Dla ról podlegających ograniczeniu sprawdzamy, czy istnieje już jakieś powiązanie. Jeżeli tak, to rzucamy wyjątek. Jeżeli powiązań nie ma, to sterowanie przechodzi do standardowej metody tworzącej połączenie pomiędzy obiektami.

- (6). Wywołanie „zwykłej" metody (z nadklasy) tworzącej powiązanie pomiędzy obiektami.

Tę przykładową implementację można uzupełnić o obsługę ograniczenia w przypadku dodawania powiązania „z drugiej strony".

3.4.6 Implementacja innych ograniczeń

W poprzednich podrozdziałach przedstawiliśmy najpopularniejsze „standardowe" ograniczenia sformułowane w ramach notacji UML. Należy jednak pamiętać, że analityk ma całkowitą swobodę w formułowaniu ograniczeń i nie musi używać jedynie tych, które dostarczyli twórcy UML. Co zrobić w takiej sytuacji?

W przypadku dowolnych, innych ograniczeń, np. pisanych „zwykłym tekstem" trzeba:

- Zrozumieć „co autor miał na myśli". Ta część będzie prawdopodobnie najtrudniejsza, szczególnie gdy ograniczenie jest napisane bez wykorzystywania formalizmów, czyli zwykłym tekstem. W razie wątpliwości należy zwrócić się do osoby, która je sformułowała. Na pewno jest to bezpieczniejsze podejście niż zdawanie się na własne wyczucie. Może się bowiem okazać (np. na etapie testów), że z powodu niezrozumienia wymagań na system zachodzi konieczność dokonywania dość znaczących zmian w kodzie programu.

- Zaimplementować je, wykorzystując odpowiednie metody.

Czasami z pozoru drobne ograniczenie może wymagać nawet zmiany podejścia do tworzenia programu, np. zastosowanie ortodoksyjnej hermetyzacji. I trzeba się nastawić, że rzadko będziemy mogli skorzystać z gotowych rozwiązań.

3.5 Model relacyjny

Na początku tego rozdziału warto sobie zadać pytanie: po co w książce dotyczącej podejścia obiektowego zajmować się modelem relacyjnym?

Krótka odpowiedź brzmi: bo jest popularny. Praktycznie każdy projektant czy programista tworzący trochę większy system korzysta z bazy danych. A w dzisiejszych czasach termin „baza danych" oznacza zwykle „relacyjna baza danych". Z punktu widzenia projektanta czy programisty, przyczyny takiego stanu rzeczy nie są bardzo istotne[21]. Najważniejsze jest, aby sobie umieć radzić w takiej sytuacji. Na czym polega problem z wykorzystywaniem modelu relacyjnego? Otóż musimy połączyć dwa, a właściwie trzy światy: obiektowy stosowany w analizie, obiektowy wykorzystywany w językach programowania (uboższy) ze światem relacyjnym stosowanym w składzie danych. Innymi słowy: analogicznie do metod przejścia z bogatszego modelu obiektowego (analiza) do uboższego modelu obiektowego (Java, C#, C++), musimy zapewnić mapowanie (tłumaczenie) konstrukcji obiektowych na relacyjne i odwrotnie. Zjawisko to nosi miano niezgodności impedancji (termin zaczerpnięty z elektroniki).

Ponieważ nie jest to książka poświęcona modelowi relacyjnemu jako takiemu, to nie będziemy go tutaj dokładnie opisywali. Osoby, które nie miały z tym zagadnieniem zbyt dużo do czynienia, mogą poszerzyć swoją wiedzę, czytając, np. [Bana04].

Jednakże w celu przypomnienia krótko opiszemy zasady i pojęcia modelu relacyjnego.

- Odpowiednikiem klas są tabele.

- Każda tabela może mieć dowolną liczbą kolumn przechowujących wartości atrybutów. Każda taka kolumna jest określonego typu, np. liczba, tekst, wartość binarna. Tak naprawdę typów jest więcej, ale nie będziemy się tym dokładnie zajmowali.

- Wiersze tabel (zwane krotkami lub rekordami) opisują poszczególne wystąpienia – są odpowiednikiem obiektów.

[21] Tak dla porządku możemy wymienić kilka z nich. Najważniejsza wydaje się być wygoda płynąca ze stosowania języka zapytań (SQL). Nie bez znaczenia są też istniejące systemy spadkowe, z którymi musimy współpracować. Na rynku istnieje też duży wybór systemów zarządzania bazami danych, poczynając od darmowych (np. mySQL), przez produkty Microsoftu czy Oracle, a kończąc na dużych systemach IBM. Na pewno nie pomaga też brak dobrego, i co ważniejsze zaimplementowanego, standardu dotyczącego obiektowych baz danych (tym języka zapytań).

- Kolejność występowania rekordów w tabeli nie ma znaczenia biznesowego.

- Każda tabela może mieć klucz główny oraz klucze obce.

- Klucz główny służy do jednoznacznej identyfikacji poszczególnych rekordów i ma zasadnicze znaczenie przy tworzeniu relacji pomiędzy tabelami (odpowiednik asocjacji).

- Wartość klucza obcego identyfikuje powiązaną krotkę, przeważnie z innej tabeli, np. mamy tabele `Osoba` oraz `Firma`. Któraś z osób ma wartość klucza głównego, np. 12345. W tabeli `Firma` występuje kolumna zawierająca klucz obcy odnoszący się do osoby i informujący nas, że osoba o określonej wartości swojego klucza głównego (12345) jest właścicielem firmy.

- Niektóre kolumny w tabelach mogą przechowywać specjalną wartość NULL. Oznacza to brak wartości, np. brak nazwiska panieńskiego.

- Relacje o licznościach wiele-do-wielu muszą być modelowane za pomocą tabel pośredniczących.

Jak widać, powyższy model jest bardzo prosty i pozbawiony większości konstrukcji znanych z obiektowości. Myślę, że się zgodzimy, iż jego stosowanie oznacza pracę na niższym poziomie abstrakcji, bardziej techniczną w porównaniu z modelem obiektowym.

Jak wspomnieliśmy wcześniej, przykrą konsekwencją wykorzystywania relacyjnych baz danych z poziomu obiektowych języków programowania jest niezgodność impedancji. W efekcie musimy dopasowywać do siebie „dwa różne światy": relacyjny i obiektowy. Następne rozdziały będą właśnie poświęcone omówieniu tego zagadnienia.

3.5.1 Mapowanie klas

Mapowanie klas na tabele jest stosunkowo proste, ale trzeba liczyć się z pewnymi ograniczeniami. Ogólnie możemy stwierdzić, że:

- Klasy po prostu zastępujemy tabelami.

- Każdy atrybut jest przechowywany w oddzielnej kolumnie. Jej typ jest zbliżony do typu atrybutu w modelu obiektowym. Zasada ta do-

tyczy tylko typów prostych. W innych przypadkach musimy zastosować dedykowane rozwiązania – patrz dalej.

- Dla każdej tabeli dodajemy specjalny atrybut – klucz jednoznacznie identyfikujący „obiekt”. W tym celu raczej nie wykorzystujemy atrybutów „biznesowych”. Ze względów wydajnościowych warto aby była to liczba.

Istnieje też grupa specjalnych atrybutów (nie prostych), którymi trzeba zająć się indywidualnie. Zaliczamy do nich atrybuty:

- Złożone. Tutaj można sobie wyobrazić trzy podejścia (omówimy je na przykładzie adresu):
    - Jako jeden „spłaszczony atrybut”, np. Adres: „ul. Marszałkowska 12, 03-333 Warszawa, Polska”
    - Jako jeden atrybut, ale z użyciem specjalnej notacji np. XML (listing 3-63). Dzięki takiej strukturalnej budowie będzie łatwiej go przetwarzać.

```
<Adres>
        <Ulica>Marszałkowska</Ulica>
        <NrDomu>12</NrDomu>
        <KodPocztowy>03-333</KodPocztowy>
        <Kraj>Polska</Kraj>
</Adres>
```

3-63 Przykład mapowania atrybutu złożonego na format XML

    - Stworzenie oddzielnej tabeli Adres i połączenie jej z „główną” za pomocą klucza – rysunek 3-95.

IdAdres	Ulica	NrDomu	KodPocztowy	Kraj
328	Marszałkowska	12	03-333	Polska

3-95 Przykład mapowania atrybutu złożonego za pomocą oddzielnej tabeli

- Opcjonalny. Tutaj sprawa jest prosta: kolumna musi dopuszczać przechowywanie wartości `null`.

- Powtarzalny. W tym przypadku również można zastosować dwa podejścia:

    - Jako jedna kolumna ze specjalną składnią. Poszczególne wartości oddzielone np. przecinkami lub wykorzystanie XML – podobnie jak przy atrybucie złożonym.

    - Stworzenie oddzielnej tabeli i połączenie jej z „główną".

- Klasowy. W przeciwieństwie do klas, w modelu relacyjnym nie przewidziano konieczności zapamiętywania jednej, tej samej wartości dla wszystkich krotek danej tabeli. W związku z tym musimy zastosować jakieś własne rozwiązanie, np.

    - Utworzenie specjalnej tabeli zawierającej wartości atrybutów klasowych z różnych klas.

    - Zapamiętanie go poza bazą danych, np. w kodzie programu lub pliku konfiguracyjnym.

- Wyliczalny. W językach programowania tego typu atrybuty obsługiwaliśmy za pomocą metod. W przypadku modelu relacyjnego, a właściwe relacyjnej bazy danych możemy utworzyć dedykowaną:

    - Perspektywę[22] - oczywiście jeżeli system zarządzania bazą danych na to pozwala (nie wszystkie to obsługują),

    - Metodę w języku programowania bazy danych, np. w SQL,

    - Metodę w języku programowania (poza bazą danych).

[22] w świecie baz danych istnieje kilka rodzajów perspektyw. Z naszego punktu widzenia najważniejsze jest to, iż umożliwiają one „przykrycie" rzeczywistego (fizycznego) modelu danych przez taki, który sobie sami zdefiniujemy. Może to dotyczyć ukrycia lub przedefiniowania kolumn, tabel czy dodania dowolnych elementów do modelu. System zarządzania bazą danych z dobrze zaimplementowanymi perspektywami daje programiście całkowitą „iluzję" pracy z prawdziwymi danymi.

Z powyższego wywodu wynika, że mapowanie klas nie jest specjalnie trudnym procesem, ale za to może być dość żmudnym. Na szczęście niektóre narzędzia CASE mogą nas w tym wspomóc. Niemniej nie możemy oczekiwać całkowitego zautomatyzowania, choćby ze względu na to, że niektóre problemy można rozwiązać na kilka sposobów. Mimo wszystko, wygenerowanie szkieletu kodu w języku SQL (do późniejszej modyfikacji) i tak jest lepsze niż pisanie go w całości od początku.

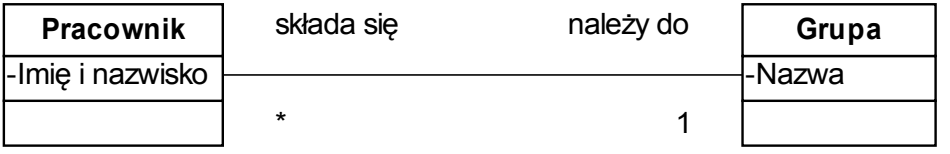
3.5.2 Mapowanie asocjacji

Jak wspomnieliśmy, w modelu relacyjnym związki pomiędzy tabelami są opisywane za pomocą klucza głównego oraz kluczy obcych. W porównaniu z modelem obiektowym jest to dość duża niewygoda, ponieważ zmusza nas do wprowadzania do modelu biznesowego elementów czysto technicznych. Jakby tego było mało, nie dysponujemy bardziej zaawansowanymi konstrukcjami będącymi odpowiednikami, np. asocjacji z atrybutem.

Przeanalizujmy teraz poszczególne sposoby radzenia sobie z różnymi typami asocjacji.

3.5.2.1 Asocjacje binarne

Podstawowy typ asocjacji zastępujemy za pomocą pojedynczej relacji. Jest tak dla liczności 1 – 1 oraz 1 - *. Spójrzmy na rysunek 3-96 przedstawiający zależność pomiędzy pracownikiem a grupą.



3-96 Asocjacja opisująca zależność pomiędzy pracownikiem a grupą

Powyższą sytuację biznesową możemy przedstawić w modelu relacyjnym tak jak pokazano to na rysunku 3-97. Jak widać utworzyliśmy dwie tabele. W tabeli pracownik znajduje się:

3-97 Przykładowy model relacyjny dla klas z rysunku 3-96

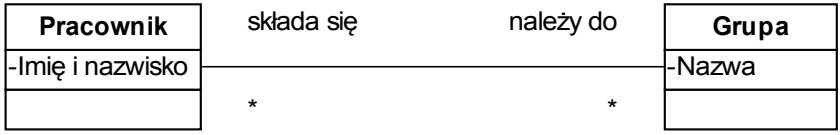
- Klucz główny (PK – *primary key*),
- Kolumna zapamiętująca imię i nazwisko,
- Klucz obcy (FK – *foreign key*) przechowujący identyfikator grupy, do której należy pracownik.

W tabeli `Grupa` znajdują się tylko dwie kolumny: klucz główny oraz nazwa.

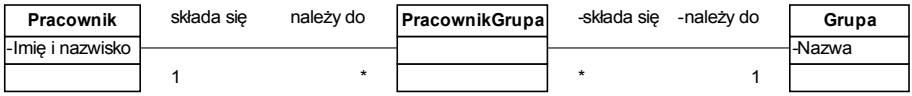
Zastanówmy się przez chwilę, w jaki sposób uzyskamy listę wszystkich pracowników należących do konkretnej grupy? w modelu obiektowym, mając „w ręku" instancję klasy `Grupa` (czyli referencję lub wskaźnik do obiektu klasy `Grupa`), od razu mamy dostęp do wszystkich powiązanych z nią obiektów. Niestety, model relacyjny ma to do siebie, że aby uzyskać analogiczne informacje, trzeba przejrzeć wszystkich pracowników, jacy tylko są w systemie i wybrać tych, którzy mają właściwy identyfikator grupy. Jak widać, jest to bardzo poważna wada. Na szczęście, dzięki umiejętnemu indeksowaniu[23] takie wyszukiwanie odbywa się dość szybko.

Powyższe rozwiązanie działa dla liczności „1 - *". A co z przypadkiem „* - *"? Spójrzmy na zmodyfikowany diagram klas z rysunku 3-98. W opisywanej tam sytuacji grupa może mieć wielu pracowników (tak jak poprzednio), ale pracownik może należeć do wielu grup. Aby tę sytuację przedstawić w modelu relacyjnym, musimy najpierw wprowadzić klasę pośredniczącą. Dzięki temu jedną asocjację „* - *" zastąpimy dwiema asocjacjami o licznościach „1 - *". Efekt takiego przekształcenia pokazano na rysunku 3-99.

[23] Ogólnie rzecz biorąc indeksowanie w bazie danych polega na utworzeniu (przez system) specjalnej struktury przechowującej powiązania do określonych krotek. Przypomina to trochę pojemnik mapujący (lub asocjację kwalifikowaną), gdzie kluczem jest indeksowana wartość, a wartością krotka zawierająca tę wartość.



3-98 Zmodyfikowana wersja diagramu z rysunku 3-96



3-99 Diagram przedstawiający efekt zastąpienia asocjacji „* - *”

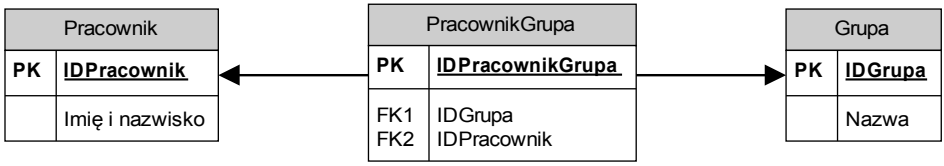
Ponieważ teraz mamy dwie asocjacje „1 - *”, możemy je bez problemu przedstawić w modelu relacyjnym:

- Trzy klasy mapujemy na trzy tabele.
- W „środkowej” tabeli umieszczamy klucze obce.

Przykładowy diagram relacyjny został przedstawiony na rysunku 3-100. W tabeli pośredniczącej umieszczamy tylko klucze obce wiążące pracowników z grupami oraz, ewentualnie, klucz główny dla takiej pary. Jak to bywa w modelu relacyjnym, aby znaleźć:

- pracowników należących do danej grupy,
- grupy, do których należy konkretny pracownik

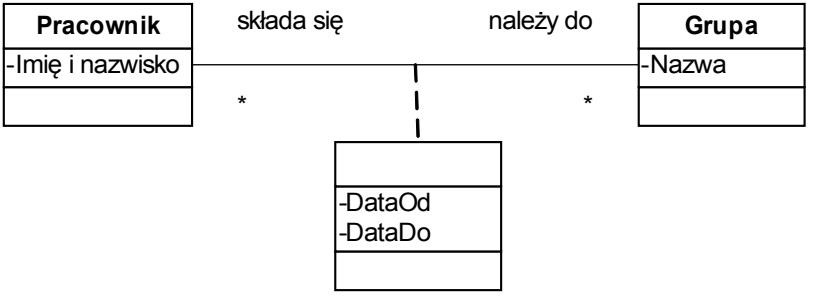
musimy przejrzeć wszystkie krotki z tabeli `PracownikGrupa`. Oczywiście wydajność takiego rozwiązania możemy znacząco poprawić, stosując odpowiednie indeksowanie (po stronie bazy danych).



3-100 Przykładowy model relacyjny dla klas z rysunku 3-99

3.5.2.2 Asocjacje z atrybutem

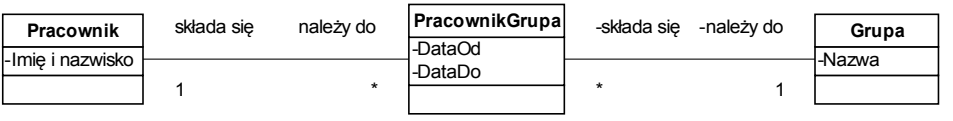
Bardzo podobnie jak w przypadku asocjacji binarnej „* - *”, postępujemy z asocjacją z atrybutem. Przykładowy diagram klas jest pokazany na rysunku 3-101. Rozszerza on poprzedni przykład o informację o czasie przynależności do konkretnej grupy.



3-101 Asocjacja z atrybutem

Co musimy zrobić, aby móc zapamiętać takie zależności w modelu relacyjnym? Chyba już wszyscy się domyślają:

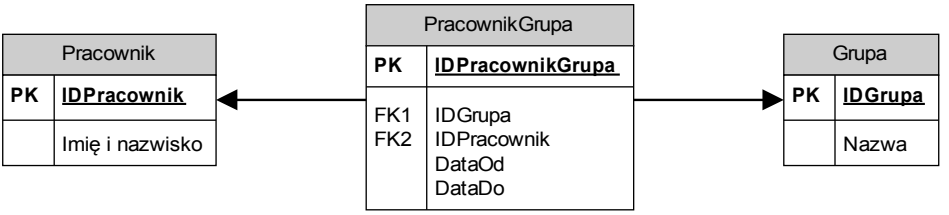
- Wprowadzamy klasę pośredniczącą z atrybutami pochodzącymi z klasy asocjacji (rysunek 3-102).



3-102 Wprowadzenie klasy pośredniczącej

- Trzy klasy mapujemy na trzy tabele.
- W „środkowej” tabeli umieszczamy klucze obce oraz atrybuty pochodzące z klasy pośredniczącej (klasy asocjacji).

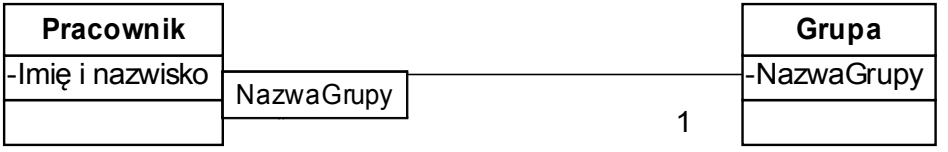
Relacyjny (a może również rewelacyjny) efekt naszych przekształceń został pokazany na rysunku 3-103. Widzimy, że oprócz kluczy obcych (tak jak w poprzednim przykładzie), dodaliśmy również kolumny opisujące daty rozpoczęcia oraz zakończenia.



3-103 Przykładowy model relacyjny dla diagramu klas z rysunku 3-102

3.5.2.3 Asocjacje kwalifikowane

Asocjacje kwalifikowane nie mają bezpośredniego odpowiednika w modelu relacyjnym. Możemy je częściowo „symulować", stosując jako klucz obcy kwalifikator. Spójrzmy na przykładowy diagram klas zawierający asocjację kwalifikowaną (rysunek 3-104).



3-104 Przykładowa asocjacja kwalifikowana

Analityk stworzył taką konstrukcję, chcąc szybko docierać do grupy konkretnego pracownika na podstawie jej nazwy. Możemy to przedstawić w modelu relacyjnym, stosując nazwę grupy jako klucz główny (w tabeli `Grupa`) oraz jako klucz obcy w tabeli `Pracownik`. Przykładowy diagram zawierający dwie tabele oraz jedną relację pokazano na rysunku 3-105. Widzimy, że tabela Pracownik, oprócz swojego klucza głównego (`IDPracownik`), zawiera kolumnę „`Imię i nazwisko`" oraz klucz obcy (`NazwaGrupy`) będący nazwą grupy. Dzięki temu jesteśmy w stanie nawigować od tabeli pracownik do tabeli grupa. Można dyskutować, czy tworzenie klucza głównego w postaci tekstowej jest właściwym rozwiązaniem[24]. Na pewno jest dopuszczalnym (przez system zarządzania bazą danych).

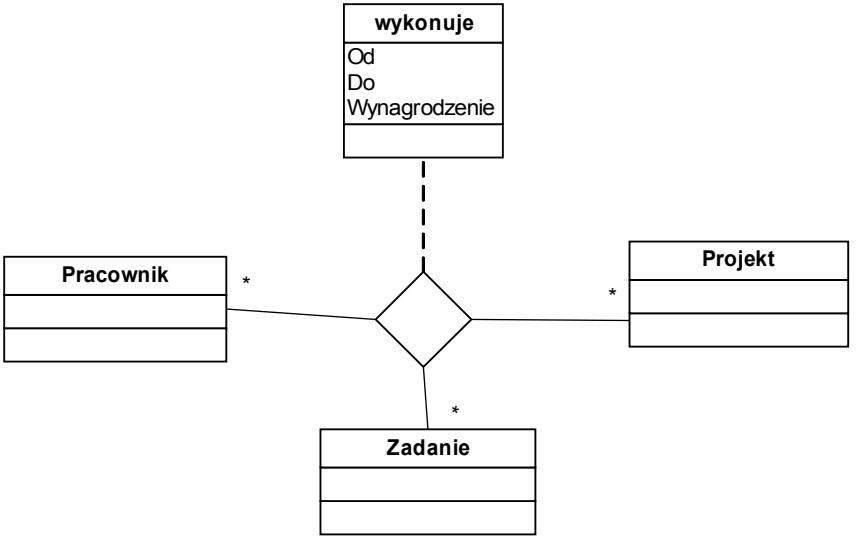
[24] Moje wątpliwości biorą się stad, że porównywanie liczb jest dużo szybsze niż porównywanie tekstów. A przecież klucze są nieustannie poddawane porównywaniom. Klucz liczbowy również zajmuje mniej miejsca: zwykle 4 bajty, a tekst co najmniej kilka razy więcej.

