Mariusz Trzaska

Modelowanie i implementacja systemów informatycznych

Fragmenty książki pobrane z http://www.mtrzaska.com

Kup tę książkę:

- Wersja elektroniczna:
  - edycja PDF (ADE),
  - edycja ePub (ADE).
- Wersja drukowana.

Warszawa 2008/12

Copyright by Mariusz Trzaska
Warszawa 2008

Copyright by Wydawnictwo PJWSTK
Warszawa 2008

Wszystkie nazwy produktów są zastrzeżonymi nazwami handlowymi lub znakami towarowymi odpowiednich firm.

Książki w całości lub w części nie wolno powielać ani przekazywać w żaden sposób, nawet za pomocą nośników mechanicznych i elektronicznych (np. zapis magnetyczny) bez uzyskania pisemnej zgody Wydawnictwa.

Adres autora:

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

Edytor
Leonard Bolc

Redaktor techniczny
Ada Jedlińska

Korekta
Anna Bittner

Komputerowy skład tekstu
Mariusz Trzaska

Projekt okładki
Andrzej Pilich

Wydawnictwo Polsko-Japońskiej Wyższej Szkoły Technik Komputerowych
ul. Koszykowa 86, 02-008 Warszawa
tel. 022 58 44 526, fax 022 58 44 503

Oprawa miękka
ISBN 978-83-89244-71-3
Nakład: 500 egz.

Notka biograficzna

Dr inż. Mariusz Trzaska jest adiunktem w Polsko-Japońskiej Wyższej Szkole Technik Komputerowych, gdzie zajmuje się działalnościa dydaktyczną oraz naukową. Oprócz tego bierze udział w różnych projektach naukowych oraz komercyjnych. Prowadzi także szkolenia oraz warsztaty. Jego zainteresowania 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

Spis rozdziałów

1 Wprowadzenie 7

2 Analiza 9

2.1 Wymagania klienta 10
2.2 Wymagania dla Wypożyczalni wideo 11
2.3 Przypadki użycia 14
2.4 Diagram klas 23
2.5 Diagram aktywności 82
2.6 Diagram stanów 84

3 Projektowanie 86

3.1 Klasy 87
3.2 Asocjacje 118
3.3 Dziedziczenie 157
3.4 Ograniczenia i inne konstrukcje 185
3.5 Model relacyjny 195
3.6 Użyteczność graficznych interfejsów użytkownika 224
3.7 Projekt dla Wypożyczalni wideo 234

4 Implementacja i testowanie 252

4.1 Wprowadzenie 252
4.2 Zarządzanie danymi 258
4.3 Logika biznesowa 263
4.4 Implementacja Graficznego Interfejsu Użytkownika 266
4.5 Testowanie 269

5 Uwagi końcowe ... 272

Bibliografia ... 274

Ważniejsze informacje związane z Wypożyczalnią wideo ... 276

Indeks ... 278

Spis ilustracji ... 280

Spis listingów ... 286

Spis treści

1 Wprowadzenie ... 7

2 Analiza ... 9

2.1 Wymagania klienta ... 10

2.2 Wymagania dla Wypożyczalni wideo ... 11

2.3 Przypadki użycia ... 14

2.3.1 Ogólny diagram przypadków użycia dla Wypożyczalni wideo ... 16

2.3.2 Szczegółowy diagram przypadków użycia ... 20

2.4 Diagram klas ... 23

2.4.1 Obiekt ... 24

2.4.2 Klasa ... 25

2.4.2.1 Atrybuty ... 26

2.4.2.2 Metody ... 31

2.4.3 Asocjacje ... 32

2.4.3.1 Asocjacja binarna ... 35

2.4.3.2 Asocjacja n-arna ... 36

2.4.3.3 Asocjacja kwalifikowana ... 37

2.4.3.4 Asocjacja rekurencyjna (zwrotna) ... 38

2.4.3.5 Klasa asocjacji ... 39

2.4.3.6 Agregacja i kompozycja ... 41

2.4.4 Dziedziczenie ... 43

2.4.4.1 Dziedziczenie pojedyncze ... 43

2.4.4.2 Klasa abstrakcyjna i polimorfizm metod ... 44

2.4.4.3 Dziedziczenie wielokrotne ... 48

2.4.4.4 Dziedziczenie typu overlapping ... 49

2.4.4.5 Dziedziczenie wieloaspektowe ... 50

2.4.4.6 Dziedziczenie dynamiczne ... 51

2.4.5 Ograniczenia ... 52

2.4.5.1 Ograniczenie {subset} ... 52

2.4.5.2 Ograniczenie {ordered} ... 53

2.4.5.3 Ograniczenie {bag} oraz {history} ... 53

2.4.5.4 Ograniczenie {xor} ... 54

2.4.6 Diagram klas dla Wypożyczalni Wideo ... 54

2.5 Diagram aktywności ... 82

2.6 Diagram stanów ... 84

3 Projektowanie ... 86

3.1 Klasy ... 87
3.1.1 Obiekt ... 87
3.1.2 Klasa ... 88
3.1.3 Ekstensja klasy ... 89
3.1.3.1 Implementacja ekstensji klasy w ramach tej samej klasy ... 90