3-105 Przykładowe przedstawienie asocjacji kwalifikowanej w modelu relacyjnym

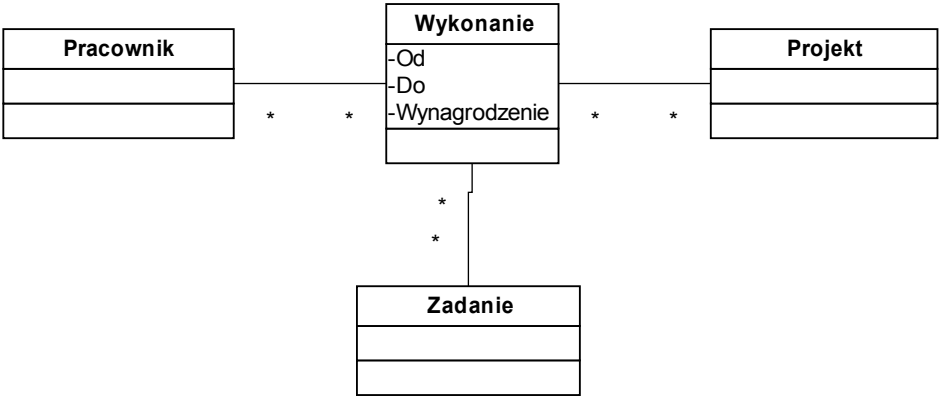
3.5.2.4 Asocjacje n-arne

Przekształcenie asocjacji n-arnej na model relacyjny jest dość proste (chociaż, jak pamiętamy, same asocjacje n-arne już takie nie są – str. 39 oraz 151). Podobnie jak w poprzednich przypadkach, wprowadzamy klasę pośredniczącą. Dzięki temu otrzymujemy n-asocjacji binarnych. A z nimi już wiemy, jak sobie poradzić.

Spójrzmy na rysunek 3-106 przedstawiający przykładową asocjację n-arną (dla n=3). Zgodnie z zasadami opisanym w podrozdziale 3.2.3.5 (strona 151) przekształcamy ją na trzy asocjacje binarne oraz klasę pośredniczącą (rysunek 3-107).



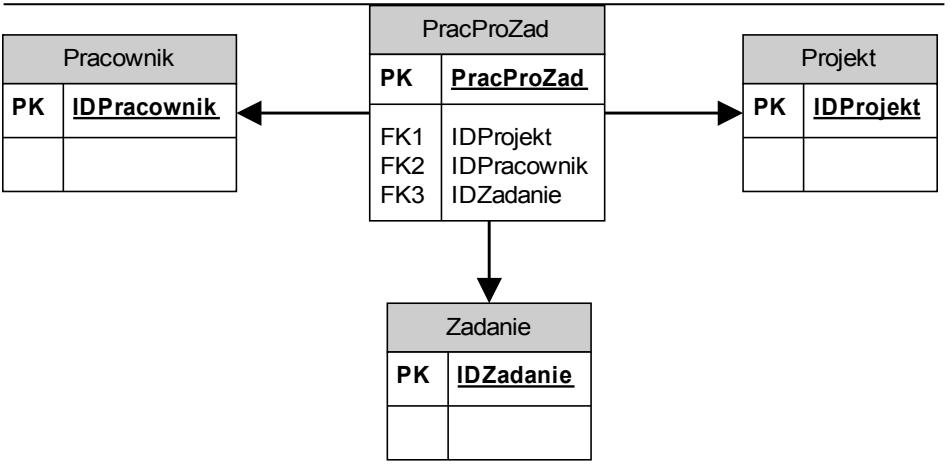
3-106 Przykładowa asocjacja n-arna



3-107 Asocjacja n-arna po przekształceniu

Następnie tak przekształconą asocjację modelujemy za pomocą tabel oraz relacji. Przykładowy efekt naszych działań jest pokazany na rysunku 3-108. Ze względu na czytelność rysunku nie uwzględniliśmy tabel pośredniczących (koniecznych dla liczności „* - *”). Podsumowując:

- Cztery klasy (n = 3) mapujemy na cztery tabele.
- W „środkowej” tabeli umieszczamy klucze obce oraz ewentualnie atrybuty pochodzące z klasy pośredniczącej (klasy asocjacji).



3-108 Przykładowy model relacyjny dla asocjacji n-arnej

3.5.2.5 Agregacja i kompozycja

Analogicznie jak przy przejściu z modelu analitycznego do implementacyjnego, agregacje realizujemy dokładnie tak samo jak asocjacje. Czyli, w zależności od konkretnej sytuacji, stosujemy relacje, ewentualnie z tabelami pośredniczącymi (patrz poprzednie podrozdziały).

W przypadku kompozycji możemy postarać się o bardziej wyrafinowane podejście. W zależności od możliwości systemu zarządzania bazą danych wykorzystujemy:

- Odpowiednią perspektywę, która dokładniej opisuje nasze potrzeby projektowe.

- Specyficzne więzy integralności oraz dedykowane wyzwalacze (*triggers*). Dzięki temu możemy dowolnie dokładnie przestrzegać zasad rządzących kompozycją. Tworząc odpowiednie wyrażenia SQL (a czasami nawet PL-SQL), jesteśmy w stanie wymusić np. kaskadowe usuwanie czy blokowanie współdzielenia części.

Innym rozwiązaniem, ale zdecydowanie gorszym jest implementacja cech kompozycji po stronie klasycznego języka programowania. Innymi słowy: baza danych traktuje połączenie jak zwykłe relacje, a o ich prawidłowe tworzenie oraz usuwanie dbamy na poziomie np. Javy czy C++.

3.5.3 Mapowanie dziedziczenia

W tradycyjnych relacyjnych bazach danych dziedziczenie jako takie nie występuje. W niektórych nowszych systemach dziedziczenie występuje, ale przeważnie w najprostszej postaci: rozłączne i pojedyncze. W przypadku jego braku można próbować różnych konstrukcji, które problem „obchodzą". Podkreślmy po raz kolejny, że wszędzie tam, gdzie dana konstrukcja występuje w formie natywnej, należy z niej korzystać. Jak sama nazwa wskazuje, obchodzenie problemu nie jest jego rozwiązaniem. Dlatego też, jeżeli system zarządzania bazą danych umożliwia dziedziczenie, należy to wykorzystać. W dalszej części tego rozdziału zajmiemy się sytuacją, gdy takiej możliwości nie mamy i musimy sami jakoś dziedziczenie zastąpić.

Można powiedzieć, że sposoby obejścia braku dziedziczenia w relacyjnych bazach danych są zbliżone do tych wykorzystywanych przy przejściu z modelu pojęciowego do implementacyjnego. Generalnie zaliczamy do nich dwa rozwiązania:

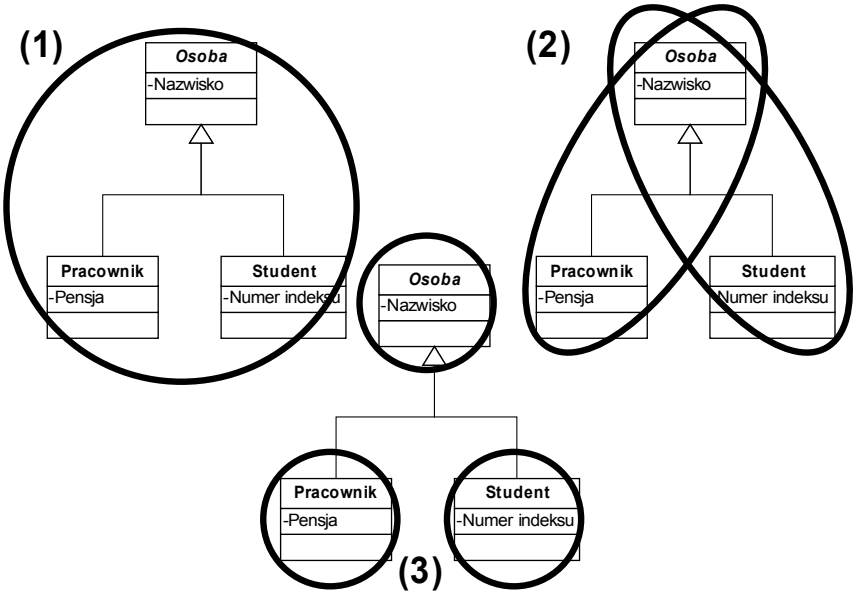
- Wykorzystanie relacji (asocjacji) do opisania zależności pomiędzy tabelami (klasami),
- Spłaszczenie hierarchii.

Aby ułatwić zapamiętanie tych rozwiązań, pokusiłem się o sporządzenie rysunku przedstawiającego obrazowo trzy różne podejścia (rysunek 3-109).

Możemy je podsumować jako:

- (1). Całkowite spłaszczenie hierarchii i umieszczenie wszystkich klas w jednej tabeli. Znajdą się w niej wszystkie atrybuty ze wszystkich klas występujących w drzewie dziedziczenia. Oprócz tego dodajemy kolumnę określającą typ krotki. Innymi słowy, taki dyskryminator informuje nas, „do jakiej klasy należy ten obiekt". Niewątpliwą zaletą tego podejścia jest ogromna prostota realizacji. Wadą, niestety, niewykorzystywanie części z atrybutów. Cechy te są bardzo podobne do tych występujących przy okazji rozwiązywania problemu dziedziczenia *overlapping* (patrz podrozdział 3.3.3 na stronie 181).
- (2). Częściowe spłaszczenie hierarchii i stworzenie dwóch tabel. Pomysł polega na utworzeniu jednej tabeli dla każdej z podklas. Umieszczamy w niej elementy z zastępowanej podklasy oraz ze

wszystkich jej nadklas. Zaletą jest względna prostota implementacji. Wadą jest to, iż dla każdej podklasy powtarzamy inwarianty ze wspólnych nadklas. W tym konkretnym przypadku w każdej z tabel wystąpi kolumna `Nazwisko`.



3-109 Trzy różne sposoby obejścia dziedziczenia

- (3). Stworzenie trzech tabel i połączenie ich relacjami. Każda klasa ma swoją własną tabelę. Dziedziczenie zastępujemy agregacjami, czyli w przypadku modelu relacyjnego, relacjami pomiędzy tabelami. Dodajemy odpowiednie klucze główne i obce. Zaletą tego rozwiązania jest pełne wykorzystanie wszystkich kolumn. Nic się nie marnuje, bo w tabeli są tylko elementy należące do konkretnej klasy. Zwykle nie ma rozwiązań bez wad, więc i tu jakieś znajdziemy. Szczególnie jedną: dość rozbudowana implementacja oraz skomplikowane zarządzanie taką konstrukcją, np. chcąc dowiedzieć się o nazwisko studenta, trzeba je odczytać z połączonej tabeli `Osoba`. Oczywiście w przypadku bardziej rozbudowanej hierarchii przechodzenie przez wiele tabel/relacji jeszcze bardziej utrudnia nam życie.

Z powyższych wywodów jasno wynika, że nie ma jednego, idealnego rozwiązania. Każde z nich ma swoje zalety, np. łatwość implementacji kosztem niepełnego wykorzystania kolumn (podejście nr 1), oraz wady, np. dość trudne używanie, za cenę optymalnego gospodarowania wykorzystaniem miejsca (rozwiązanie nr 3). Konkretne podejście trzeba dostosować do indywidualnej sytuacji.

3.5.4 Relacyjne bazy danych w obiektowych językach programowania

Każdy z popularnych języków programowania udostępnia biblioteki do obsługi relacyjnych baz danych. W przypadku Javy są to dwie generalne kategorie:

- JDBC (*Java Database Connectivity*). Jest to pewien standardowy interfejs obsługiwany przez większość baz danych dostępnych na rynku. Umożliwia wszelkiego rodzaju operacje na systemie bazy danych, włączając w to pobieranie danych oraz metadanych[25], uaktualnianie danych, tworzenie tabel, relacji itp. W zamyśle miały izolować funkcjonalność konkretnej bazy danych od kodu w języku Java. Dzięki temu nasz program powinien działać w ten sam sposób, bez względu na podłączoną bazę danych. Niestety, jak to w życiu bywa, rzeczywistość nie jest taka idealna i czasami po zmianie silnika BD trzeba dokonać pewnych zmian w kodzie programu. Ze względu na swoją uniwersalność nie zawsze są też najlepszym wyborem pod względem wydajności czy łatwości użycia.

- Natywne rozwiązania dla konkretnych baz dostarczane przez ich producentów. Dzięki temu, że konkretne API jest przeznaczone dla określonego systemu zarządzania bazą danych, zwykle jest lepiej zoptymalizowane niż JDBC. Dlatego, jeżeli nie zależy nam na przenaszalności, to warto sprawdzić, czy dla naszej bazy istnieją odpowiednie biblioteki programistyczne.

[25] Metadane, czyli dane o danych. W przypadku relacyjnej bazy danych, dane to rekordy zawierające np. nazwiska pracowników, nazwy firm, czy ilości sprzedanych produktów. Meta dane to informacje o budowie danych, czyli m.in. tabelach (nazwy, kolumny i ich typy) oraz relacjach pomiędzy nimi.

Na platformie Microsoft .NET, włączając w to język C#, dostęp do bazy danych jest realizowany za pomocą technologii ADO.NET. Jej podstawowa funkcjonalność oraz sposób pracy jest zbliżony do innych rozwiązań tego typu (patrz dalej). Wyjątkiem jest niedawno wprowadzona technologia LINQ (*Language Integrated Query*), która przenosi język zapytań (funkcjonalnie podobny do SQL) bezpośrednio do języka programowania (np. C#). Daje to ogromne możliwości i znacząco zmniejsza, lub wręcz likwiduje, niezgodność impedancji. Ale nie będziemy się tym teraz zajmowali, ponieważ, ze względu na swoją objętość, jest to temat na osobną książkę, np. [Fabr07].

Podobnie ma się sprawa z C++. Np. W przypadku dialektu rozwijanego przez Microsoft są to biblioteki ADO lub ODBC (*Open DataBase Connectivity*)[26].

Oprócz wyżej wymienionych rozwiązań istnieje wiele innych dostarczanych przez niezależnych programistów.

3.5.4.1 Wykorzystanie JDBC

Jak wspomnieliśmy, najpopularniejszym sposobem łączenia się z bazą danych w języku Java jest wykorzystanie funkcjonalności udostępnianej przez JDBC. Przykładowy program (listing 3-64) łączący się z bazą danych i wykonujący na niej różne operacje jest pokazany na listingu. Przeanalizujmy ważniejsze fragmenty:

- (1). Pozornie może się wydawać, że ten kod nic nie robi (bo nie korzystamy z utworzonego obiektu). Tak naprawdę jest po to, aby sprawdzić, czy odpowiedni sterownik JDBC jest w systemie. Jeżeli nie, to dostaniemy wyjątek.

- (2). Otwieramy połączenie do bazy danych.

- (3). Przygotowujemy instancję obiektu służącego do tworzenia poleceń dla BD.

- (4). Wysyłamy polecenie w języku SQL, które tworzy nową tabelę dla pracowników.

[26] Technologia ta jest odpowiednikiem JDBC, a właściwie to JDBC jest odpowiednikiem ODBC dla języka Java (bo historycznie najpierw było ODBC). Możliwe jest także wykorzystanie ODBC jako pomostu dla JDBC, gdy natywne sterowniki dla Javy nie istnieją.

- (5). Wysyłamy kolejne polecenie do bazy danych. Tym razem wstawiamy rekordy do tabeli – tworzymy informacje o pracowniku.

- (6). Wywołanie tej metody spowoduje wykonanie wcześniej przesłanych poleceń.

- (7). Wysyłamy zapytanie do bazy danych, zwracające nam wszystkich pracowników.

- (8). W efekcie otrzymaliśmy obiekt klasy `ResultSet`, będący swego rodzaju kolekcją. Udostępnia on m.in. metody umożliwiające poruszanie się po poszczególnych rekordach.

- (9). I tu dochodzimy do zasadniczej części pracy z danymi, jak również do konkretnego przykładu ilustrującego niezgodność impedancji. Zwróćmy uwagę, w jaki sposób pracujemy z otrzymanymi rekordami. Pobieramy wartości poszczególnych kolumn, korzystając z odpowiednich metod: różnych dla tekstu, liczb i innych rodzajów danych. W efekcie mamy różne wartości typu `string`, `int` itp., a nie konkretne obiekty.

```
   // Próba wczytania driver'a
   try
   {
(1)        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();
   }
   catch (ClassNotFoundException ex)
   {
           // Błąd
           System.err.println ("Wyjatek: " + ex);
           System.exit(0);
   }

(2) Connection db_conn = DriverManager.getConnection
   ("jdbc:odbc:dbname", "user", "pass");

(3) Statement db_statement = db_conn.createStatement();

(4) db_statement.executeUpdate("create table pracownik { int id,
   char(50) nazwisko};");
(5) db_statement.executeUpdate("insert into pracownik values (1, 'Jan
   Kowlski');");
(6) db_connection.commit();

   // [...]

   // Wykonaj zapytanie
```

```
(7) ResultSet result = db_statement.executeQuery("select * from
    pracownik");

    // Przetwarzanie wynikow
(8) while (result.next() )
    {
(9)         System.out.println ("ID : " + result.getInt("id"));
            System.out.println ("Nazwisko : " +
    result.getString("nazwisko"));
    }
```

3-64 Przykład wykorzystania JDBC

Jak widać, powyższy sposób pracy z danymi nie jest najwygodniejszy, szczególnie gdy porównamy go z modelem obiektowym, np. tym, który zaimplementowaliśmy w klasach `ObjectPlus`.

3.5.4.2 Wykorzystanie gotowej biblioteki do pracy z danymi (Hibernate)

Myślę, że zgodzimy się, iż bezpośrednia praca z relacyjną bazą danych nie jest zbyt wygodna. Podobnego zdania byli też twórcy szeregu dodatkowych bibliotek, których zadaniem jest ułatwienie dostępu do relacyjnej bazy danych, a właściwie łatwiejsze uzyskanie trwałości danych. Jedną z takich bibliotek jest Hibernate (http://www.hibernate.org/). Projekt ten jest rozwijany jako *open source* od roku 2001. W tej chwili jest już dojrzały i wykorzystywany przez setki tysięcy programistów na całym świecie. Jako ciekawostkę można dodać, że zasadnicza część kodu źródłowego liczy 76 tys. linii, a każdego dnia jest pobierany 3000 razy. Istnieje też jego wersja dla Microsoft .NET 1.1 oraz 2.0. W dalszej części rozdziału zajmiemy się krótkim omówieniem tej interesującej technologii. Więcej informacji na ten temat można znaleźć np. w [Mint06].

Biblioteka Hibernate nie likwiduje całkowicie problemu niezgodności impedancji, a tylko go zmniejsza. Programista cały czas jest świadomy istnienia relacyjnej bazy danych i częściowo przez nią ograniczany. Niemniej, jej wykorzystanie i tak znacząco ułatwia jego pracę.

W każdym dużym projekcie, a do takich Hibernate też jest wykorzystywany, kluczowe znaczenie ma wydajność zastosowanego rozwiązania. Jeżeli jest ona niewystarczającą, to nawet najbardziej wspaniała biblioteka pozostanie ciekawostką. Na szczęście ten problem nie występuje w przypadku omawianego komponentu. Twórcy twierdzą, że biblioteka jest wydajna (co potwierdza jej ogromna popularność) i wykorzystuje różne techniki optymalizacji:

- Cache'owanie obiektów. Polega na „sprytnym" dostępie do obiektów. Jeżeli od ostatniego użycia nie uległ on zmianie, to nie jest wydobywany z bazy danych (co musiałoby trochę potrwać), ale ze specjalnej struktury istniejącej w języku programowania (np. Javie czy C#).

- Cache'owanie wyników zapytań. Metoda podobna jak powyżej, ale dotycząca wyników zapytań. Jeżeli dane, których dotyczyło zapytanie, nie uległy zmianie, to jego rezultat nie jest pobierany z bazy danych, ale ze specjalnego „kontenera".

- Polecenia SQL wykonywane dopiero wtedy, gdy są naprawdę potrzebne, np. nie wtedy, kiedy zlecimy wykonanie zapytania, ale wtedy, gdy „skonsumujemy" jego wynik.

- Brak uaktualniania niezmodyfikowanych obiektów. W Hibernate jest specjalna metoda powodująca uaktualnienie obiektu i przesłanie go do bazy. Jeżeli zawartość obiektu nie została zmieniona, to nie ma potrzeby wysyłać go do bazy danych.

- Efektywne zarządzanie kolekcjami.

- Łączenie wielu zmian w jedno UPDATE. Dzięki temu można m.in. wykorzystać optymalizacje udostępnianie przez natywny język bazy danych (np. SQL).

- Leniwa inicjalizacja obiektów i kolekcji.

Zaletą powyższych optymalizacji jest to, że są przezroczyste dla programisty. Dzięki temu nie musi wykonywać żadnych specjalnych czynności, aby z nich korzystać. Wszystkim zajmuje się system.

Dalsza zawartość rozdziału bazuje na oficjalnym tutorialu dostępnym w [Hibe06]. Wykorzystamy bazę danych o nazwie HSQL (http://hsqldb.org/) napisaną w całości w języku Java. Dzięki temu unikniemy (czasami) skomplikowanej instalacji i konfiguracji. Musimy utworzyć katalog `lib`, do którego rozpakowujemy pobrane pliki jar. Następnie bazę uruchamiamy tak jak pokazano na listingu 3-65.

```
java -classpath ../lib/hsqldb.jar org.hsqldb.Server
```

3-65 Uruchomienie systemu bazy danych HSQL

Każda klasa, która chce wykorzystywać pełnię możliwości biblioteki, musi posiadać specyficzny atrybut służący do identyfikacji wystąpień. Zwykle jest on prywatny i typu liczbowego, np. `private long id`. Zmianą jego wartości zajmuje się Hibernate. Zalecane jest wykorzystywanie konwencji JavaBean, czyli tworzymy dedykowane metody set/get służace do zmiany oraz odczytu jego wartości.

W naszym przykładzie stworzymy prostą aplikację umożliwiającą planowanie zdarzeń oraz łączenie ich z uczestnikami. Przykładowy kod klasy `Event` (zdarzenie) pokazany jest na listingu 3-66. Jak widzimy, mamy dedykowany atrybut wymagany przez Hibernate, a służący do zapamiętywania identyfikatora obiektu (1). Oprócz tego mamy informację o nazwie zdarzenia (2) oraz jego dacie (3). Dla każdego z atrybutów utworzyliśmy odpowiednie metody służące do zmiany jego wartości oraz jej odczytu.

```
      public class Event {
(1)       private Long id;
(2)       private String title;
(3)       private Date date;
          public Event() {}

          public Long getId() {
              return id;
          }
          private void setId(Long id) {
              this.id = id;
          }
          public Date getDate() {
              return date;
          }
          public void setDate(Date date) {
              this.date = date;
          }
          public String getTitle() {
              return title;
          }
          public void setTitle(String title) {
              this.title = title;
          }
      }
```

3-66 Przykładowa klasa służąca do przechowywania informacji o zdarzeniach

Podstawą działania Hibernate jest XMLowy plik mapujący wykorzystywany do mapowania klas języka Java (model obiektowy) na elementy w bazie danych (model relacyjny). Każda klasa, którą ma obsługiwać Hibernate, musi być opisana za pomocą pliku o nazwie: `NazwaKlasy.hbm.xml`. Plik

ten musi znajdować się w tym samym katalogu co odpowiadający mu plik źródłowy `NazwaKlasy.java`. Listing 3-67 zawiera przykładową zawartość takiego pliku.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-
3.0.dtd">
<hibernate-mapping>
    <class name="events.Event" table="EVENTS">
        [...]
    </class>
</hibernate-mapping>
```

3-67 Przykładowa zawartość pliku mapującego Hibernate

Jak to z plikami XML bywa, zawiera on określone tagi. Większość z nich może posiadać dodatkowe atrybuty precyzujące znaczenie konkretnego tagu. Do najważniejszych z nich zaliczamy:

- Tag `class` może zawierać atrybuty (listing 3-67):
    - `name` oznacza nazwę klasy języka Java,
    - `table` informuje, w jakiej tabeli przechowywać wystąpienia tej klasy.
- Tag `id` służy do tworzenia identyfikatorów (kluczy) dla instancji konkretnej klasy (tabeli). Przykład pokazany jest na listingu 3-68. Atrybut:
    - `name` oznacza nazwę atrybutu klasy języka Java,
    - `column` informuje, w której kolumnie przechowywać jego wartości.
    - `generator` określa strategię zapewniania unikalności liczb.

```
<class name="events.Event" table="EVENTS">
        <id name="id" column="EVENT_ID">
            <generator class="native"/>
        </id>
        <property name="date" type="timestamp"
    column="EVENT_DATE"/>
```

```
        <property name="title"/>
    </class>
```

3-68 Przykład tagu id

- Tag `property` mapuje atrybuty z konkretnej klasy (listing 3-68).
    - `type` definiuje typ kolumny. Należy go użyć, gdy nie da się tego „odgadnąć" automatycznie, np. java.util.Date może być przedstawione jako `SQLdate`, `Timestamp` lub `time`.

Oprócz powyższych, specyficznych dla klasy plików konfiguracyjnych potrzebujemy poinformować bibliotekę o ogólnych ustawieniach. Przykładowa zawartość takiego pliku jest pokazana na listingu 3-69. Nie będziemy go szczegółowo omawiać. Podsumujemy tylko, że zawiera on informacje dotyczące połączenia z bazą danych, rodzaju języka SQL, sposobu zarządzania sesjami czy postępowania ze schematem bazy danych. Więcej informacji na ten temat można znaleźć w [Hibe06] czy [Mint06].

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-
3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!-- Database connection settings -->
        <property
name="connection.driver_class">org.hsqldb.jdbcDriver</property>
        <property
name="connection.url">jdbc:hsqldb:hsql://localhost</property>
        <property name="connection.username">sa</property>
        <property name="connection.password"></property>
        <!-- JDBC connection pool (use the built-in) -->
        <property name="connection.pool_size">1</property>
        <!-- SQL dialect -->
        <property
name="dialect">org.hibernate.dialect.HSQLDialect</property>
        <!-- Enable Hibernate's automatic session context management
-->
        <property
name="current_session_context_class">thread</property>
        <!-- Disable the second-level cache  -->
        <property
name="cache.provider_class">org.hibernate.cache.NoCacheProvider</proper
ty>
        <!-- Echo all executed SQL to stdout -->
        <property name="show_sql">true</property>
        <!-- Drop and re-create the database schema on startup -->
        <!-- <property name="hbm2ddl.auto">create</property> -->
        <mapping resource="events/Event.hbm.xml"/>
```

```
    </session-factory>
</hibernate-configuration>
```

3-69 Przykładowy plik konfiguracyjny dla Hibernate

Do rozpoczęcia pracy brakuje nam tylko jeszcze pewnej klasy pomocniczej. Nazwiemy ją `HibernateUtil`, a będzie odpowiedzialna za uruchomienie Hibernate oraz tworzenie sesji. Zawiera globalny obiekt `SessionFactory` i zapewnia bezpieczną obsługę wielu wątków. Jej kod przedstawiony jest na listingu 3-70.

```
public class HibernateUtil {

    private static final SessionFactory sessionFactory;

    static {
        try {
            // Create the SessionFactory from hibernate.cfg.xml
            sessionFactory = new
Configuration().configure().buildSessionFactory();
        } catch (Throwable ex) {
            // Make sure you log the exception, as it might be
swallowed
            System.err.println("Initial SessionFactory creation
failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}
```

3-70 Pomocnicza klasa dla Hibernate

Wypróbujmy wreszcie bibliotekę w działaniu i spójrzmy na kod z listingu 3-71. Pamiętajmy również, aby właściwie zlokalizować pliki konfiguracyjne xml. Ważniejsze miejsca:

- (1). Wywołanie głównej metody tworzącej zdarzenie. Przekazujemy jej dwa parametry: nazwę oraz datę zdarzenia.
- (2). Zamknięcie sesji Hibernate (cała aktywność odbywa się wewnątrz metody `createAndStoreEvent`).
- (3). Tworzymy nową sesję.

- (4). Rozpoczynamy transakcję. Jak przystało na poważny system pracy z bazą danych, Hibernate obsługuje transakcje. Jeżeli transakcja nie zostanie potwierdzona, to dane nie zostaną przesłane do bazy.

- (5). Tworzymy nowe zdarzenie i definiujemy jego parametry (6).

- (7). Zapisujemy zdarzenie w sesji i potwierdzamy transakcję (8).

```
public class EventManager {
    public static void main(String[] args) {
        EventManager mgr = new EventManager();

(1)     mgr.createAndStoreEvent(„Zdarzenie 01", new Date());

(2)     HibernateUtil.getSessionFactory().close();
    }

    private void createAndStoreEvent(String title, Date theDate)
{
(3)
        Session session = HibernateUtil.getSessionFactory()
            .getCurrentSession();
(4)
        session.beginTransaction();

(5)
        Event theEvent = new Event();
(6)     theEvent.setTitle(title);
        theEvent.setDate(theDate);
(7)
        session.save(theEvent);
(8)
        session.getTransaction().commit();
    }
}
```

3-71 Przykładowy kod ilustrujący wykorzystanie Hibernate

Efekt działania Hibernate możemy podejrzeć w pliku *log*. Dzięki temu możemy śledzić czynności wykonywane przez bibliotekę, co jest nieocenioną pomocą przy poprawianiu błędów (konsola 3-9).

```
20:24:36,897 DEBUG SchemaExport:303 - create table EVENTS
(EVENT_ID bigint generated by default as identity (start with 1),
EVENT_DATE timestamp, title varchar(255), primary key (EVENT_ID))
20:24:36,975 DEBUG ConnectionManager:421 - opening JDBC
connection
20:24:36,975 DEBUG AbstractSaveEventListener:153 - saving
[events.Event#<null>]
20:24:36,975 DEBUG AbstractSaveEventListener:244 - executing
insertions
20:24:36,991 DEBUG AbstractSaveEventListener:297 - executing
identity-insert immediately
20:24:36,991 DEBUG AbstractEntityPersister:2144 - Inserting
entity: events.Event (native id)
20:24:36,991 DEBUG AbstractBatcher:366 - about to open
PreparedStatement (open PreparedStatements: 0, globally: 0)
20:24:36,991 DEBUG SQL:401 - insert into EVENTS (EVENT_DATE,
title, EVENT_ID) values (?, ?, ?)
Hibernate: insert into EVENTS (EVENT_DATE, title, EVENT_ID)
values (?, ?, ?)
20:24:36,991 DEBUG AbstractBatcher:484 - preparing statement
20:24:36,991 DEBUG AbstractEntityPersister:1992 - Dehydrating
entity: [events.Event#<null>]
20:24:36,991 DEBUG AbstractBatcher:374 - about to close
PreparedStatement (open PreparedStatements: 1, globally: 1)
20:24:36,991 DEBUG AbstractBatcher:533 - closing statement
20:24:36,991 DEBUG AbstractBatcher:366 - about to open
PreparedStatement (open PreparedStatements: 0, globally: 0)
20:24:36,991 DEBUG SQL:401 - call identity()
Hibernate: call identity()
20:24:37,007 DEBUG ThreadLocalSessionContext:300 - allowing
proxied method [getTransaction] to proceed to real session
20:24:37,007 DEBUG JDBCTransaction:103 - commit
```

3-9 Przykładowy plik log dokumentujący działanie Hibernate

W tym momencie w naszej bazie danych mamy utworzone już informacje dotyczące co najmniej jednego zdarzenia. Aby nie były one usuwane przy każdym uruchomieniu programu, konieczne jest zakomentowanie odpowiedniej linii w pliku konfiguracyjnym Hibernate, tak jak pokazano to na listingu 3-69 (linia zaczynająca się od `<property name="hbm2ddl.auto">`).

Tak jak wspomnieliśmy, Hibernate obsługuje język zapytań. Najczęściej korzystamy z jego natywnego rozwiązania o nazwie (niezbyt zaskakującej): *Hibernate Query Language* (HQL). Funkcjonalnie (pod względem możliwości) jest zbliżony do SQL, ale trochę różni się składnią. Spójrzmy, jak może wyglądać metoda zwracająca całą ekstensję klasy `Event`. Korzysta ona właśnie z języka HQL – listing 3-72. Zwróćmy uwagę, że to, co otrzymujemy, jest zwykłą listą języka Java. Dzięki temu możemy ją wykorzystać

w dowolnym miejscu naszego programu, np. tak jak pokazano na listingu 3-73. I teraz uwaga: jak widać, pracujemy na kompletnych obiektach (klasy `Event`), a nie na (jak w przypadku „czystego JDBC) na oddzielnych wartościach (np. `String`) z bazy danych (przykład z listingu 3-64 na stronie 235).

```
private List listEvents() {
        Session session = HibernateUtil.getSessionFactory()
            .getCurrentSession();
        session.beginTransaction();
        List result = session.createQuery("from Event").list();
        session.getTransaction().commit();

        return result;
}
```

3-72 Metoda zwracająca ekstensję klasy (korzystająca z HQL w Hibernate)

```
EventManager mgr = new EventManager();

List events = mgr.listEvents();

for (int i = 0; i < events.size(); i++) {
        Event theEvent = (Event) events.get(i);

        System.out.println("Event: " + theEvent.getTitle() + "
        Time: "
                                + theEvent.getDate());
}

HibernateUtil.getSessionFactory().close();
```

3-73 Wykorzystanie metody zwracającej ekstensję klasy

Jak dotąd zajmowaliśmy się tylko bardzo prostą klasą, która nie była połączona z innymi klasami. Oczywiście w prawdziwych projektach tego typu sytuacja ma miejsce niezwykle rzadko. Na szczęście Hibernate umożliwia nam też dość komfortową pracę z asocjacjami, które są (prawie) automatycznie mapowane na relacje w bazie danych. Elementy asocjacji, które trzeba uwzględnić, to:

- kierunek,
- liczności,
- zachowanie się kolekcji służącej do implementacji (po stronie Javy).

```
public class Person {
    private Long id;
```

```
    private int age;
    private String firstname;
    private String lastname;

    public Person() {}
    // Metody get/set oraz prywatny set dla id.
}
```

3-74 Przykładowy kod dla klasy Person

Aby rozbudować nasz przykład, stworzymy klasę `Osoba` (`Person`) i powiążemy ją z wydarzeniami (`Events`). Jej przykładowy kod został przedstawiony na listingu 3-74. Jest on dość prosty i nie wymaga szczególnego komentarza. Zwróćmy tylko uwagę, że umieściliśmy prywatny identyfikator wymagany przez Hibernate. Ze względu na swoją prostotę metody set oraz get zostały pominięte na listingu.

Plik mapujący dla klasy `Person` jest pokazany na listingu 3-75. Widzimy tam definicję identyfikatora oraz trzy mapowania atrybutów. Każdy z nich należy do typu, który można jednoznacznie odwzorować. Dzięki temu, nie było potrzeby podawać dodatkowych parametrów mapowania. Musimy również wykonać nowy wpis do pliku konfiguracyjnego (odpowiednik zapisu `<mapping resource="events/Event.hbm.xml"/>` w listingu 3-69, strona 240).

```
<class name="events.Person" table="PERSON">
        <id name="id" column="PERSON_ID">
            <generator class="native"/>
        </id>
        <property name="age"/>
        <property name="firstname"/>
        <property name="lastname"/>
</class>
```

3-75 **Plik mapujący dla klasy Person**

Do klasy `Person` dodamy informację o jej wydarzeniach (`Events`) (listing 3-76) oraz uzupełnimy jej plik mapujący (listing 3-77). Wykorzystaliśmy pojemnik typu `Set` (obsługiwane są też inne rodzaje).

```
public class Person {
    // [...]
    private Set events = new HashSet();
    public Set getEvents() {
        return events;
    }
    public void setEvents(Set events) {
        this.events = events;
```

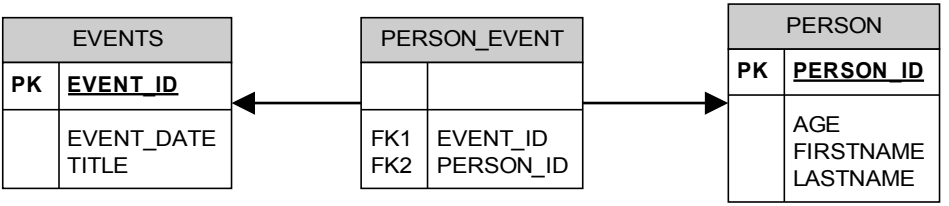
```
    }
```

3-76 Modyfikacja klasy Person umożliwiająca przechowywanie asocjacji

```
<class name="events.Person" table="PERSON">
     […]
       <set name="events" table="PERSON_EVENT">
              <key column="PERSON_ID"/>
              <many-to-many column="EVENT_ID"
              class="events.Event"/>
       </set>
</class>
```

3-77 Modyfikacja pliku mapującego klasy Person umożliwiająca przechowywanie asocjacji

W efekcie otrzymaliśmy schemat relacyjny pokazany na rysunku 3-110. Ale uwaga, coś tu jest dziwnego. Skąd się wzięła tabela pośrednicząca? Otóż Hibernate wygenerowało ją „automatycznie". Dzięki temu zaoszczędziliśmy trochę pracy.



3-110 Schemat relacyjny „wygenerowany" przez Hibernate

Na poziomie języka Java istnieje połączenie asocjacją (referencjami języka Java) od klasy `Person` do `Event` (ale uwaga: nie w drugą stronę).

Spójrzmy, jak wygląda metoda tworząca nowe powiązanie pomiędzy `Person` a `Event` – listing 3-78. Ważniejsze miejsca:

- (1). Tworzymy nową sesję.
- (2). Rozpoczynamy transakcję.
- (3). Bazując na podanym identyfikatorze, wczytujemy (pobieramy) odpowiadający mu obiekt klasy `Person` oraz `Event` (4).
- (5). Do obiektu klasy Person, a właściwie do jego kolekcji przechowującej powiązania, dodajemy informację o zdarzeniu. Zwróćmy

uwagę, że operacja odbywa się na zwykłych, natywnych dla Javy kontenerach. Hibernate ją automatycznie wykrywa i w tle uaktualnia bazę danych. Analogiczna „automatyka" istnieje dla atrybutów.

- (6). Potwierdzamy transakcję.

```
    private void addPersonToEvent(Long personId, Long eventId) {
(1)         Session session =
    HibernateUtil.getSessionFactory().getCurrentSession();
(2)         session.beginTransaction();

(3)         Person aPerson = (Person) session.load(Person.class,
    personId);
(4)         Event anEvent = (Event) session.load(Event.class,
    eventId);
(5)         aPerson.getEvents().add(anEvent);
(6)         session.getTransaction().commit();
    }
```

3-78 Metoda tworząca powiązanie

I znowu nadszedł czas na wypróbowanie nowej funkcjonalności. Przykładowy kod przedstawiony jest na listingu 3-79. Tworzymy zdarzenie (1) oraz osobę (2). Następnie dodajemy osobę do zdarzenia (3).

```
    EventManager mgr = new EventManager();

(1) Long eventId = (Long) mgr.createAndStoreEvent("Zdarzenie 01", new
    Date());
(2) Long personId = (Long) mgr.createAndStorePerson("Jan",
    "Kowalski");
(3) mgr.addPersonToEvent(personId, eventId);
    System.out.println("Dodano osobe " + personId + " do zdarzenia "
    + eventId);

    HibernateUtil.getSessionFactory().close();
```

3-79 Przykład wykorzystania powiązań w Hibernate

Jak dotąd wykorzystywaliśmy asocjację skierowaną. Teraz nadszedł czas na zastosowanie asocjacji dwukierunkowej. Dzięki temu będziemy mogli nawigować w dwie strony. W tym celu dodajemy pojemnik z osobami uczestniczącymi w zdarzeniu do klasy `Event` (listing 3-80) oraz uzupełniamy plik mapujący `Event.hbm.xml` (listing 3-81). Zwróćmy uwagę na tag `set`. Jego użycie jest prawie takie samo jak w przypadku asocjacji skierowanej. Nazwy kolumn `key` oraz `many-to-many`, dla dwóch plików mapujących, są zamienione. Dodaliśmy również atrybut `inverse="true"`.

```
public class Event {
        // [...]
        private Set participants = new HashSet();
        public Set getParticipants() {
                return participants;
        }
        public void setParticipants(Set participants) {
                this.participants = participants;
        }
}
```

3-80 Modyfikacja klasy Event w celu obsługi asocjacji dwukierunkowych

```
<hibernate-mapping>
    <class name="events.Event" table="EVENTS">
                [...]
                <set name="participants" table="PERSON_EVENT"
                inverse="true">
                    <key column="EVENT_ID"/>
                    <many-to-many column="PERSON_ID"
                class="events.Person"/>
                </set>
    </class>
</hibernate-mapping>
```

3-81 Modyfikacja pliku mapującego klasy Event w celu obsługi asocjacji dwukierunkowych

W celu kompleksowej obsługi asocjacji dwukierunkowych pozostało nam jeszcze dodać odpowiednie metody do klasy `Person` – listing 3-82.

```
public class Person {
        // [...]
        public void addToEvent(Event event) {
                this.getEvents().add(event);
                event.getParticipants().add(this);
        }
        public void removeFromEvent(Event event) {
                this.getEvents().remove(event);
                event.getParticipants().remove(this);
        }
}
```

3-82 Modyfikacja klasy Person w celu obsługi asocjacji dwukierunkowych

Warto podkreślić, że mimo zmian w plikach konfiguracyjnych oraz kodzie źródłowym schemat relacyjny nie ulega zmianie.

Podsumowując: w celu zamiany asocjacji skierowanej na dwukierunkową należy:

- Oznaczyć jedną ze stron jako `inverse`,

- Dla liczności „1 - *” musi to być strona „*”,
- Dla liczności „* - *” można wybrać dowolną stronę.

W Hibernate mamy również możliwość wykorzystania atrybutów powtarzalnych. Noszą one nazwę kolekcji wartości (*Collection of values*). Różnica w stosunku do asocjacji polega na tym, że wartości nie są współdzielone (a obiekty oczywiście tak). Dla klasy Osoba (`Person`) dodamy listę adresów e-mail. W tym celu wykorzystamy specjalny tag `element`. Oznacza on, że kolekcja nie zawiera odniesień do innych wystąpień, ale listę elementów np. typu `String`.

A zatem do dzieła. Dodajemy pojemnik z adresami e-mail do klasy Person (listing 3-83) oraz modyfikujemy plik mapujący `Event.hbm.xml` (listing 3-84).

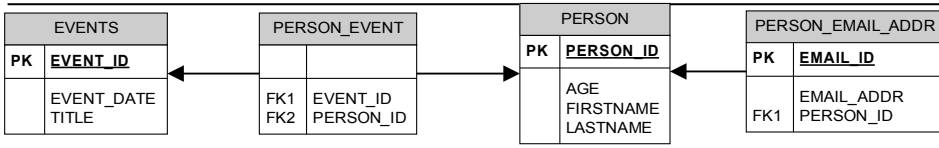
```
public class Person {
        // [...]

        private Set emailAddresses = new HashSet();

        public Set getEmailAddresses() {
                return emailAddresses;
        }

        public void setEmailAddresses(Set emailAddresses) {
                this.emailAddresses = emailAddresses;
        }
}
```

3-83 Modyfikacja klasy Person umożliwiająca przechowywanie atrybutów powtarzalnych

```
<hibernate-mapping>
    <class name="events.Event" table="EVENTS">
                [...]
                <set name="emailAddresses"
                table="PERSON_EMAIL_ADDR">
                    <key column="PERSON_ID"/>
                    <element type="string" column="EMAIL_ADDR"/>
                </set>
    </class>
</hibernate-mapping>
```

3-84 Modyfikacja pliku mapującego klasy Person umożliwiająca przechowywanie atrybutów powtarzalnych

W efekcie naszych działań rozbudowie uległ też schemat relacyjny. Jego nowa wersja jest pokazana na rysunku 3-111.



3-111 Zmodyfikowany schemat relacyjny umożliwiający zapamiętywanie atrybutów powtarzalnych

I tak szczęśliwie dobrnęliśmy do końca rozdziału traktującego o roli modelu relacyjnego w obiektowych bazach danych. Możemy pokusić się o krótkie podsumowanie.

Niezgodność impedancji, czyli trudności w pogodzeniu świata relacyjnego oraz obiektowego jest bardzo poważnym, praktycznym problemem. Można wyróżnić dwa generalne podejścia do jego rozwiązania:

- Modyfikacje języka programowania, tak aby posiadał jak najwięcej zalet baz danych. Przykładem może być Microsoft C# 3.0 razem z LINQ. Język zapytań (funkcjonalnie podobny do SQL) stał się częścią języka programowania. Niezgodność impedancji znika całkowicie, czyli nie musimy mapować konstrukcji obiektowych na relacyjne i odwrotnie. Dzięki temu mamy m.in. kontrolę typologiczną.

- Stworzenie dodatkowych bibliotek ułatwiających pracę z danymi (np. Hibernate dla Java oraz MS .NET). Biblioteka rzeczywiście ułatwia pracę z bazami danych. Niestety zamiast natywnych referencji wykorzystuje identyfikatory.

Wydaje się, że zdecydowanie lepszym podejściem jest dodanie cech baz danych do języka programowania, tak jak w Microsoft LINQ. Wymaga to jednakże zaangażowania twórców języka i raczej nie może być zrobione przez stworzenie dodatkowych bibliotek.

3.6 Użyteczność graficznych interfejsów użytkownika

Na początek małe pytanie: jaki jest najbardziej niedoceniany składnik wytwarzanego oprogramowania? Chyba jest nim interfejs użytkownika, który dla większości współczesnego oprogramowania oznacza: graficzny interfejs użytkownika (ang. *Graphical User Interface* – GUI). Dlaczego tak jest? Dlaczego programiści przeważnie skupiają się na biznesowej funkcjonalności aplikacji, traktując GUI jako zło konieczne? Jest to tym bardziej dziwne, że użytkownik ocenia aplikację właśnie przez pryzmat jej interfejsu, szcze-

gólnie w dzisiejszych czasach, gdy programów o podobnej funkcjonalności jest bardzo dużo. Który z nich użytkownik wybierze skoro ich funkcje są bardzo podobne? Prawdopodobnie ten, który będzie mu bardziej odpowiadał cenowo oraz taki, którego łatwiej się używa. I tu właśnie dochodzimy do kwestii interfejsu.

Graficzne interfejsy użytkownika są zagadnieniem z pogranicza informatyki (m.in. programowania), psychologii, ergonomii i pewnie jeszcze kilku dziedzin, których tu nie wymieniliśmy. Z racji swej interdyscyplinarności, tematyka ta jest dość skomplikowana, a co najgorsze, czasami trudno mierzalna. Mam tu na myśli, że nie zawsze jest łatwo ocenić, które rozwiązanie jest lepsze. Na szczęście, od jakiegoś czasu, w informatyce, a dokładniej w inżynierii oprogramowania, funkcjonuje pojęcie użyteczności (*usability*). Istnieje wiele portali internetowych poświęconych temu zagadnieniu. Do najciekawszych zaliczyłbym [Uzyt07] oraz [Usei07]. Dlatego też wykorzystałem część informacji tam zawartych. Ale wracając do terminu użyteczność - co on oznacza?

3.6.1 Co to jest użyteczność?

Ogólnie można powiedzieć, że użyteczność (*usability*) to zdolność do zaspokajania potrzeb użytkownika przez:

- Urządzenie,
- Aplikację,
- Interfejs.

Dotyczy kombinacji czynników kształtujących odczuwane doświadczenia użytkownika w pracy z produktem. Można do nich zaliczyć m.in.:

- Przydatność praktyczna. Czy system wykonuje zadania, które odpowiadają potrzebom użytkownika?
- Łatwość nauki i obsługi. Jak szybko można nauczyć się obsługi systemu? Czy dla większości osób obsługa systemu jest wystarczająco łatwa?
- Skuteczność. Czy system zapewni wynik zadania w takiej postaci, jak oczekuje tego użytkownik?

- Efektywność. Czy pożądany wynik osiąga się przy umiarkowanym wysiłku ze strony użytkownika?

- Zadowolenie. Czy użytkownik lubi pracować z systemem i czy rekomendowałby go innym?

Jak widzimy, czynniki te dotyczą wielu różnych aspektów, poczynając od utylitarnych (przydatność praktyczna), przez edukacyjne, aż po dość subiektywne (takie jak zadowolenie).

3.6.2 Kształtowanie użyteczności

W jaki sposób możemy kształtować użyteczność? Ogólnie możemy stwierdzić, że kluczem jest współpraca z użytkownikami. W czasie całego procesu wytwórczego musimy mieć dobry kontakt z osobami, które będą używały naszego oprogramowania. W tym celu warto:

- Przeprowadzać częste badania opinii użytkowników podczas prac nad projektem,

- Testować prototypy (bo będzie ich zapewne dość dużo) z udziałem docelowych użytkowników,

- Obserwować sposób pracy użytkowników z systemem w rzeczywistych warunkach,

- Stosować techniki ankietowe: wywiady i kwestionariusze.

3.6.3 Testowanie użyteczności

Gdy już mamy jakiś element procesu wytwórczego nadający się do testowania (zwykle będzie to jakaś wersja aplikacji, ale równie dobrze może być to papierowy projekt GUI), należy przeprowadzić testy użyteczności. Jak to zrobić? Należy kierować się poniższymi regułami:

- Osoby testujące muszą być reprezentatywne. Innymi słowy, jeżeli robimy aplikację dla prawników, którzy nie są ekspertami komputerowymi, to należy sformować właśnie taka grupą testującą. Zwykle też w tego typu sytuacjach płacimy za udział w testach (w końcu ludzie wykonują dla nas pewną pracę).

- Testy polegają na realizacji konkretnych (typowych) zadań, np. dla edytora tekstu może to być napisanie jakiegoś pisma z konkretnym sformatowaniem.

- Esencją tych testów jest obserwowanie użytkowników. Oceniamy m.in. następujące aspekty:

    - Co użytkownicy robią?

    - W których miejscach mają trudności?

- Cała procedura testowa powinna być rejestrowana. Możemy wykorzystać kamerę wideo czy specjalne programy. A najlepiej i jedno, i drugie, ponieważ samo nagranie zawartości ekranu nie dostarczy nam pełnej informacji o reakcjach użytkowników.

- Aby maksymalnie zbliżyć się do prawdziwych warunków pracy, testujący powinni być zdani tylko na siebie. Dlatego w czasie całego testu obowiązuje całkowity zakaz udzielania rad, pomagania itp.

Na szczęście dla nas, inaczej niż w badaniach statystycznych, zwykle w tego typu testach nie jest wymagana duża liczba osób. Przeważnie grupa 5-ciu, 10-ciu osób jest w stanie wychwycić główne problemy związane z użytecznością. Osoby w liczniejszych grupach przeważnie identyfikują te same problemy. Dlatego, lepiej jest iteracyjnie ulepszać projekt i testować go na małych grupach (np. 10 razy po 10 osób) niż przeprowadzić jeden test na 100 osobach.

Można też zaryzykować twierdzenie, że ważniejsze jest obserwowanie, co testujący robią, niż wysłuchanie ich późniejszego komentarza (który oczywiście też może być użyteczny). Powodem tego jest fakt, iż „bieżące" reakcje ludzi są bardziej spontaniczne. Później ich opinie mogą ulec pewnym modyfikacjom, np. na skutek faktu, że im płacimy (i nie chcą źle mówić o naszym produkcie).

3.6.4 Użyteczność niestety kosztuje

Jak widać z powyższego opisu, testy użyteczności mogą być dość drogie. Przy profesjonalnym podejściu trzeba mieć specjalne laboratorium wyposażone w stanowiska komputerowe, weneckie lustro, kamery i specjali-

styczne oprogramowanie. Czy w związku z tym to się opłaca? Wróćmy do tego, co mówiliśmy wcześniej. W dzisiejszym świecie wiele aplikacji posiada bardzo podobną funkcjonalność. W takim razie, jakimi kryteriami kierują się klienci? Na pewno biorą pod uwagę cenę, ale również łatwość wykorzystania funkcji systemu. A ten komfort przekłada się na zadowolenie z użytkowania systemu. A to, miejmy nadzieję, przełoży się na nasze zyski. Ostatnio można zresztą zauważyć pewien trend: nowe wersje aplikacji nie koncentrują się na dodawaniu nowych funkcji, ale na ułatwieniu dostępu (poprawie użyteczności) do już istniejących, np. Microsoft Office 2007. Można dyskutować, czy ten nowy interfejs nam się podoba, czy nie. Ale fakt jest faktem: główny powód podawany przez Microsoft do zakupu nowej wersji MS Office to właśnie nowy, bardziej użyteczny interfejs.

A czy są jakieś sposoby obniżenia kosztów związanych z użytecznością? Jak najbardziej! Zanim zaczniemy nowy projekt, przetestujmy jego poprzednią wersję (gdy jakaś była). Zidentyfikujmy pozytywne i negatywne elementy. Rozbudujmy te pierwsze i wyeliminujmy te drugie. Warto również przyjrzeć się rozwiązaniom konkurencji (bo zwykle jakąś mamy). W naszych pracach powinniśmy używać (wielu) prototypów. Mogą być one też „papierowe", ale w dobie narzędzi RAD nie jest problemem szybkie wytworzenie programu „udającego" pełnoprawny system. W przypadku pracy iteracyjnej każda z nich powinna kończyć się testami. No i należy zwrócić uwagę na sprawdzenie projektu w kontekście znanych zaleceń dotyczących użyteczności (patrz dalej).

A jakie mamy korzyści wynikające z wysokiej użyteczności? Na pewno możemy do nich zaliczyć:

- Krótszy czas ukończenia operacji,
- Zmniejszenie liczby błędów popełnianych przez użytkownika,
- Krótszy czas nauki i mniejszy wysiłek potrzebny do opanowania obsługi produktu,
- Zaufanie do produktu i chęć jego dalszego rozwijania,
- Trwałe zadowolenie z produktu i zamiar polecania go innym nabywcom.

Podsumujmy optymistycznie: dobra użyteczność po prostu się opłaca!

3.6.5 Zalecenia dotyczące Graficznego Interfejsu Użytkownika

Podstawowe i dość ogólne wytyczne dotyczące GUI można podsumować w następujących punktach:

- Spójność. Cała aplikacja powinna wyglądać i być obsługiwana w ten sam sposób.

- Przejrzystość i intuicyjność. W większości przypadków oznacza to nawiązywanie do istniejących zwyczajów dotyczących GUI.

- Stosowanie się do wytycznych przygotowanych dla konkretnej platformy. Każda platforma (np. Windows, Linux, Mac) ma swoje własne zasady konstruowania aplikacji. W ogólnym zarysie są one podobne, ale mogą różnić się szczegółami, np. traktowanie okien aplikacji czy wykorzystanie menu kontekstowego. Z tego powodu należy je uwzględniać. W przeciwnym wypadku, użytkownik będzie się irytował, że aplikacja pracuje i/lub wygląda niezgodnie z jego oczekiwaniami.

- Właściwe gospodarowanie przestrzenią ekranu. Zwykle sprowadza się to do właściwego skalowania okien. Naturalnie, nie zawsze ma to sens, np. gdy podajemy tylko kilka krótkich informacji (np. okno logowania), nie ma potrzeby zmiany wielkości okna. Niemniej jest to bardzo ważny aspekt oceny jakości GUI, jednakże czasami dość trudny w implementacji.

W następnej części tego podrozdziału zaprezentujemy listę kontrolną GUI (częściowo zaczerpnięta z [Uzyt07]). Warto, aby tworząc GUI, mieć ją cały czas w pamięci, a po skończeniu implementacji (lub projektowania) ponownie sprawdzić, czy wszystkie elementy naszego interfejsu są z nią zgodne. Część z tych zaleceń może wydawać się oczywista i banalna, ale widziałem systemy, które łamały każdą z tych zasad. Dlatego warto je wszystkie uważnie przeanalizować.

3.6.5.1 Wymagania dotyczące funkcjonalności

- Czy zwracasz równie dużą uwagę na funkcjonalność, jak i na wygląd aplikacji?

- Czy Twoje okno jest w 100% zgodne z założeniami? Czy odpowiada projektowi?

- Czy posiada wszystkie wyznaczone funkcje?

- Czy posiada tylko i wyłącznie wyznaczone funkcje? Nie należy umieszczać dodatkowej (nieujętej w projekcie) funkcjonalności.

- Czy pomagasz użytkownikowi w unikaniu typowych błędów poprzez umiejętne zaprojektowanie okna? Czy podanie błędnych danych jest właściwie obsługiwane?

- Czy Twoje okno zgodne jest z innymi oknami tej samej aplikacji (układ kontrolek, konwencje)?

3.6.5.2 Wymagania związane z wykorzystywaną platformą

- Czy Twój projekt jest w pełni zgodny z wytycznymi dotyczącymi danego interfejsu/systemu operacyjnego?

    - Czy wykorzystuje odpowiednie kontrolki, czcionki, kolorystykę,

    - Właściwe wymiary okien (w tym minimalne),

    - Odpowiednie ułożenie elementów,

    - Zachowanie się okien oraz kontrolek,

Odstępstwa od powyższych zaleceń mogą oznaczać błędną pracę GUI i/lub brak „systemowych" sposobów modyfikacji. W efekcie może np. nie działać skalowanie wielkości czcionek czy zmiana tematu graficznego.

3.6.5.3 Wymagania dotyczące okien

- Czy Twoje okno ma nadany tytuł?

- Czy ten tytuł jest taki sam lub zbliżony do nazwy przycisku lub opcji z menu, która otworzyła to okno?

- Okna bez lub z domyślnym tytułem są trudne do zidentyfikowania, np. w pasku zadań.

- Czy w pasku tytułu obecne są odpowiednie przyciski (m.in. minimalizuj, maksymalizuj, przywróć, pomoc)? Jeżeli tak, to czy właściwie działają?

- Czy możliwa jest zmiana wielkości (maksymalizacja, minimalizacja) okna? Jeśli tak, czy taka możliwość jest uzasadniona? Czy cała zawartość okna jest właściwie skalowana?

- Czy widoczny jest przycisk kontekstowej pomocy?

- Czy dobrano odpowiedni rodzaj ramki do okna (np. *dialog*)?

- Czy Twoje okno jest odpowiednio umieszczone w hierarchii?

- Czy stosujesz modalność (blokowanie interakcji z programem przy użyciu innych okien), tam gdzie jest ona wymagana?

3.6.5.4 Wymagania dotyczące zarządzania oknami dialogowymi

- Czy w Twoim oknie dialogowym jest obecny domyślny przycisk? Czy jest on oznaczony?

- Czy Twoje okno da się zamknąć w standardowy sposób? Np.:

    - przycisk na pasku tytułu,

    - klawisz Esc,

    - przycisk „Anuluj”.

- Czy jest możliwe zamknięcie okna lub wybranie „Anuluj” bez dokonywania zmian?

- Czy przyciski „OK”. „Anuluj” i inne odnoszące się do manipulacji okienkiem są oddzielone od innych przycisków?

Pamiętaj, aby użytkownik miał zawsze wyjście awaryjne z każdej, nawet najdziwniejszej sytuacji.

3.6.5.5 Wymagania dotyczące kontrolek

- Czy wykorzystujesz intuicyjne ułożenie kontrolek i dobierasz odpowiednio ich typy, zamiast w sztuczny sposób opisywać czynności i tworzyć instrukcje?
- Czy kontrolki rozmieszczone są zgodnie ze sposobem, w jaki użytkownik czyta lub skanuje ekran?
- Czy najczęściej wykorzystywane opcje są umieszczone „najwcześniej”?
- Czy liczba kontrolek w oknie nie jest za duża?
- Jeśli tak, to czy nie lepiej usunąć niektóre rzadziej wykorzystywane lub pokazywać je dopiero po wciśnięciu odpowiedniego przycisku?
- Czy stosujesz kontrolki zgodnie z ich przeznaczeniem opisanym w wytycznych i wykorzystywanym w popularnych aplikacjach? Np.
    - Wykluczające się opcje: *radio buttons*,
    - Wiele możliwości: *check box*y.
    - Listy z możliwością jedno- lub wielokrotnego wyboru.
- Czy Twoje kontrolki reagują w standardowy sposób? Np.:
    - przyciski po wciśnięciu powinny wykonywać komendę lub otwierać okno,
    - przyciski radiowe czy *check box*y powinny tylko zmieniać stan.
- Czy kontrolki zbliżone znaczeniowo są pogrupowane z wykorzystaniem ramek lub pokrewnych elementów interfejsu?

- Czy nie stosujesz ramek nadmiernie, aby grupować tylko pojedyncze kontrolki?

- Czy oprogramowanie odpowiednio włącza i wyłącza nieaktywne kontrolki w razie potrzeby?

3.6.5.6 Wymagania dotyczące menu

- Czy w kontrolkach z menu (oraz w *combo boxes*) jest domyślnie wybrana jakaś opcja?

- Czy otwierająca się lista jest odpowiednio duża (długa) w pionie, tak aby oszczędzić użytkownikowi przewijania?

- Czy otwierająca się lista jest odpowiednio szeroka, tak aby każda pozycja była widoczna w całości?

3.6.5.7 Wymagania dotyczące podpisów

- Czy przy kontrolkach są odpowiednie opisy (*labels*) tłumaczące ich przeznaczenie (szczególnie przy listach i menu)?

- Czy wszystkie wyświetlane dane są opisane?

- Czy opisy są tak samo wyrównane (w stosunku do tekstu kontrolki i do innych opisów)?

- Czy dwukropki przy opisach są konsekwentnie stosowane lub niestosowane?

- Czy stosujesz etykietki (*tooltips*) wyjaśniające niektóre elementy interfejsu?

- Pamiętaj, że etykietki (*tooltips*) powtarzające standardowy opis kontrolki nie mają większego sensu.

- Czy stosujesz polskie litery? Opisy i nazwy kontrolek bez polskich liter wyglądają niechlujnie i są mniej czytelne.

3.6.5.8 Wymagania dotyczące pracy z klawiaturą

- Czy w oknie zastosowano skróty klawiszowe, dzięki którym zaawansowani użytkownicy będą mogli szybciej skorzystać z niektórych opcji?
- Czy litery wywołujące opcje nie powtarzają się?
- Czy użytkownik może przemieszczać kursor za pomocą klawiszy *Tab* i *Shift-Tab*?
- Czy kolejność przemieszczania jest zgodna z układem okienka?
- Czy można się w ten sposób dostać do wszystkich kontrolek?

3.6.6 Jakość interfejsu graficznego

Niestety trzeba się liczyć z tym, że interfejs graficzny zahacza trochę o gusta oraz upodobania użytkowników. W związku z tym nie zawsze da się obiektywnie zmierzyć jego użyteczność czy porównać różne rozwiązania.

Na szczęście inżynieria oprogramowania wypracowała pewne metody, które czasami pomagają zmierzyć jakość interfejsu oraz urządzenia wskazującego. Jedną z nich jest tzw. eksperyment Fittsa oraz prawo nazwane jego imieniem. W skrócie mówi ono tyle, że czas wybrania celu (np. przycisku) zależy od odległości i wielkości celu. Czyli np. umieszczając element GUI znajdujący się dość daleko od centrum okna, postarajmy się, aby był większy (chyba że nie ma większego znaczenia). Więcej na ten temat można znaleźć np. w wikipedii: *http://en.wikipedia.org/wiki/Fitts'_law*.

Oczywiście w ocenie jakości interfejsu bardzo duże, a wręcz kluczowe znaczenie mają testy użyteczności (patrz podrozdział 3.6.3 na stronie 251).

3.7 Projekt dla wypożyczalni wideo

I oto wreszcie dobrnęliśmy do kluczowego podrozdziału dotyczącego projektowania. Zajmiemy się w nim opracowaniem projektu dla naszej wypożyczalni wideo. Będzie on obejmował kilka diagramów, z których najważniejszy to diagram klas. Kolejne podrozdziały będą właśnie poświęcone stworzeniu tych elementów.

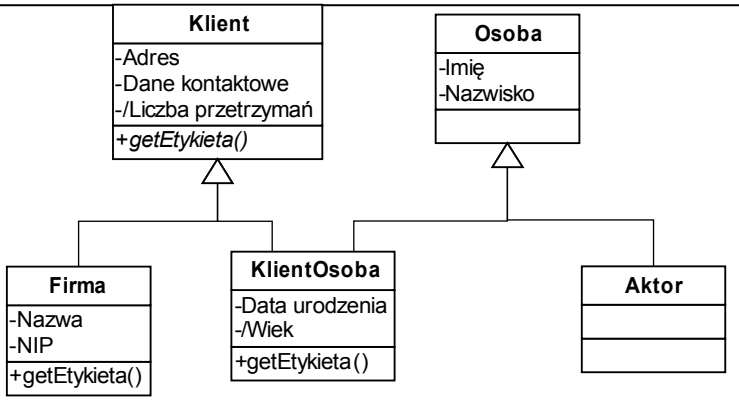
3.7.1 Diagram klas dla wypożyczalni wideo

Przypomnę krótko, że nasz projekt musi obejmować przekształcenie diagramu klas, tak aby był „implementowalny" w naszym języku programowania. Innymi słowy, musimy usunąć lub zastąpić wszelkie konstrukcje znajdujące się na diagramie, a nieistniejące w języku Java (bo w nim będzie odbywała się implementacja). Część elementów niewystępujących w Javie, uda nam się „automatycznie" zaimplementować przy użyciu klas opracowanych w poprzednich rozdziałach. Mam tu na myśli poniższe klasy oraz wsparcie dla konstrukcji:

- `ObjectPlus`: zarządzanie ekstensją, trwałość - podrozdział 3.1.7 na stronie 124,
- `ObjectPlusPlus`: obsługa powiązań i częściowo kompozycji – podrozdział 3.2.4 na stronie 161,
- `ObjectPlus4`: wsparcie dla niektórych rodzajów ograniczeń - podrozdział 3.4.2 (i dalsze) na stronie 209.

Elementy wspierane przez powyższe klasy umieszczamy na diagramie, tak jakby występowały w Javie. Abyśmy mogli korzystać z tej funkcjonalności, wszystkie nasze klasy biznesowe muszą dziedziczyć z `ObjectPlus4`. Pokazanie tego na diagramie mocno by go skomplikowało. Dlatego nie będziemy tego robić, pamiętając o takiej konieczności na etapie implementacji.

Ostateczny diagram klas z fazy analizy jest pokazany na rysunku 2-66 (strona 93). Jest on dla nas podstawą do dalszych prac. Analogicznie jak przy okazji jego tworzenia (podrozdział 2.4.6, strona 60), będziemy analizowali poszczególne elementy (wtedy były to wymagania) i rozważali potencjalne sposoby przekształcania. Nie w każdym przypadku takie zmiany będą konieczne. Na nasze szczęście, niektóre części diagramu mogą zostać przeniesione bez żadnych zmian. Poniżej będziemy umieszczać fragmenty diagramu razem z komentarzami. A zatem – do pracy.



3-112 Tworzenie projektowego diagramu klas – krok 1

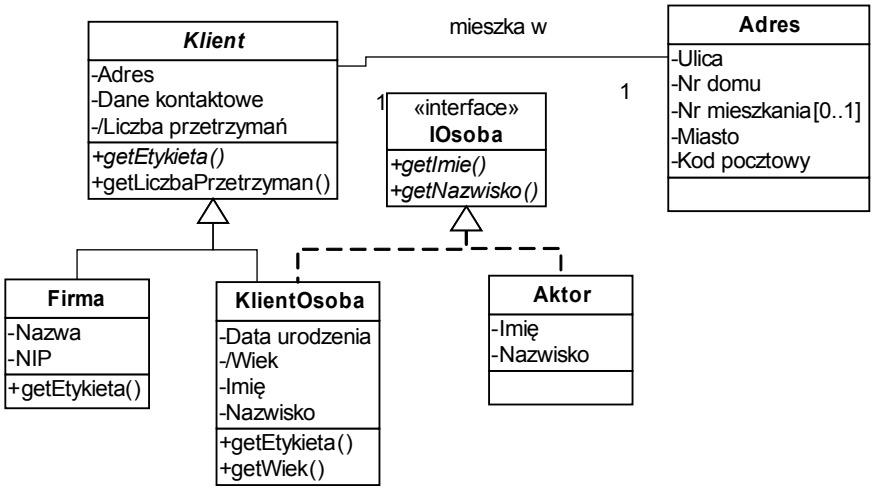
- Rysunek 3-112

I już na samym początku mamy problem. Jaki? Dziedziczenie wielokrotne, które nie występuje w Javie. `KlientOsoba` dziedziczy i z klasy `Klient` i z klasy `Osoba`. A jak pamiętamy z podrozdziału 3.3.5 (strona 190) takowe w Javie nie występuje. Co z tym możemy zrobić? Możliwości jest kilka (patrz podrozdział 3.3.5 na stronie 190). Jednak w tym przypadku myślę, że zdecydujemy się na podział tej jednej hierarchii na dwie niezależne od siebie. Dzięki temu unikniemy komplikacji związanych z obsługą kompozycji. Dodatkowo `KlientOsoba` oraz `Aktor` będą implementowały interfejs `IOsoba`. Musimy jeszcze jakoś poradzić sobie z atrybutami wyliczalnymi `/Liczba ptzetrzymań` (klasa `Klient`) oraz `/Wiek` (klasa `KlientOsoba`). W tym celu stworzymy specjalne metody o nazwach, jak się łatwo domyślić, `getLiczbaPrzetrzyman()` oraz `getWiek()`.

Czy to już koniec? Prawie... Nie widać tego bezpośrednio na diagramie (bo nie ma do tego specjalnej notacji), ale musimy podjąć jeszcze jedną decyzję. Jakieś sugestie? No tak – mamy atrybut złożony przechowujący adres. Mamy kilka możliwości, ale najporządniejsza wydaje się ta polegająca na wprowadzeniu dodatkowej klasy i połączeniu ją asocjacją z klientem. W takiej sytuacji trzeba określić liczności. Możemy pozwolić, aby kilku klientów mieszkało pod tym samym adresem (mało

prawdopodobne), albo zdecydować się na „1 – 1”. I myślę, że właśnie tak postąpimy. Jeszcze kwestia atrybutu opcjonalnego w numerze mieszkania. Możemy się umówić, że `0` oznacza brak wartości (bo raczej się takich numerów nie stosuje). Dzięki temu unikniemy konieczności stosowania klasy opakowującej (podrozdział 3.1.4.3 na stronie 106).

Czyli po naszych modyfikacjach odpowiedni fragment diagramu wyglądałby tak jak na rysunku 3-113.

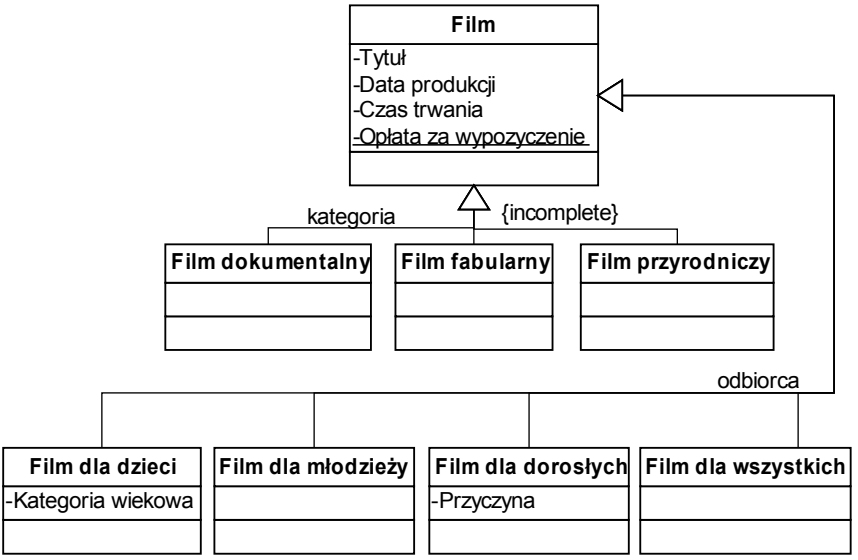


3-113 Tworzenie projektowego diagramu klas – krok 2

- Rysunek 3-114

Kolejnym fragmentem, który musimy przeanalizować, jest ten dotyczący rodzajów filmów znajdujących się w naszej wypożyczalni. Od razu widać, że i tu będziemy musieli coś pozmieniać. Jak wiemy, w Javie nie występuje dziedziczenie wieloaspektowe, więc trzeba będzie je jakoś przekształcić. Jakie mamy możliwości? Ich paleta została przedstawiona w podrozdziale 3.3.6 na stronie 196. W tej konkretnej sytuacji musimy pozbyć się jednej z hierarchii dziedziczenia. Pytanie tylko której? Tak jak pisaliśmy wcześniej, zostawiamy tę, gdzie bardziej korzystamy z dobrodziejstw generalizacji. W tym przypadku zostawimy aspekt „odbiorca”. Jak w takim ra-

zie zapamiętamy informacje z aspektu „kategoria”? Myślę, że najlepiej będzie stworzyć nową klasę, nazwać ją właśnie „KategoriaFilmu” i połączyć asocjacją z filmem. Przy okazji również załatwimy ograniczenie {incomplete}. Dzięki połączeniu asocjacją, w czasie działania systemu, bez problemu dodamy nowe kategorie (będą to po prostu nowe instancje klasy `KategoriaFilmu`).



3-114 Tworzenie projektowego diagramu klas – krok 3

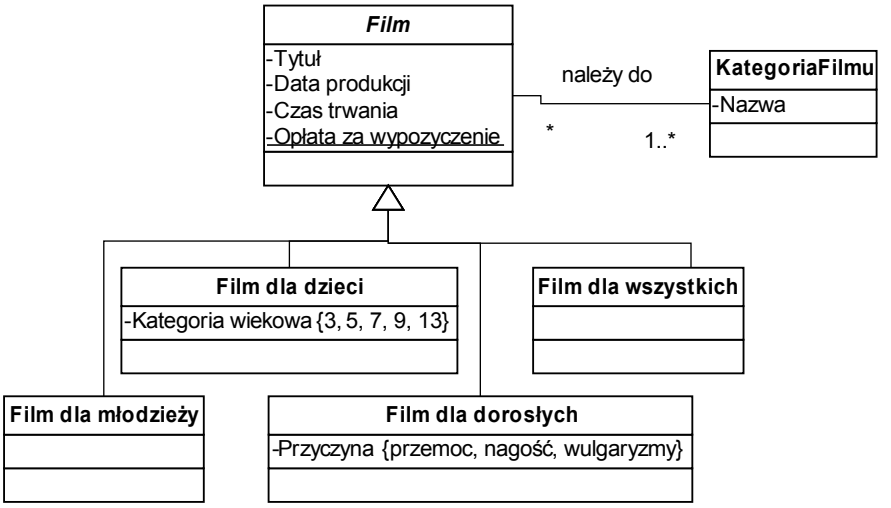
W klasie `Film dla dzieci` znajduje się atrybut `Kategoria wiekowa`. Zgodnie z wymaganiami definiowany jest przy pomocy określonych liczb: 3, 5 itd. Czyli będziemy po prostu zapamiętywali jakąś liczbę.

Natomiast w klasie Film dla dorosłych musimy przechowywać przyczynę takiej kategoryzacji. Przyczyny te są zdefiniowane i nie zmieniają się w czasie działania programu. W związku z tym możemy wykorzystać typ wyliczeniowy.

Pozostało nam już tylko określić liczności. Konkretny film może należeć do kilku kategorii (np. dokumentalny i przyrodniczy). No i kategoria może być połączona

z wieloma filmami. Innymi słowy liczność będzie „* - *”.

Odpowiedni fragment diagramu po modyfikacjach (przy okazji też oznaczyliśmy klasę Film jako abstrakcyjną) przedstawia rysunek 3-115.

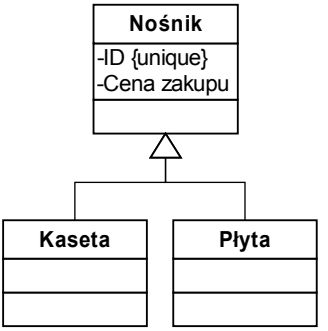


3-115 Tworzenie projektowego diagramu klas – krok 4

- Rysunek 3-116

Teraz zajmiemy się sposobem przechowywania informacji o nośnikach wypożyczanych w naszym systemie. W fazie analizy zdecydowano, że do tego celu będzie wykorzystywana hierarchia klas. Jak widzimy, podklasy `Kaseta` oraz `Płyta` są „puste”. Jak wiemy z poprzednich rozdziałów, tak naprawdę nie możemy ich po prostu usunąć, ponieważ przechowują informację o rodzaju nośnika. Gdyby ktoś jednak bardzo chciał się ich pozbyć, to musi znaleźć inny sposób na pamiętanie tych informacji. Może być nim np. stworzenie klasy `Rodzaj` i połączenie jej z klasą `Nosnik`. Tego typu rozwiązanie umożliwia również dynamiczne (w czasie działania systemu) dodawanie informacji o nowych nośnikach, np. *BlueRay*. Podkreślmy jeszcze raz: zastąpienie dziedziczenia za pomocą klasy „słownikowej” jest możliwe tylko

dzięki temu, że w podklasach nie przechowujemy informacji specyficznych dla konkretnych rodzajów nośników. Ponieważ w naszym systemie nie występuje taka potrzeba oraz zyskujemy możliwość dynamicznego modyfikowania rodzajów nośników, zdecydujemy się na rozwiązanie z klasą słownikową.

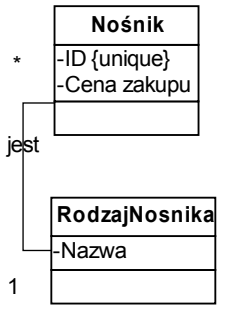


3-116 Tworzenie projektowego diagramu klas – krok 5

Musimy się jeszcze jakoś odnieść do ograniczenia {unique}. Na szczęście jest ono na tyle proste w realizacji, że każdy programista sobie z tym poradzi i nie musimy go uwzględniać w projekcie.

Warto jeszcze zaakcentować zmianę nazwy. Patrząc na diagram, od razu widzimy, że klasa `Rodzaj` opisuje rodzaj nośnika. Niestety w kodzie programu, gdy natkniemy się na klasę o nazwie `Rodzaj`, nie będziemy wiedzieli czego dotyczy ten rodzaj. Z tego powodu warto zmienić jej nazwę na `RodzajNosnika`.

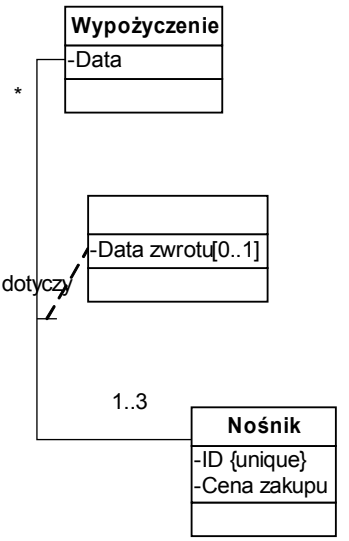
Nowa wersja diagramu pokazana jest na rysunku 3-117.



3-117 Tworzenie projektowego diagramu klas – krok 6

- Rysunek 3-118

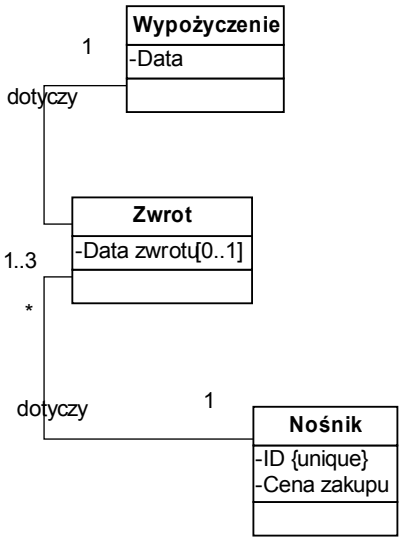
Informacje dotyczące wypożyczenia opisywane za pomocą asocjacji z atrybutem. Jak ustaliliśmy w podrozdziale 3.2.3.3 (strona 147), taka konstrukcja nie występuje w języku Java. W związku z tym wprowadzamy klasę pośredniczącą. Nazwijmy ją zwrot. Co będzie zawierać? Oczywiście datę zwrotu. Ale czy w momencie tworzenia powiązania będziemy ją znali? Ponieważ mówimy o faktycznej dacie, a nie o zakładanej, to odpowiedź brzmi: nie. Czyli, podobnie jak na diagramie pierwotnym, atrybut będzie opcjonalny. Teoretycznie moglibyśmy tego uniknąć i w momencie wypożyczania tworzyć bezpośrednie powiązanie pomiędzy instancją klasy `Wypożyczenie` a `Nośnik`. Następnie przy zwrocie usuwać je i łączyć te klasy za pośrednictwem klasy `Zwrot`. Ale wydaje mi się to trochę zbyt skomplikowane i tak naprawdę nic nie wnosi do naszego projektu. Dlatego proponuję pozostać przy naszym pierwotnym rozwiązaniu (tym z klasą pośredniczącą).



3-118 Tworzenie projektowego diagramu klas – krok 7

Po dodaniu nowej klasy i utworzeniu dwóch asocjacji musimy określić ich liczności oraz nazwy. W tej konkretnej sytuacji problemem może być wymyślenie dobrych nazw. Tym razem proponuję pójść „na skróty" i powielić słowo-wytrych „dotyczy".

Zmodyfikowany diagram pokazany jest na rysunku 3-119.



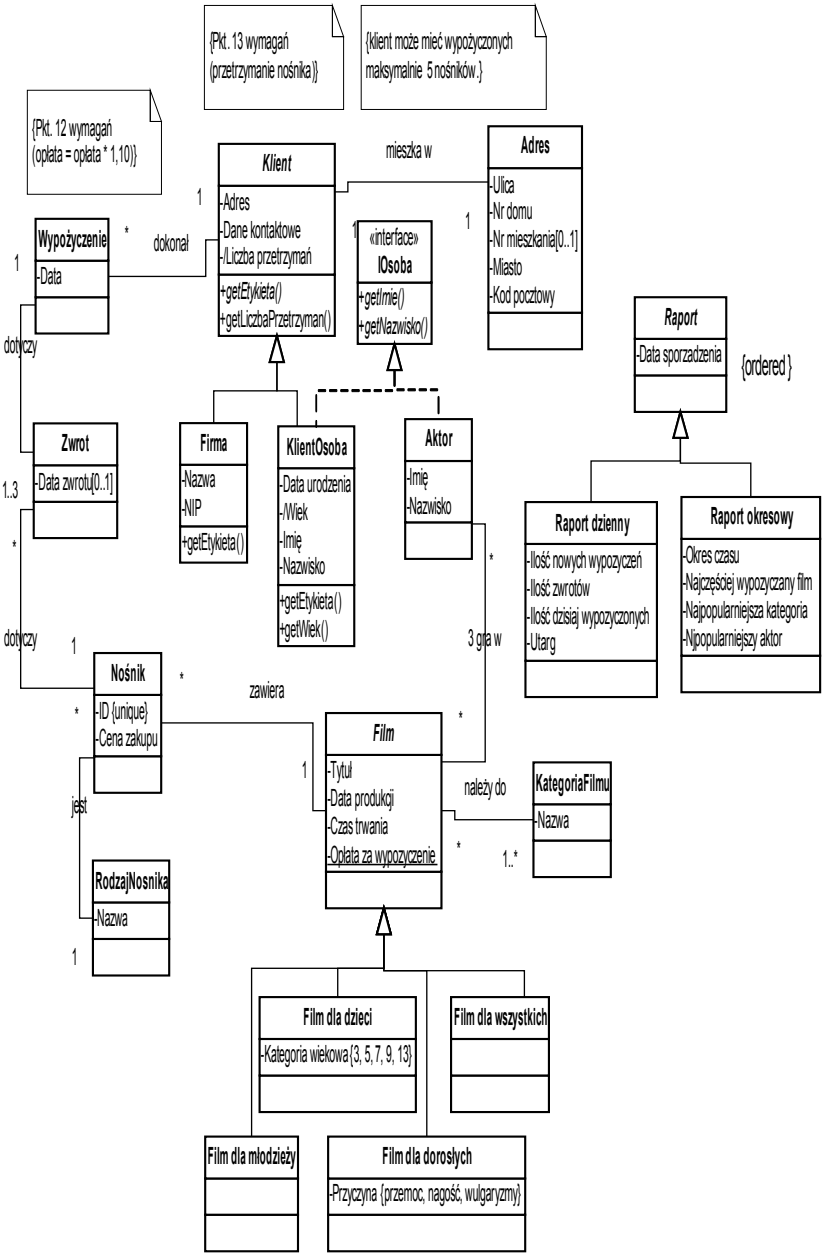
3-119 Tworzenie projektowego diagramu klas – krok 8

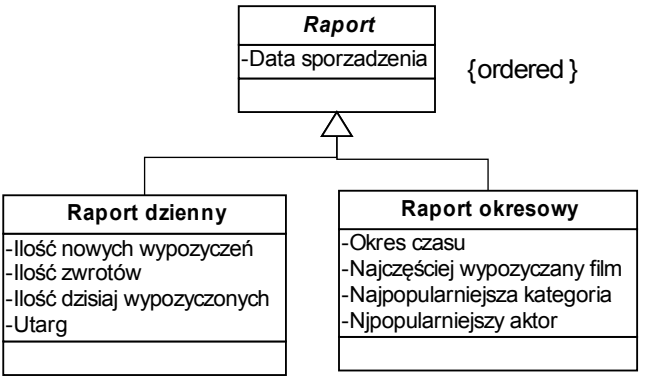
- Rysunek 3-120

Ostatnim elementem diagramu, któremu powinniśmy się przyjrzeć, jest część dotycząca raportów. Jak wspomnieliśmy w komentarzu do rysunku 2-66 (strona 93), dotyczącym właśnie raportów, w celu ich generowania będziemy posługiwali się metodami klasowymi.

Ograniczenie {ordered} obsłużymy przez dodanie (na etapie implementacji) dedykowanej struktury, przechowującej raporty w określonej kolejności.

Zmodyfikowany diagram pokazany jest na rysunku.


{Pkt. 13 wymagań (przetrzymanie nośnika)}
{klient może mieć wypożyczonych maksymalnie 5 nośników.}
{Pkt. 12 wymagań (opłata = opłata * 1,10)}
Klient
-Adres
-Dane kontaktowe
-Liczba przetrzymań
+getEtykieta()
+getLiczbaPrzetrzyman()
mieszka w
Adres
-Ulica
-Nr domu
-Nr mieszkania[0..1]
-Miasto
-Kod pocztowy
Wypożyczenie
-Data
dokonał
«interface»
IOsoba
+getImie()
+getNazwisko()
dotyczy
Zwrot
-Data zwrotu[0..1]
Firma
-Nazwa
-NIP
+getEtykieta()
KlientOsoba
-Data urodzenia
-Wiek
-Imię
-Nazwisko
+getEtykieta()
+getWiek()
Aktor
-Imię
-Nazwisko
Raport
-Data sporządzenia
{ordered}
Raport dzienny
-Ilość nowych wypozyczeń
-Ilość zwrotów
-Ilość dzisiaj wypozyczonych
-Utarg
Raport okresowy
-Okres czasu
-Najczęściej wypozyczany film
-Najpopularniejsza kategoria
-Nipopularniejszy aktor
Nośnik
-ID {unique}
-Cena zakupu
zawiera
Film
-Tytuł
-Data produkcji
-Czas trwania
-Opłata za wypozyczenie
3 gra w
należy do
KategoriaFilmu
-Nazwa
jest
RodzajNosnika
-Nazwa
Film dla dzieci
-Kategoria wiekowa{3, 5, 7, 9, 13}
Film dla wszystkich
Film dla młodzieży
Film dla dorosłych
-Przyczyna {przemoc, nagość, wulgaryzmy}

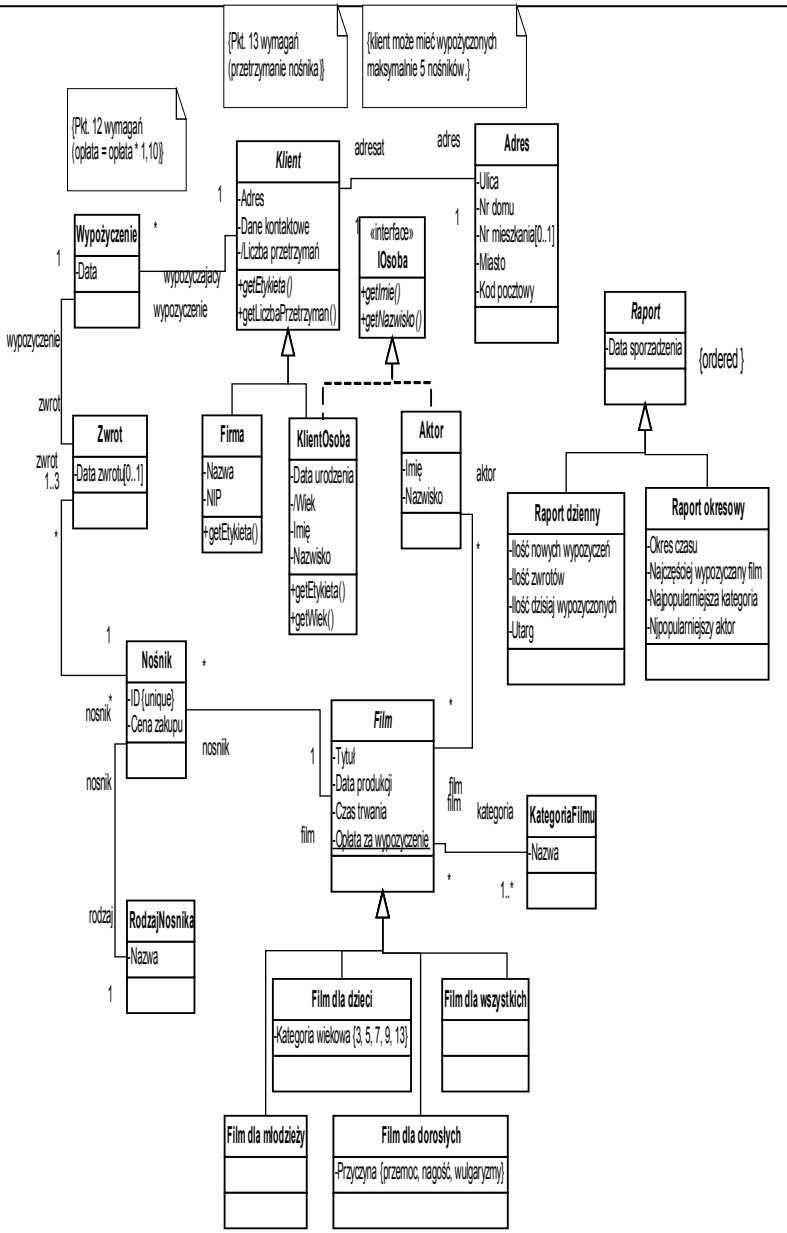


3-120 Tworzenie projektowego diagramu klas – krok 9

I tym oto sposobem prawie dobrnęliśmy do końca naszego projektu dotyczącego struktury przechowywanych danych – rysunek 3-121.

Musimy jeszcze tylko stworzyć wersję naszego diagramu, która będzie zawierała nazwy ról zamiast nazw asocjacji. Będzie to potrzebne przy implementacji, ponieważ musimy jakoś odnosić się do klas docelowych z punktu widzenia klas źródłowych. Taki diagram pokazany jest na rysunku 3-122.

3-121 Tworzenie projektowego diagramu klas – krok 10



3-122 Diagram klas wypożyczalni wideo z nazwami ról (bez metod)

3.7.2 Projekt działania systemu

To, co do tej pory zrobiliśmy, określa, jak nasz system będzie przechowywał dane. Niestety prawie nic nie mówi, jak ma działać. Działanie systemu jest „zaszyte" w jego metodach. Co prawda, znajdują się one na diagramie klas, ale nie wynika z niego, jak będą wywoływane. Innymi słowy, nie jesteśmy w stanie precyzyjnie ustalić, która metoda woła którą metodę w celu wykonania jakiejś akcji zleconej przez użytkownika.

Aby to określić, musimy skorzystać z innego rodzaju diagramów. Mamy kilka możliwości, ale chyba najczytelniejsze będą diagramy sekwencji.

Należą one do kategorii diagramów dynamicznych (razem z diagramami stanu oraz aktywności). Krótko objaśnijmy, z czego się składają oraz jak należy je wykorzystywać (więcej informacji można znaleźć np. w [Fowl04] lub [Płod05]):

- Diagram przedstawia sekwencję wołania metod.
- Jego czytanie rozpoczynamy od góry i kontynuujemy zgodnie z pionową osią czasu.
- W poziomie pokazane są poszczególne obiekty razem z wykonywanymi metodami.
- Na przedstawione czynności (głównie wołania metod) możemy nakładać różnego rodzaju ograniczenia i warunki, tak aby jak najdokładniej doprecyzować przyszłą implementację.

Przypomnijmy sobie diagram aktywności dla dodawania klienta (rysunek 2-67, strona 94). Teraz na jego podstawie zaplanujemy cały proces bardziej szczegółowo – tak aby dało się go w miarę precyzyjnie zaimplementować. Wynikowy diagram sekwencji został pokazany na rysunku 3-123 Zwróćmy uwagę na kilka elementów:

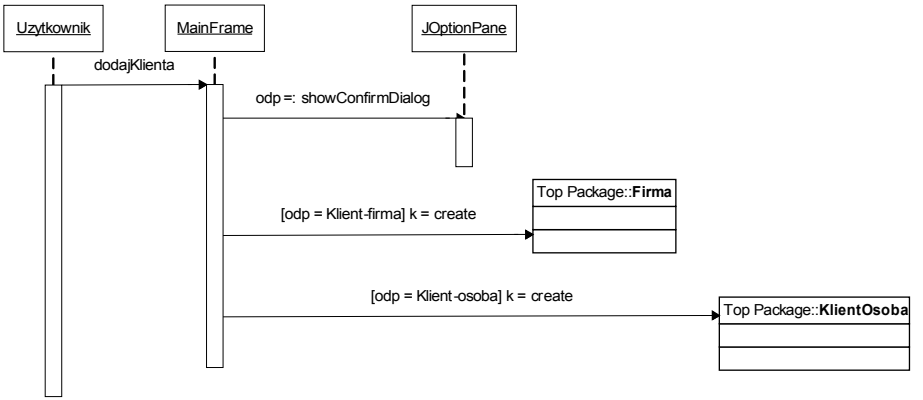
- pierwsza metoda, która zostanie wywołana, to `dodajKlienta` z klasy `MainFrame`. Tak naprawdę zachodzi tu małe oszustwo, ponieważ musimy połączyć dwa światy: rzeczywistego użytkownika oraz wnętrze programu komputerowego. Tego typu wołanie metody zwykle odbywa się za pomocą GUI, a więc prawdopodobnie wykorzystany

zostanie jakiś widget sterowany zdarzeniami, procedura obsługi itp. Nie pokazujemy tego, aby nie zaciemniać obrazu sytuacji.

- Metoda showConfirmationDialog zwraca nam informację na temat rodzaju klienta, którego chce dodać użytkownik.

- Na tej podstawie (w zależności od spełnienia warunku) utworzymy instancję klasy `Firma` lub `KlientOsoba`.

- Chyba że użytkownik anuluje całą akcję i nie zostanie wywołana żadna z metod.



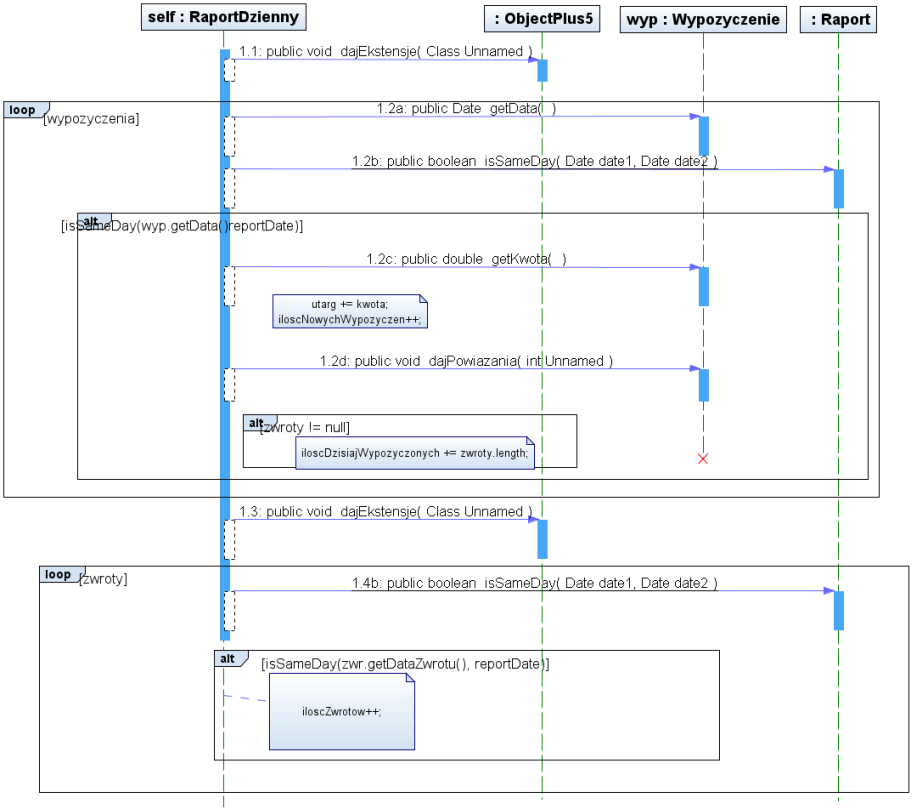
3-123 Diagram sekwencji ilustrujący dodawanie nowego klienta

Teoretycznie tego typu diagramy należy utworzyć dla opisu działania całego systemu. W praktyce robi się to dla ważniejszych (i/lub bardziej skomplikowanych) fragmentów projektowanego systemu. Oczywiście im dokładniejszy wykonamy projekt (tzn. im więcej stworzymy diagramów), tym łatwiejszą będzie miał pracę programista. I nie chodzi tu o lenistwo, ale o jednoznaczne działanie aplikacji. Im mniejsza dowolność w implementacji (przez programistę), tym większa zgodność z efektami pracy analityka oraz projektanta. A w efekcie z oczekiwaniami klienta.

Pokażmy jeszcze jeden przykładowy diagram (rysunek 3-124), tym razem precyzujący sposób generowania raportu dziennego (źródłowy diagram aktywności jest na rysunku 2-68, strona 95). Ze względu na oszczędność miejsca nie pokazano aktora inicjującego wołanie metody `generate()` z klasy `RaportDzienny` (a tylko jej wnętrze).

Opiszmy pokrótce co zawiera diagram:

- Najpierw pobieramy ekstensję klasy Wypożyczenie (1.1).
- Dzięki temu będziemy w stanie wybrać wypożyczenia, które odbyły się interesującego nas dnia (1.2a, 1.2b).
- Jeżeli dane wypożyczenie nastąpiło określonego dnia, to zwiększamy dzienny utarg oraz liczbę nowych wypożyczeń.
- Następnie dla konkretnego wypożyczenia (wyp) sprawdzamy liczbę wypożyczonych nośników (1.2d). Jeżeli informacja jest różna od null, to zwiększamy liczbę dzisiaj wypożyczonych.
- Analogicznie postępujemy ze zwrotami.



3-124 Diagram sekwencji ilustrujący generowanie raportu dziennego

Zwróćmy uwagę, że część informacji musieliśmy pokazać nieformalnie – za pomocą komentarzy (notatek). Jest tak dlatego, że diagram sekwencji nie ma odpowiedniej notacji do definiowania operacji na lokalnych (dla danej metody) zmiennych. Ten brak występuje tylko wtedy, gdy taka operacja nie jest związana z wołaniem jakieś metody.

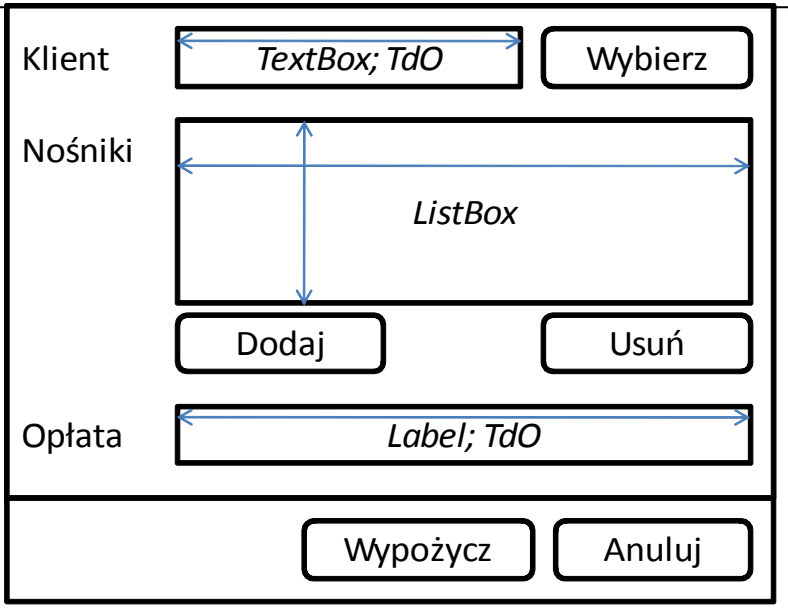
Warto wspomnieć jeszcze o jednej sprawie. Niektórzy twierdzą, że nie warto tworzyć wielu, stosunkowo niewielkich metod, które ze sobą współpracują. Zamiast tego preferują kilka dużych realizujących jakąś funkcjonalność od początku do końca. Dlaczego to nie jest dobre podejście? Otóż podstawowy argument jest taki, że większość tych małych funkcji, których niektórzy nie chcą tworzyć, jest wielokrotnie wykorzystywana z wielu różnych miejsc programu. Stąd się właśnie bierze takie wzajemne wywoływanie me-

tod, doskonale widoczne na diagramach sekwencji. Dzięki temu nie musimy powielać kodu oraz wielokrotnie wykonywać tej samej pracy. Dodatkowym argumentem jest fakt, iż dużo łatwiej przeanalizować 10 metod np. po 20 linii każda niż jedną mającą 200 linii. Powstały nawet specjalne metryki mierzące jakość kodu, między innymi na podstawie rozmiaru oraz liczby metod w klasach.

3.7.3 Projekt interfejsu użytkownika

Projekt graficznego interfejsu użytkownika może być wykonany na kilka sposobów. Niezależnie jednak od przyjętej metody należy przestrzegać zaleceń dotyczących użyteczności (podrozdział 3.6, strona 249). Najbardziej popularne podejścia wykorzystują diagramy zawierające wzory formatek (formularzy, okien). Znajdują się na nich poszczególne widgety (*visual gadgets*; elementy GUI) przedstawione w postaci figur geometrycznych czy istniejących kotrolek (np. Swing czy Windows Forms). Projekt powinien zawierać informacje dotyczące:

- rozmieszczenia poszczególnych elementów,
- ich opisu,
- typu widgetu (np. *Combo Box*, *List Box*),
- sposobu reakcji na zmiany wielkości okna (jest to bardzo ważny aspekt, niestety dość często lekceważony),
- ewentualnego wykorzystania podpowiedzi (*tooltips*),
- ewentualnych informacji dodatkowych, np.
    - kontrolka tylko do odczytu,
    - sposób zaznaczania danych, np. w *ListBox* można pozwolić na wielokrotny wybór,
    - zachowanie się okna (modalność).



3-125 Projekt formularza umożliwiającego wypożyczanie nośników

Rysunek 3-125 przedstawia projekt formularza umożliwiającego wypożyczanie nośników klientom naszej wypożyczalni. Przy jego tworzeniu przyjęliśmy kilka założeń dotyczących notacji. Naturalnie nie jest to żaden formalny standard, ale pewna arbitralna decyzja. W zależności od potrzeb, można dokonać własnych modyfikacji, w szczególności opisy słowne zastąpić ikonkami, czy kolorami. Przyjęte założenia:

- Czcionką pochyłą opisaliśmy typy kontrolek, które chcemy wykorzystać. Warto zwrócić uwagę, że np. do wyświetlenia nazwy klienta może równie dobrze być wykorzystany np. *Label*.

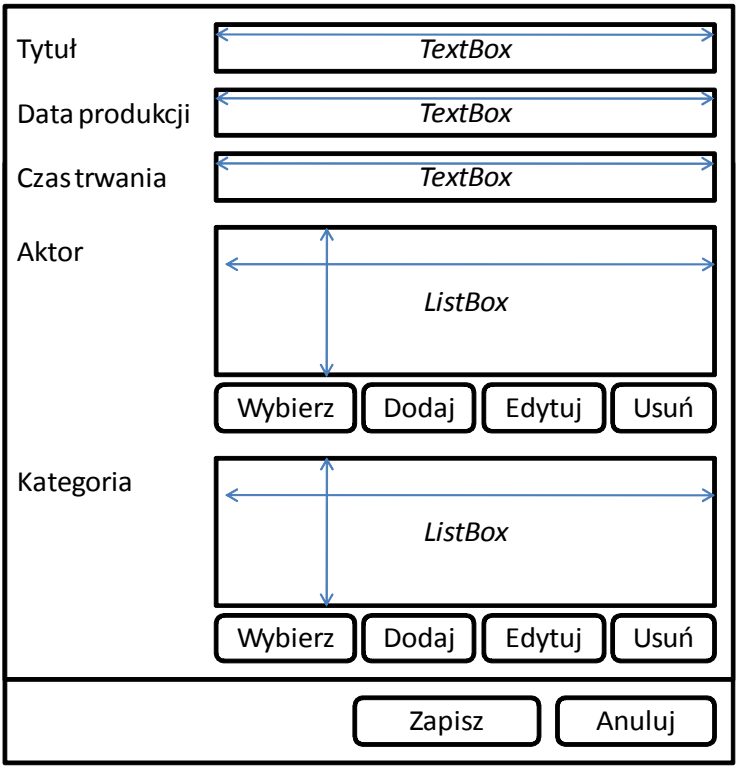
- Skrót TdO oznacza „Tylko do Odczytu”. Innymi słowy, użytkownik nie jest w stanie bezpośrednio modyfikować zawrtości danego widgetu, np. aby zmienić klienta, trzeba go wybrać za pomocą odpowiedniej funkcji. Analogicznie, opłata za wypożyczenie jest naliczana przez system i nie może być ręcznie modyfikowana.

- Prostokąty z zaokrąglonymi rogami oznaczają przyciski.

- Cienkie linie ze strzałkami pokazują, jak kontrolka ma reagować na zmiany wielkości całego okna:

    - pojedyncza pozioma linia oznacza, że kontrolka tylko się rozszerza (nie zmienia swojej wysokości),

    - linie pojedyncza oraz pozioma mówią nam, że widget powinien zmieniać swoją wysokość oraz szerokość. Pamiętajmy, że gdyby takie oznaczenie wystąpiło na więcej niż jednym elemencie, konieczne byłoby ustalenie wzajemnych zależności przy skalowaniu okna, np. proporcjonalnie (50% - 50% czy 80% - 20%).



3-126 projekt okna umożliwiającego dodanie informacji o filmie

W zależności od „typowości” danego okna konieczne może być uzupełnienie projektu dodatkowymi oznaczeniami lub opisem słownym.

Tego typu projekty najlepiej jest wykonać dla wszystkich okien występujących w aplikacji. Dzięki temu unikniemy niespodzianek w postaci nieprzemyślanych formatek. Wyjątkiem może być skorzytanie z biblioteki, która automatycznie generuje GUI na podstawie modelu danych. W takiej sytuacji ogólne reguły tworzenia poszczególnych elementów są „zaszyte” wewnątrz biblioteki. Tego typu podejście znacząco redukuje nakład pracy konieczny do zaprojektowania oraz zaimplementowania GUI. Zwykle odbywa się to kosztem generyczności rozwiązania. Wygenerowane okna są dość podobne do siebie i nie zawsze mogą być odpowiednie dla konkretnego rodzaju aplikacji. Dlatego tego typu podejście najlepiej sprawdza się w typowych, biznesowych zastosowaniach.

Rysunek 3-126 zawiera projekt jeszcze jednego okna o nieco bardziej skomplikowanej konstrukcji. Występują tam dwa elementy, które zmieniają swój rozmiar w pionie oraz poziomie (lista aktorów oraz kategorii). Zwróćmy uwagę, że dane, które wyświetlają, są powiązane z obiektem klasy `Film` za pomocą asocjacji. Zwykle w tego typu sytuacjach (gdy pokazujemy inne obiekty dostępne przez powiązanie) będziemy wykorzystywali *ListBoxy*.

4 Implementacja i testowanie

Implementacja jest chyba najbardziej znaną fazą wytwarzania oprogramowania. Zwykle początkujący twórcy systemów komputerowych kładą na nią największy nacisk. Ma to pewne uzasadnienie, ponieważ bez niej nie mielibyśmy działającego produktu, a tylko jego papierową wersję. Niemniej, po raz kolejny powtórzmy, że prawidłowa analiza oraz właściwy projekt są podstawą do pozytywnego zakończenia implementacji.

W zdecydowanej większości przypadków implementacja dotyczy:

- zarządzania danymi, czyli zdolności do zapamiętywania, modyfikowania, usuwania oraz wyszukiwania informacji,
- logiki biznesowej (zachowania się aplikacji),
- interfejsu użytkownika, który w większości przypadków przybiera formę graficzną.

Implementacji tych elementów poświęcimy kolejne podrozdziały, ale najpierw przedyskutujemy kilka kwestii podstawowych.

4.1 Wprowadzenie

W tym podrozdziale poruszymy kilka dość fundamentalnych spraw, które warto przemyśleć zanim zajmiemy się implementacją najważniejszych elementów aplikacji.

4.1.1 Nazewnictwo i formatowanie kodu źródłowego

Jedną z podstawowych kwestii, którą trzeba rozstrzygnąć na samym początku implementacji, jest kwestia organizacji kodu źródłowego. Mamy tu na myśli takie kwestie jak:

- kryteria podziału kodu na poszczególne pliki projektu,
- nazewnictwo klas, metod, atrybutów itp.,
- sposób tworzenia dokumentacji oraz pisania komentarzy.

W zależności od konkretnego języka implementacji niektóre decyzje mogą być nam narzucone, np. język Java wymaga, aby każda publiczna klasa znajdowała się w oddzielnym pliku tekstowym. Z początku może wydawać się to kłopotliwe, ale po jakimś czasie docenimy porządek, jaki dzięki temu jest automatycznie wprowadzany. Nawet jeżeli niektóre języki programowania nie wymuszają takiego postępowania, to i tak warto umieszczać główne klasy w oddzielnych plikach.

Zwykle współczesne języki programowania sugerują pewne konwencje dotyczące nazewnictwa. Jedną z pierwszych, która zdobyła popularność, była tzw. notacja węgierska[27] opracowana przez Microsoft dla języka C oraz C++. Ogólnie rzecz biorąc, polegała na wprowadzeniu do nazwy elementu pewnych informacji dotyczących między innymi jego typu. W efekcie zmienna przechowująca tzw. „długi" wskaźnik do stałego stringu mogłaby się nazywać: `lpcszNazwisko`.

Kolejną popularną notacją jest tzw. notacja camelowa (wielbłądowa) oryginalnie stworzona na potrzeby Wiki, a spopularyzowana przez język Java. Polega na tym, że nazwy tworzymy przez usunięcie spacji, a pierwsze litery złączanych wyrazów piszemy wielką litera, np. `nazwiskoPracownikaFirmy`. Skąd się wzięła nazwa? Po prostu wielkie litery wewnątrz napisu mogą przypominać garby wielbłąda. W przypadku języków programowania (np. Java) dochodzą do tego jeszcze wytyczne dotyczące nazw metod (powinny się rozpoczynać od małej litery) czy nazw klas – te powinny być pisane dużą literą. Oprócz kwestii nazewniczych obowiązują jeszcze zalecenia dotyczące m.in. Umieszczania nawiasów określających bloki programu czy wcięć sygnalizujących np. instrukcje warunkowe.

Niestety, początkujący programiści mają tendencje do ignorowania tego typu standardów, co często prowadzi do nieczytelnego kodu. Natomiast większość firm programistycznych bardzo dokładnie przestrzega tego typu wytycznych, więc warto się tego nauczyć już na samym początku.

I jeszcze kilka uwag dotyczących nazewnictwa klas, atrybutów i metod:

- Nie bójmy się długich, treściwych nazw. Co prawda więcej czasu zajmie ich napisanie, ale czytając, będziemy dokładnie wiedzieli, o co chodzi. Pamiętajmy, że kod pisze się raz, a czyta wiele razy, więc ten wysiłek na pewno się opłaci. Niektórzy twierdzą, że takie metody zajmują więcej miejsca w skompilowanym kodzie, czy wolniej się uru-

[27] Jej nazwa wzięła się od narodowości jednego z programistów Microsoftu (Charles Simonyi), który ją opracował.

chamiają. Faktycznie, współczesne języki programowania zwykle zapisują nazwę metody w kodzie wykonywalnym, ale dla współczesnych komputerów naprawdę nie stanowi to żadnego obciążenia.

- Zwróćmy uwagę na nazewnictwo klas: rzeczownik w liczbie pojedynczej; podobnie z atrybutami. Liczba mnoga jest dozwolona tylko wtedy, gdy pojedynczy element (np. wystąpienie klasy) przechowuje zwielokrotnioną informację, np. klasa Pracownicy i jej jeden obiekt pamięta dane wielu pracowników.

- Zwykle nazwy metod są poleceniami, np. dodajKlienta(), a nie dodanieKlienta czy dodawanieKlienta. Do tego dochodzi jeszcze często stosowana konwencja get/set, czyli przedrostek informujący o rodzaju akcji. Zwróćmy uwagę na dosłowność, np. jeżeli metoda nie zwraca wartości, a tylko ją oblicza i gdzieś uaktualnia, to nie zaczynajmy jej nazwy od get.

- Często można spotkać się ze stosowaniem angielskiego nazewnictwa klas, atrybutów, metod itp. Co prawda na potrzeby tej książki nie zdecydowałem się na takie podejście (nie byłem pewien, na ile Czytelnicy życzą sobie takiego rozwiązania), ale myślę, że jest to niezły pomysł:

    - język angielski można uznać za międzynarodowy język informatyki;

    - unikamy problemów związanych z odmianą wyrazów, płcią itp.;

    - nasz kod może być czytany przez ludzi z całego świata, co w dobie Internetu i częstej współpracy międzynarodowej jest jak najbardziej wskazane;

    - no i doskonalimy naszą znajomość języka angielskiego, co na pewno się przyda.

Jak wspomnieliśmy, współczesne języki programowania[28] sugerują pewien styl. Wydaje się, że niezależnie od własnych przyzwyczajeń warto się

[28] Wytyczne dla języka Java są dostępne w [Java], a dla MS C# w [NFGR].

go trzymać. Dzięki temu kody źródłowe dla języka Java czy C# mają zunifikowany wygląd, co ułatwia ich czytanie i zrozumienie.

Oddzielnym zagadnieniem jest dokumentowanie kodu. I tu warto od razu dobrze zapamiętać, że tworzenie komentarzy w kodzie jest niezbędne i powinno być tak oczywiste jak pisanie samego kodu. Można tu wyróżnić dwie kategorie komentarzy (i obydwie są jednakowo ważne):

- opisywanie całych metod czy klas. W przypadku języka Java mamy do tego celu specjalny mechanizm zwany JavaDoc, a dla MS .NET analogiczne rozwiązanie wykorzystujące XML. Warto je poznać i konsekwentnie stosować. Szczególnie, że istnieją narzędzia, które potrafią m.in. na ich podstawie wygenerować dobrą dokumentację API,

- komentowanie wybranych fragmentów kodu (głównie wewnątrz metod). Dzięki temu nawet po dłuższej przerwie będziemy w stanie zrozumieć, co się dzieje wewnątrz metody i dlaczego tak, a nie inaczej wygląda jej kod.

4.1.2 Zintegrowane środowisko programistyczne (IDE)

Współczesne zintegrowane środowisko programistyczne (IDE - *Integrated Development Environment*) to prawdziwy kombajn, którego podstawowym elementem jest edytor kodu – w większości przypadków tekstowy. Programista otrzymuje pomoc m.in. w postaci:

- łatwego kompilowania oraz linkowania aplikacji,

- kolorowania składni,

- podpowiadania kodu programu, a nawet automatycznego jego uzupełniania w zależności od kontekstu,

- systemu pomocy zintegrowanego z Internetem,

- łatwego debugowania programu, włączając w to podglądanie stanu obiektów, zmiennych itp.,

- refactoringu, czyli automatycznego dokonywania modyfikacji w programie. Możliwości jest bardzo dużo, poczynając od zmian nazw

zmiennych (i odwołań do nich), a kończąc na przebudowie klas czy wyodrębnianiu interfejsu. Ta dziedzina jest jedną z bardziej dynamicznie rozwijających się funkcjonalności IDE,

- wygodnego zarządzania plikami projektu,
- ułatwień dotyczących pracy grupowej (jednoczesna, bezkonfliktowa praca nad jednym projektem przez wielu programistów),
- różnych wtyczek umożliwiających realizację dodatkowych funkcji. Najpopularniejsze środowiska mają biblioteki zawierające po kilkaset dodatków.

Jak widać, trudno sobie wyobrazić współczesnego programistę, który nie korzysta z IDE. Jasne, że można pisać programy w prostym edytorze i kompilować je z linii poleceń, ale po co, skoro można pracować dużo wygodniej?

Najpopularniejsze IDE dla platformy .NET:

- Microsoft Visual Studio (aktualna wersja to 2010) razem z darmową edycją Express – [MSVS],
- MonoDevelop (bezpłatne, open source) – [MoDe]
- SharpDevelop (bezpłatne, open source) http://www.icsharpcode.net/OpenSource/SD/ - [ShDe],

Środowisko Java też może pochwalić się bardzo dobrymi IDE:

- Eclipse (bezpłatne, open source) – bardzo duży kombajn, który potrafi też pracować z C++, PHP i wieloma innymi językami - [Ecli],
- NetBeans (bezpłatne, open source) – bardzo interesująca, nowa wersja (6) posiadająca m.in. zintegrowaną obsługę diagramów UML - [NeBe],
- IntelliJ IDEA (płatne) – [InJi].

Warto jeszcze wymienić rozwiązania dla Linuxa, dedykowane głównie dla C/C++: KDevelop [Kdev] oraz Anjuta [Anju].

Wydaje się, że te najpopularniejsze środowiska reprezentują podobny poziom i często kwestia wyboru zależy od osobistych przyzwyczajeń.

4.1.3 Wykorzystanie narzędzi CASE

Narzędzia CASE (*Computer-Aided Software Engineering*) stanowią bardzo interesującą kategorię aplikacji. Początkowo ich funkcjonalność ograniczała się do rysowania różnego rodzaju diagramów. Aktualnie są to bardzo rozbudowane środowiska, które umożliwiają:

- oczywiście rysowanie diagramów – przeważnie w notacji UML, ale nie tylko,
- tworzenie dokumentacji projektowej,
- generowanie kodu źródłowego na podstawie diagramów. W najprostszym przypadku jest to szkielet klas wygenerowany na podstawie diagramu klas, ale można spotkać też rozwiązania wykorzystujące diagramy sekwencji.
- generowanie diagramów na podstawie kodu źródłowego. Zastosowania analogiczne do powyższych.

Połączenie dwóch powyższych cech nosi nazwę inżynierii wahadłowej (*round-trip engeeinering*) i oznacza zdolność do pracy z kodem oraz diagramem. Innymi słowy: zmiany w kodzie źródłowym znajdują swoje odzwierciedlenie w diagramie i odwrotnie.

Jak łatwo zauważyć, ta ostatnia cecha łączy się z głównym przeznaczeniem IDE (praca z kodem źródłowym) – i tak jest w istocie. Można znaleźć rozwiązania, które dodają cechy systemów CASE do aplikacji typu IDE oraz bardzo rozbudowane narzędzia CASE, które mogą być śmialo wykorzystywane zamiast IDE.

Oferta rynkowa tego typu aplikacji jest bardzo duża. Poczynając od prostych edytorów graficznych, kończąc na wielofunkcyjnych kombajnach zastępujących IDE. Podobnie jest z rozpiętością cen: od rozwiązań darmowych po bardzo drogie produkty renomowanych firm.

Mimo wielu prób nie udało mi się znaleźć jednego narzędzia CASE, które by satysfakcjonowało mnie pod każdym względem. Jeżeli rysowanie diagramów było dobrze rozwiązane, to występowały braki w funkcjonalności związanej z pracą z kodem. Jeżeli te elementy były dobrze rozwiązane, to brakowało takich prozaicznych rzeczy jak np. eksport diagramów w for-

macie wektorowym (co jest dość ważne przy pisaniu książki). Jednym z lepszych systemów, niestety płatnych, jest Visual Paradigm [ViPa]. Posiada on wersję darmową, ale działającą z pewnymi ograniczeniami.

4.1.4 Użyteczne biblioteki pomocnicze

W większości aplikacji, które będziemy tworzyć, oprócz zasadniczych funkcjonalności, nazwijmy je merytorycznymi, występuje konieczność obsługi pewnych działań o charakterze pomocniczym. Mogą do nich należeć zarządzanie ustawieniami, plikami czy dziennikiem zdarzeń. W tego typu sytuacjach zawsze warto najpierw poszukać gotowej biblioteki realizującej określony cel. Dzięki temu zaoszczędzimy sporo czasu, który musielibyśmy zużyć na zaprojektowanie, zaimplementowanie oraz testowanie własnego rozwiązania.

Jedną z funkcjonalności, które warto umieścić w każdej komercyjnej aplikacji, jest tworzenie specjalnego pliku zawierającego informacje o akcjach wykonywanych przez system. Dzięki temu, w przypadku problemów jesteśmy w stanie dość łatwo stwierdzić, w którym miejscu coś poszło nie tak. Nazywa się to logowaniem (*logging*) lub śledzeniem (*tracing*). Najprostszym sposobem realizacji jest wykorzystywanie metody `System.out.println` (w Javie). I tak się dzieje w wielu aplikacjach. Niemniej, warto skorzystać z dedykowanej biblioteki, dzięki czemu przy tym samym nakładzie pracy uzyskamy dużo lepsze efekty. Jedną z najpopularniejszych bibliotek dla języka Java, realizujących tę funkcjonalność, była Log4j rozwijana w ramach projektu Apache Software Foundation. Od pewnego czasu coraz częściej wykorzystywana jest natywna biblioteka Javy z pakietu `java.util.logging`. Co prawda ta pierwsza ma więcej możliwości, ale ta druga jest częścią dystrybucji, a więc nie wymaga dodatkowych plików. Na poziomie podstawowym oferują podobną funkcjonalność umożliwiającą:

- określenie odbiorcy lub odbiorców komunikatów, np. konsola, plik tekstowy lub sformatowany, sieć (*socket*).
- zdefiniowanie „ważności" komunikatu. W zależności od jego poziomu oraz ustawień odbiorcy niektóre z nich są ignorowane.
- zewnętrzne skonfigurowanie aktualnych parametrów, np. za pomocą pliku lub linii poleceń.

```
try {
        FileHandler fh = new FileHandler(plikLog);
        fh.setFormatter(new SimpleFormatter());
        // fh.setFormatter(new XMLFormatter());
        logger.addHandler(fh);
        logger.setLevel(Level.ALL);
} catch (Exception e) {
        // Blad w czasie twporzenia loggera
        e.printStackTrace();
}
```

4-85 Kod inicjalizujący logger

Listing 4-85 zawiera kod inicjalizujący logger (w natywnej wersji Javy). Jest on wykonywany tylko raz, na samym początku działania aplikacji. Wykorzystanie mechanizmu jest bardzo proste i sprowadza się do wywoływania kilku metod. Przykładowe zapisanie informacji może wyglądać tak jak na listingu 4-86. Tego typu wołania (o różnym poziomie ważności) należy umieszczać w kluczowych miejscach aplikacji. Możliwe są bardziej wyrafinowane działania niż tylko zapisywanie prostych komunikatów. Ich szczegółowy opis można znaleźć w odpowiedniej dokumentacji API.

```
logger.log(Level.INFO, "Informacja do zapisu");
```

4-86 Wykorzystanie mechanizmu logowania do zapisu informacji

4.2 Zarządzanie danymi

Ten rozdział będzie paradoksalnie dość krótki. Dlaczego? Przecież zarządzanie danymi jest bardzo ważną częścią każdej aplikacji! To fakt, ale pamiętajmy, że w poprzednim rozdziale po to się tak namęczyliśmy, tworząc dość rozbudowaną bibliotekę `ObjectPlus`, aby właśnie teraz z tego skorzystać.

Podstawą do naszych prac implementacyjnych będzie ostateczna wersja diagramu klas z fazy projektowania. Ogólnie rzecz biorąc, należy dla każdej klasy na diagramie stworzyć oddzielną klasę publiczną języka Java. Oczywiście dziedziczącą z `ObjectPlus4`. Dla potrzeb testowania aplikacji warto przesłonić metodę `toString()` tak, aby otrzymywać bardziej wartościowe komunikaty.

```
public class KlientOsoba extends Klient implements IOsoba {

        private String imie;
        private String nazwisko;
```

```
	private Date dataUrodzenia = new Date();

	public KlientOsoba() {
		super();
	}

	public String getImie() {
		return imie;
	}

	public String getNazwisko() {
		return nazwisko;
	}

	public String getEtykieta() {
		return imie + " " + nazwisko;
	}

	public int getWiek() {
		Calendar dataAktualna = Calendar.getInstance();
		Calendar dataUrodzeniaKal = Calendar.getInstance();
		dataUrodzeniaKal.setTime(dataUrodzenia);

		return dataAktualna.get(Calendar.YEAR) -
dataUrodzeniaKal.get(Calendar.YEAR);
	}

	public String getDataUrodzenia() {
		return dateFormatter.format(dataUrodzenia);
	}

	public void setDataUrodzenia(String data) throws ParseException
{
		dataUrodzenia = dateFormatter.parse(data);
	}

	public void setImie(String imie) {
		this.imie = imie;
	}

	public void setNazwisko(String nazwisko) {
		this.nazwisko = nazwisko;
	}

	@Override
	public String toString() {
		return getEtykieta();
	}

	final static String formatDaty = "yyyy-MM-dd";
	private final static SimpleDateFormat dateFormatter = new
SimpleDateFormat(formatDaty);
}
```

4-87 Kod źródłowy klasy KlientOsoba

Listing 4-87 zawiera kompletny kod źródłowy klasy KlientOsoba (bez komentarzy). Zwróćmy uwagę na kilka spraw:

- Dziedziczymy z klasy `Klient`, która dziedziczy z klasy `ObjectPlus4`. Dzięki temu mamy dostęp do funkcjonalności zarządzania danymi.

- Implementujemy interfejs `IOsoba`, który umożliwa nam obejście problemu braku wielodziedziczenia w języku Java.

- Zgodnie z wymaganiami naszej biblioteki wywołujemy konstruktor z nadklasy.

- Przesłoniliśmy metodę `toString()`.

- Wprowadziliśmy stałą definiującą format daty. We wszystkich miejscach programu, które wymagają określonego formatu, posługujemy się właśnie tym elementem, a nie oddzielnym stringiem. Dzięki temu, gdy np. klient stwierdzi, że chciałby inny format daty, zrobimy to bez żadnych problemów. W ten sposób należy postepować ze wszelkiego rodzaju stałymi biznesowymi, np. ograniczenie na liczbę maksymalnie wypożyczonych nośników.

Czy to znaczy, że już wszystko jest jasne? a co z asocjacjami? Przyjęte rozwiązanie sprawia, że nie widać ich bezpośrednio w kodzie klasy. Cała funkcjonalność do ich obsługi jest umieszczona w którejś wersji `ObjectPlus`, więc nic nie musimy implementować w klasach biznesowych. Zajmiemy się nimi dopiero na etapie wypełniania danymi naszego systemu, a dokładniej tworzenia powiązań pomiędzy obiektami. Dla celów testowych warto stworzyć kilka obiektów razem z powiązaniami (listing 4-88). Naturalnie „prawdziwe" dane zostaną wprowadzone do systemu za pomocą graficznego interfejsu użytkownika (GUI) lub zaimportowane z jakiegoś źródła danych (np. repozytorium).

```
Firma f1 = new Firma("1234", "Alfa");
Firma f2 = new Firma("2", "Beta");
```

```
Adres adres1 = new Adres("miasto 1", "ulica 1", 1, 1);
Adres adres2 = new Adres("miasto 2", "ulica 2", 2, 2);

f1.dodajPowiazanie(RolePowiazan.adres, RolePowiazan.adresat, adres1);
f2.dodajPowiazanie(RolePowiazan.adres, RolePowiazan.adresat, adres2);

KlientOsoba ko1 = new KlientOsoba();
KlientOsoba ko2 = new KlientOsoba();

RodzajNosnika rodzajDVD = new RodzajNosnika("Płyta DVD");
Nosnik nosnik1 = new Nosnik(49.0, rodzajDVD);
Nosnik nosnik2 = new Nosnik(45.0, rodzajDVD);
Nosnik nosnik3 = new Nosnik(45.0, rodzajDVD);
Nosnik nosnik4 = new Nosnik(39.0, rodzajDVD);

KategoriaFilmu kategoriaF = new KategoriaFilmu("Fantastyka");
KategoriaFilmu kategoriaS = new KategoriaFilmu("Sensacja");

Film film1 = new FilmDlaWszystkich(90, new Date(), "Film 1",
kategoriaF);
Film film2 = new FilmDlaWszystkich(110, new Date(), "Film 2",
kategoriaS);

film1.dodajPowiazanie(RolePowiazan.nosnik, RolePowiazan.film,
nosnik1);
film1.dodajPowiazanie(RolePowiazan.nosnik, RolePowiazan.film,
nosnik2);
film1.dodajPowiazanie(RolePowiazan.nosnik, RolePowiazan.film,
nosnik3);
film2.dodajPowiazanie(RolePowiazan.nosnik, RolePowiazan.film,
nosnik4);

Aktor aktor1 = new Aktor("Jan", "Kowalski");
Aktor aktor2 = new Aktor("Anna", "Nowak");
Aktor aktor3 = new Aktor("Dorota", "Abacka");

film1.dodajPowiazanie(RolePowiazan.aktor, RolePowiazan.film, aktor1);
film1.dodajPowiazanie(RolePowiazan.aktor, RolePowiazan.film, aktor2);
film1.dodajPowiazanie(RolePowiazan.aktor, RolePowiazan.film, aktor3);
film2.dodajPowiazanie(RolePowiazan.aktor, RolePowiazan.film, aktor2);
film2.dodajPowiazanie(RolePowiazan.aktor, RolePowiazan.film, aktor3);
```

4-88 Generowanie danych testowych

Przy tej okazji wprowadźmy pewne usprawnienie dotyczące tworzenia powiązań. Zgodnie z projektem, jednym z parametrów przy tworzeniu powiązania są nazwy ról podane w postaci stringów. Nie zawsze będzie to najlepsze rozwiązanie. Dlaczego? Ze względu na podatność na błędy, np. raz napiszemy „klient”, a innym razem „klien”. System nie ma możliwości wy-

chwycenia takiego błędu[29]. Pewnym półśrodkiem będzie stworzenie stałych tekstowych, przechowujących odpowiednie nazwy ról i stosowanie ich zamiast zwykłych stringów (klasa `RolePowiazan` z listingu 4-88). Lepszym rozwiązaniem[30] byłoby zastąpienie nazw ról w postaci stringów wartościami wyliczeniowymi, ale to wymagałoby pewnych zmian w klasie `ObjectPlus`.

I jeszcze jeden kod klasy. Tym razem jest to Film – listing 4-89 (pominięto komentarze oraz typowe metody zwracające oraz modyfikujące atrybuty).

```
public abstract class Film extends ObjectPlus5 {

        private String tytul;
        private Date dataProdukcji;

        private int czasTrwania;

        final static String formatDaty = "yyyy-MM-dd";
        private final static SimpleDateFormat dateFormatter = new
SimpleDateFormat(formatDaty);

        public static double OplataZaWypozyczenie = 8.0f;

        public Film(int czasTrwania, Date dataProdukcji, String
tytul, KategoriaFilmu kategoria) {
                super();

                this.czasTrwania = czasTrwania;
                this.dataProdukcji = dataProdukcji;
                this.tytul = tytul;

                // Dodaj powiazanie o kategorii
                dodajPowiazanie(RolePowiazan.kategoria,
RolePowiazan.film, kategoria);
        }

        public ObjectPlusPlus[] getKategorie() {
                if(this.czySaPowiazania(RolePowiazan.kategoria)) {
                        try {
                                return
this.dajPowiazania(RolePowiazan.kategoria);
                        } catch (Exception ignore) {}
                }
```

[29] Powodem jest to, że nasze rozwiązanie (ObjectPlus) nie przechowuje meta danych (asocjacje), a tylko dane (powiązania). Oczywiście można to zmodyfikować, ale wprowadziłoby to dodatkowe zamieszanie. Z tego powodu nie zdecydowaliśmy się na to. Może ktoś z Czytelników zaproponuje własne podejście?

[30] Dlaczego lepszym? Dlatego, że zamiast stringu-stałej ze specjalnej klasy możemy wstawić zwykły string i kompilator nie zaprotestuje. Z typem wyliczeniowym taka sztuczka/pomyłka już by się nie udała.

```
                return null;
        }

        public ObjectPlusPlus[] getAktorzy() {
                if(this.czySaPowiazania(RolePowiazan.aktor)) {
                        try {
                                return
this.dajPowiazania(RolePowiazan.aktor);
                        } catch (Exception ignore) {}
                }

                return null;
        }

        public void setDataProdukcji(String dataProdukcji) throws
ParseException {
                this.dataProdukcji =
dateFormatter.parse(dataProdukcji);
        }

        @Override
        public String toString() {
                Calendar dataProdukcjiKal = Calendar.getInstance();
                dataProdukcjiKal.setTime(dataProdukcji);

                return getTytul() + " (" +
dataProdukcjiKal.get(Calendar.YEAR) + ")";
        }

   // [...]
}
```

4-89 Fragment kodu klasy Film

Kilka ciekawszych fragmentów:

- Ponownie zdefiniowaliśmy pewne stałe.
- W konstruktorze wołamy konstruktor z nadklasy, zapamiętujemy podane atrybuty oraz dodajemy informację o kategorii filmu (w postaci powiązania).
- Metody zwracające kategorie, do których należy film oraz aktorów w nim grających korzystają z funkcjonalności zaszytej w `ObjectPlus`.
- W metodzie `toString()`, ze względu na specyficzną (i zresztą dość niewygodną) obsługę dat w Javie trochę musimy się nagimnastykować, aby wyświetlić rok produkcji.

Analogicznie implementujemy wszystkie klasy z projektowego diagramu klas.

4.3 Logika biznesowa

Logika biznesowa określa, w jaki sposób mają być wykonywane poszczególne czynności obsługiwane przez system, np. dodanie nowego klienta czy wypożyczenie nośnika. Oczywiście umieszczamy ją w odpowiednich metodach znajdujących się w klasach biznesowych. W zależności od przyjętego modelu reakcji na zdarzenia, lokalizacja poszczególnych metod może się trochę różnić. Mamy tu na myśli kwestię, czy np. dodanie klienta powinno być w klasie klient – pewnie tak i to jako metoda klasowa. Możemy też wykorzystać inne podejście: główne metody, takie jak dodanie klienta, zostaną umieszczone w lokalizacji wynikającej z wykorzystanego rodzaju GUI. Natomiast to, co jest istotne i często niedoceniane, to oddzielenie „merytorycznej" części od kodu obsługującego zdarzenie. Nawet jeżeli wspomnianą metodę umieścimy w klasie obsługującej okno GUI, to ważne jest, aby była ona wyraźnie wydzielona. Poważnym błędem jest bezpośrednie umieszczanie takiego kodu wewnątrz procedury obsługi zdarzenia. W tym miejscu powinno być tylko odwołanie do metody, która to robi. Dzięki temu, jeżeli będziemy musieli zmienić obsługę okna, to nie musimy modyfikować biznesowego aspektu takiej implementacji. Taka sama sytuacja zachodzi, gdy chcemy ten sam przypadek użycia wywoływać nie tylko za pomocą opcji w menu, ale np. również skrótu klawiaturowego. Czasami wykorzystuje się w tym celu koncepcję akcji, które są wspierane przez niektóre biblioteki GUI (m.in. Swing).

Jako podstawę do implementacji logiki biznesowej (zachowania) naszego systemu powinniśmy mieć:

- diagramy aktywności,
- diagramy sekwencji.

Na pewno przyda się też diagram klas. Idealnie by było, abyśmy dysponowali obydwoma rodzajami diagramów dla każdej funkcjonalności (przypadku użycia). Dzięki temu unikniemy zgadywania, co dana funkcja i jak ma robić (z biznesowego punktu widzenia).

Przypomnijmy sobie diagram aktywności opisujący dodanie nowego klienta (rysunek 2-67, strona 94) oraz odpowiadający mu diagram sekwencji (rysunek 3-123, strona 273). Na tej podstawie napisaliśmy kod przedstawio-

ny na listingu 4-90. Jest on dość prosty, ale na wszelki wypadek wyjaśnijmy dwie kwestie. Widzimy, że tworzymy obiekt nowego klienta. Ale gdzie jest jego dodawanie do systemu? Otóż, dzięki funkcjonalności ObjectPlus, każdy nowy obiekt jest automatycznie dodawany do ekstensji klasy (a właściwie klas). I to jest właśnie to nasze dodanie. I druga sprawa: po takim utworzeniu trzeba go wypełnić danymi. Umożliwia to odpowiedni formularz GUI (niepokazany na listingu).

```
JInternalFrame frame = null;
Klient k = null;

int odp = JOptionPane.showConfirmDialog(null, "Czy nowy klient jest
osobą prywatną?", "Zdecyduj", JOptionPane.YES_NO_CANCEL_OPTION);

if(odp == JOptionPane.YES_OPTION) {
        // Klient osoba
        k = new KlientOsoba();
}
else if(odp == JOptionPane.NO_OPTION) {
        // Klient firma
        k = new Firma();
}
else {
        // Anulowano dodawanie
        return;
}

// […] Kod wyświetlający odpowiednie GUI do podania parametrów
```

4-90 Kod dodający nowego klienta

I jeszcze raz wykorzystajmy uprzednio narysowane diagramy. Tym razem napiszemy kod generujący raport dzienny (zrobimy to na podstawie diagramów z rysunku 2-68, strona 95 oraz 3-124, strona 275). Odpowiedni fragment jest pokazany na listingu 4-91. Jest dokładnym odbiciem wymienionych diagramów, więc chyba nie wymaga dokładniejszego omówienia. No, może warto wspomnieć, że zajmuje się tylko obliczeniem wymaganych wartości, ale ich nie wyświetla. Dzięki temu, tworząc odpowiednie metody, programista może go wyświetlić, ale równie dobrze wydrukować czy wysłać mailem.

```
Vector wypozyczenia = ObjectPlus5.dajEkstensje(Wypozyczenie.class);
iloscNowychWypozyczen = 0;
utarg = 0;
iloscDzisiajWypozyczonych = 0;
for(Object obj : wypozyczenia) {
        Wypozyczenie wyp = (Wypozyczenie) obj;
        if(isSameDay(wyp.getData(), reportDate)) {
                // Wypozyczenie z tego samego dnia
```

```
                iloscNowychWypozyczen++;
                utarg += wyp.getKwota();

                // Pobierz inf. O liczbie wypoć. nosnikow dla tego
wypozyczenia
                Object[] zwroty = wyp.dajPowiazania(RolePowiazan.zwrot);
                if(zwroty != null) {
                        iloscDzisiajWypozyczonych += zwroty.length;
                }
        }
}

Vector zwroty = ObjectPlus5.dajEkstensje(Zwrot.class);
iloscZwrotow = 0;
for(Object obj : zwroty) {
        Zwrot zwr = (Zwrot) obj;
        if( isSameDay(zwr.getDataZwrotu(), reportDate)) {
                // Zwrot z podanego dnia
                iloscZwrotow++;
        }
}
```

4-91 Kod generujący zawartość raportu dziennego

I tradycyjne na koniec każdego z podrozdziałów, podkreślmy, że tego typu metody tworzymy dla wszystkich funkcjonalności (przypadków użycia) naszego systemu.

4.4 Implementacja Graficznego Interfejsu Użytkownika

Jak przekonaliśmy się z lektury podrozdziału 3.6 (strona 249), zaprojektowanie dobrego GUI nie jest sprawą prostą. Podobnie ma się sprawa z jego implementacją. Generalnie możemy wyróżnić dwa najpopularniejsze podejścia do tego zagadnienia: ręczne (programista samodzielnie wpisuje kod tworzący poszczególne widgety) oraz zautomatyzowane (przy pomocy dedykowanego edytora).

Ręczna implementacja GUI jest:

- czasochłonna,
- błędogenna,
- skomplikowana.

W związku z tym warto używać wizualnych edytorów, które pozwalają utworzyć GUI z gotowych komponentów i podłączyć do nich zdarzenia. Ale trzeba pamiętać, że czasami mimo zastosowania edytora może wystąpić konieczność wprowadzania ręcznych modyfikacji. Dlatego warto sprawdzić, w jaki sposób edytor traktuje wygenerowany kod. Głównie chodzi o ustale-

nie, czy ręcznie wprowadzone poprawki są odzwierciedlane w projekcie wizualnym. Jeżeli nie i cały kod jest od nowa regenerowany na podstawie projektu graficznego, to zwykle taki edytor nie będzie dla nas zbyt użyteczny. Należy również sprawdzić, czy generowany kod musi mieć jakąś szczególną strukturę lub też nazewnictwo. Lepiej, aby nie było tego typu wymagań.

Innym, jak na razie dość rzadko stosowanym sposobem tworzenia GUI, jest podejście deklaratywne. Programista koncentruje się na tym, co ma być zrobione, a nie jak to osiągnąć. Podobnie jak w przypadku języka SQL, pozostajemy na wyższym poziomie abstrakcji, nie zagłębiając się w szczegóły implementacyjne. Całą „czarną robotę" wykonuje specjalizowana biblioteka, która na podstawie budowy danych oraz pewnych dodatkowych informacji sama generuje odpowiednie okna z widgetami. Ze względu na skomplikowanie tego tematu, jest to zagadnienie na oddzielną książkę, która być może kiedyś powstanie.

Większość współczesnych języków programowania bazuje na bibliotekach zawierających gotowe komponenty sterowane zdarzeniami:

- dla języka Java:
    - AWT – aktualnie mało używana,
    - Swing – najbardziej popularna biblioteka GUI dla Javy,
    - SWT – wykorzystywana głównie na platformie Eclipse.
- MS .NET (C#, C++, VB):
    - Windows Forms,
    - Windows Presentation Foundation (WPF).

Oprócz tego istnieje wiele bibliotek stworzonych przez niezależne firmy i/lub społeczności (to dotyczy głównie projektów *open-source*). Do najbardziej znanych, aktywnie rozwijanych i do tego działających na najpopularniejszych platformach (Windows, Linux, czasami też Mac) należą:

- FLTK (Fast Light Toolkit), http://www.fltk.org/
- GTK+, http://www.gtk.org/

- Lgi, http://www.memecode.com/lgi.php
- Qt, http://trolltech.com/
- wxWidgets, http://www.wxwidgets.org/.

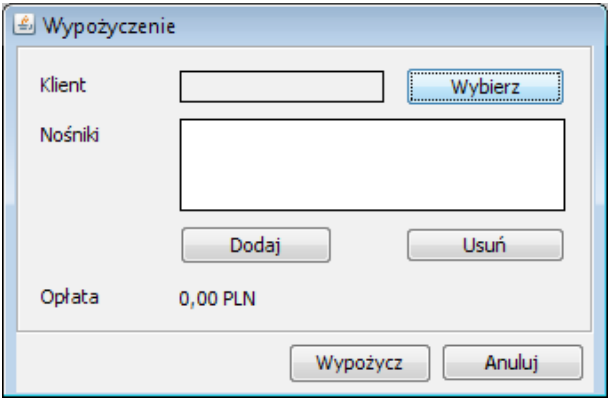
Zestaw komponentów służących do budowania interfejsu jest dość podobny we wszystkich językach oraz bibliotekach. Różne są managery rozkładu, czyli specjalne komponenty odpowiedzialne za rozmieszczenie poszczególnych widgetów w oknie. Ma to duże znaczenie gdy okno może zmieniać swoje rozmiary. W takiej sytuacji położenie oraz wielkość poszczególnych jego elementów powinna być odpowiednio dopasowana. Wydaje się, że jednym z najlepszych managerów rozkładu jest „kotwicowy" wykorzystywany w MS .NET oraz ostatnio również w Java.

Niestety, dogłębne zrozumienie zasad projektowania GUI oraz jego implementacji jest dość pracochłonne i wymaga sporo wiedzy teoretycznej oraz jeszcze więcej ćwiczeń.

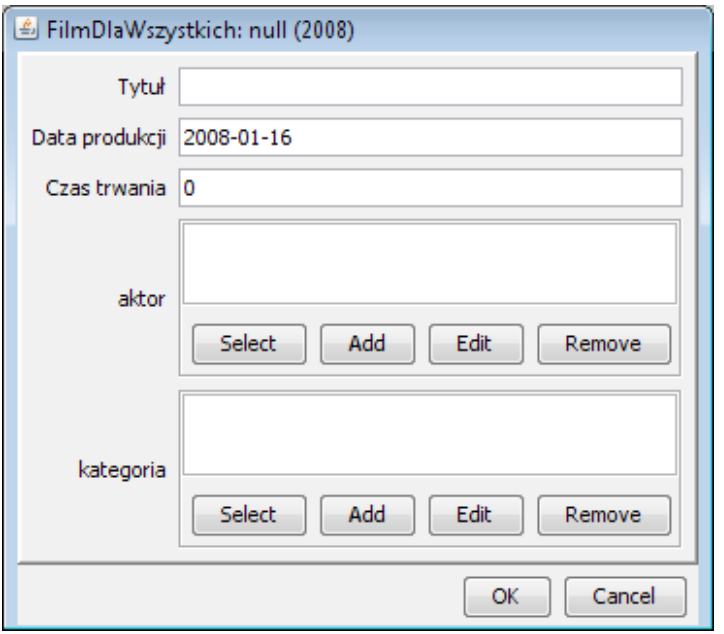
Nie będziemy tu zamieszczać konkretnego opisu implementacji GUI dla wypożyczalni wideo. Jest to spowodowane tym, że każdy może wybrać własny sposób (ręcznie lub przy użyciu któregoś z edytorów), a przykładowy projekt odpowiednich okienek znajduje się w podrozdziale 3.7.3 na stronie 276. Oprócz tego, niezależnie od wybranej metody, dokładny opis zająłby nam sposoro miejsca. Tym bardziej że w literaturze można znaleźć dużo pozycji traktujących o tworzeniu GUI, np. [Ecke06], [Guoj05], [Krau07], [Sell06], [Smar05], [Thel07], [Walr04].

Spójrzmy tylko na przykładowe realizacje ekranów na podstawie projektów z podrozdziału 3.7.3 (strona 276).

Rysunek 4-127 zawiera przykładowe okno, wykonane w technologii Swing na podstawie projektu z rysunku 3-125 (strona 277), a rysunek 4-128 analogicznie dla projektu z rysunku 3-126 (strona 278).



4-127 Przykładowe okno powstałe na podstawie projektu z rysunku 3-125



4-128 Okno umożliwiające dodawanie informacji o filmie

4.5 Testowanie

W tym rozdziale zasygnalizujemy tylko najważniejsze sprawy związane z testowaniem. Więcej informacji można znaleźć np. w [Dąbr05].

Każdy program komputerowy powinien być przetestowany. Jest takie popularne powiedzenie: „Nie ma programów bezbłędnych, a są tylko źle przetestowane". Jest w tym dużo prawdy i przekonał się o tym każdy, kto napisał choć raz trochę większą aplikację. Oczywiście jakość i intensywność przeprowadzanych testów zależy od wielu czynników takich jak:

- przeznaczenie i odbiorca programu,
- budżet,
- czas.

Oczywiste jest, że innej jakości może domagać się klient, który zapłacił za aplikację (czasami dość duże pieniądze), a czego innego może oczekiwać użytkownik darmowego oprogramowania (chociaż to czasami ma lepszą jakość niż komercyjne).

Testowanie jest chyba najmniej przyjemnym etapem wytwarzania oprogramowania. Jest to spowodowane m.in. tym, że:

- wielokrotnie musimy wykonywać podobne czynności, co jest dość nużące,
- często nie jesteśmy w stanie odtworzyć błędu,
- czasami jest trudno znaleźć jego przyczynę.

Kluczem do szybkiego ustalenia przyczyny błędu jest możliwość doprowadzenia do jego wystąpienia. Jeżeli po wykonaniu określonych akcji jesteśmy w stanie za każdym razem sprawić, że błąd wystąpi, to w 99% przypadków możemy go stosunkowo szybko poprawić. Czasami nie jest to takie łatwe, np. gdy występuje u klienta, a u nas wszystko działa prawidłowo (w takiej sytuacji przydają się tzw. logi, którymi zajmowaliśmy się w podrozdziale 4.1.4 na stronie 286).

Jedna z ważniejszych zasad dotyczących testowania mówi, że testy powinny być przeprowadzane przez osoby, które nie tworzyły danego fragmentu programu. Dzięki temu mają świeże podejście i mogą bardziej wszechstronnie dokonać sprawdzenia. Co więcej, niektóre rodzaje testów powinny być przeprowadzane przez zwykłych użytkowników niebędących programi-

stami. Zaobserwowano bowiem, że programiści, nawet podświadomie, mogą czynić pewne założenia dotyczące np. wprowadzania danych: tam, gdzie program oczekuje liczby, nie podadzą tekstu. Zwykły użytkownik nie zawsze przestrzega tego typu reguł co prowadzi do wychwycenia większej liczby błędów.

Klasyczne sposoby testowania można podzielić na dwie grupy:

- testy czarnej skrzynki. Testujący nie zna budowy testowanego elementu. Wie tylko, jaka powinna być odpowiedź na konkretne dane. Mogą dotyczyć GUI, np. sprawdzamy proces rejestracji nowego klienta w systemie lub kwestii programistycznych, np. metoda licząca średnią – znając dane, które podaliśmy, możemy sprawdzić czy wynik jest prawidłowy. W pierwszym przypadku nie muszą to być programiści; co więcej, jest zalecane, aby tak właśnie było.

- testy białej (szklanej, przezroczystej) skrzynki. Testujący ma dostęp do wnętrza sprawdzanego elementu i np. przy pomocy debuggera śledzi wykonywanie programu. Może też po prostu przeglądać kod źródłowy w edytorze.

Jednym z nowszych podejść do problemu testowania są tzw. testy jednostkowe (*unit tests*). Ogólnie rzecz biorąc, polegają na tym, że programista razem z jakąś metodą merytoryczną (np. dodanie nowego użytkownika) tworzy również metodę, która ją przetestuje. Podejście to jest wspomagane przez różne biblioteki oraz IDE. Najpopularniejsze z nich to:

- JUnit dla języka Java. Więcej informacji np. w [Kosk07].

- NUnit dla platformy MS .NET (w tym dla C#). Więcej informacji np. w [Hunt07].

Stosowanie testów jednostkowych jest dość użyteczne, ale nie może całkowicie zastąpić klasycznych testów. Powodów jest kilka. Jednym z nich jest ich przeznaczenie: nie są dedykowane do wykonywania testów integracyjnych. Drugą przyczyną jest fakt, iż są tylko tak dobre jak metody testujące stworzone przez programistę. A jak wiadomo, testy powinien przeprowadzać ktoś z zewnątrz, a nie twórca testowanego kodu.

Niezastąpionym narzędziem służącym do poszukiwania błędów jest debugger. Występuje w każdym porządnym IDE (a nawet jest dystrybuowany

z SDK[31], ale wtedy jego używanie jest mniej wygodne). Jego podstawowa funkcjonalność pozwala na:

- zatrzymanie wykonywania programu i podejrzenie aktualnego stanu systemu,
- wstawianie punktów kontrolnych. W momencie dojścia sterowania do takiego miejsca wykonywanie kodu zostaje wstrzymane i możemy dokonywać różnych operacji na programie,
- ustawianie pułapek reagujących na różne zdarzenia, np. wykonywanie programu zostanie wstrzymane w momencie gdy atrybut pensja stanie się mniejszy od 0.

Bardziej zaawansowane rozwiązania (np. dla Javy 6 czy .NET) umożliwiają nawet zatrzymanie programu, dokonanie w nim pewnych zmian i kontynuację jego działania bez ponownego kompilowania (tak naprawdę kompilacja występuje, ale jest ukryta przed użytkownikiem i dotyczy tylko fragmentu kodu).

[31] SDK, czyli *Software Development Kit* jest to zbiór bibliotek i narzędzi niezbędny do tworzenia oprogramowania na danej platformie czy języku programowania. Zwykle obejmuje też tekstową wersję debuggera (tak jest dla Javy oraz .NET), więc jego używanie nie jest zbyt wygodne.

5 Uwagi końcowe

I tak oto dobrnęliśmy do końca naszych rozważań na temat modelowania, obiektowości oraz przełożenia tego wszystkiego na implementację. Mam nadzieję, Czytelniku, że czytanie tej książki było taką samą przyjemnością dla Ciebie, jak jej pisanie dla mnie.

Naszą przygodę rozpoczęliśmy od fazy analizy, gdzie zapoznaliśmy się z podstawowymi pojęciami z dziedziny obiektowości takimi jak: klasy, ich ekstensje, asocjacje, powiązania, różne rodzaje atrybutów i dziedziczenia. Omówiliśmy również przydatność przesłaniania oraz przeciążania metod. Następnie przetestowaliśmy tę wiedzę, tworząc odpowiednie diagramy (przypadków użycia, klas, aktywności, stanów) dla naszego przykładowego systemu: wypożyczalni wideo.

W kolejnym rozdziale poświęconym projektowaniu dyskutowaliśmy, jak mają się poszczególne pojęcia obiektowości oraz analizy do współczesnych języków programowania: głównie Javy oraz trochę MS C# i C++. Omówiliśmy niektóre aspekty projektu systemu informatycznego, ze szczególnym uwzględnieniem ich wpływu na fazę implementacji. Zaprojektowaliśmy, a następnie zaimplementowaliśmy omawiane metody przejścia w postaci kilku klas (ObjectPlusX). Przedyskutowaliśmy również szczegółowo modyfikację naszego przykładowego diagramu klas dla wypożyczalni wideo, tak aby dało się go zaimplementować w popularnych językach programowania (z akcentem na język Java). Uzupełniliśmy również fragment projektu logiki biznesowej naszego systemu (w postaci diagramów sekwencji). Przedstawiliśmy również wytyczne dotyczące prawidłowego zaprojektowania graficznego interfejsu użytkownika (GUI). Dodatkowo omówiliśmy wagę użyteczności GUI razem z podstawowymi informacjami dotyczącymi przeprowadzania testów użyteczności.

Ostatni rozdział został poświęcony implementacji oraz testowaniu. Wbrew temu, co sądzą początkujący twórcy systemów komputerowych, nie są to najważniejsze fazy produkcji oprogramowania (co zresztą znalazło odbicie w jego objętości). Pokazaliśmy, jak na podstawie wyników faz analizy i projektowania dokonać implementacji w języku Java. Co prawda, nie omówiliśmy całego kodu źródłowego realizującego zadania stojące przed systemem obsługującym wypożyczalnię, ale bardziej rozbudowaną wersję kodu można pobrać z mojej strony Web (*http://www.mtrzaska.com*). Gdybyśmy

chcieli to zrobić, to pewnie zajęłoby nam to całą książkę. Wspomnieliśmy co nieco również o innych istotnych zagadnieniach związanych z implementacją (nazewnictwo, IDE, narzędzia CASE) oraz o testowaniu stworzonego programu.

Na koniec mam prośbę, abyśmy nie traktowali przedstawionych tu rozwiązań jako kompletnych i do tego jedynych możliwych. Myślmy raczej o nich jako o ilustracji wspaniałych możliwości, które dają nam współczesne języki programowania. Uczyńmy z nich wstęp do dalszych eksperymentów oraz praktycznych rozwiązań. To, co stworzyliśmy pod postacią klas ObjectPlusX, jest użyteczne, ale w przypadku komercyjnego zastosowania, wymaga dalszego rozwoju. Oczywiście, każdy może w dowolny sposób wykorzystać przedstawione tu pomysły. Będzie mi bardzo miło, jeżeli zechcesz, Czytelniku, przysłać mi list, dzieląc się ze mną swoimi uwagami. Obiecuję, że postaram się odpowiedzieć na każdy mail przysłany na adres: *mtrzaska@mtrzaska.com*.

Bibliografia

[Anju] Anjuta DevStudio: http://anjuta.org/

[Bana04] Lech Banachowski , Agnieszka Chądzynska, Krzysztof Matejewski: Relacyjne bazy danych. Wykłady i ćwiczenia. Wydawnictwo PJWSTK. 2004. ISBN: 83-89244-24-1.

[Dąbr05] Włodzimierz Dąbrowski, Kazimierz Subieta: Podstawy inżynierii oprogramowania. Wydawnictwo PJWSTK 2005. ISBN 83-89244-46-2.

[Ecke06] Bruce Eckel: Thinking in Java. Edycja polska. Wydanie IV. Wydawnictwo Helion. ISBN: 83-246-0111-2. Bezpłatna wersja *on-line* (po angielsku): http://mindview.net/Books/TIJ/DownloadSites.

[Ecli] The Eclipse Project: http://www.eclipse.org/

[Fabr07] Fabrice Marguerie, Steve Eichert and Jim Wooley: LINQ in Action. Manning Publications Co. 2007. ISBN: 1-933988-16-9.

[Fowl04] Martin Fowler: UML Distilled: a Brief Guide to the Standard Object Modeling Language (3rd Edition). Addison Wesley. 2004. ISBN 0-321-19368-7.

[Gręb96] Jerzy Gręboszc: Symfonia C++, Oficyna Kallimach, Kraków 1996.

Więcej informacji można znaleźć na: http://www.ifj.edu.pl/~grebosz/symfoniap.html.

[Guoj05] J. L. Guojie: Professional Java Native Interfaces with SWT/JFace. ISBN: 978-0470094594. Wrox. 2005.

[Hibe06] Hibernate - oficjalny tutorial:

http://www.hibernate.org/hib_docs/v3/reference/en/html/tutorial.html

[Hunt07] Andrew Hunt, David Thomas: Pragmatic Unit Testing in C# with NUnit, 2nd Ed. The Pragmatic Bookshelf, Raleigh 2007, ISBN 0-9776166-7-3.

[InJi] IntelliJ IDEA: http://www.jetbrains.com/idea/

[Java] Code Conventions for the Java Programming Language:

http://java.sun.com/docs/codeconv/

[Kdev] KDevelop: http://www.kdevelop.org/

[Kosk07] Lasse Koskela: Test Driven: TDD and Acceptance TDD for Java Developers. Manning Publications 2007. ISBN: 1932394850. 2007.

[Krau07] A. Krause: Foundations of GTK+ Development. ISBN: 978-1590597934. Apress. 2007.

[Mint06] Dave Minter, Jeff Linwood: Beginning Hibernate: From Novice to Professional. Apress. 2006. ISBN: 978-1590596937.

[MoDe] MonoDevelop: http://www.monodevelop.com

[MSVS] Microsoft Visual Studio: http://msdn.microsoft.com/vstudio,

Microsoft Visual Studio Express: http://msdn.microsoft.com/express

[NeBe] The NetBeans IDE: http://www.netbeans.org

[NFGR] .NET Framework 3.5 General Reference - Design Guidelines for Class Library Developers:

http://msdn2.microsoft.com/en-us/library/ms229042.aspx

[Płod05] Jacek Płodzień, Ewa Stemposz: Analiza i projektowanie systemów informatycznych. Wydanie drugie rozszerzone. Wydawnictwo PJWSTK 2005. ISBN 83-89244-42-X, 83-89244-43-8.

[Smar05] J. Smart, K. Hock, S. Csomor: Cross-Platform GUI Programming with wxWidgets. ISBN: 978-0131473812 . Prentice Hall. 2005.

[Sell06] Ch. Sells, M. Weinhardt: Windows Forms 2.0 Programming. ISBN: 978-0-321-26796-2. Addison Wesley Professional. 2006.

[ShDe] SharpDevelop: http://www.icsharpcode.net/OpenSource/SD/

[Thel07] J. Thelin: Foundations of Qt Development. ISBN: 978-1-59059-831-3. Apress. 2007.

[Usei07] http://www.useit.com

[Uzyt07] http://www.uzytecznosc.pl

[ViPa] Visual Paradigm: http://www.visual-paradigm.com/

[Walr04] K. Walrath, M. Campione, A. Huml, S. Zakhour: The JFC Swing Tutorial (2nd Edition). ISBN 0201914670. Prentice Hall. 2004.

[Wryc05] Stanisław Wrycza, Bartosz Marcinkowski, Krzysztof Wyrzykowski: Język UML 2.0 w modelowaniu systemów informatycznych. Helion 2005. ISBN

83-7361-892-9.

Ważniejsze informacje związane z systemem dla wypożyczalni wideo

Indeks

Spis ilustracji

Spis listingów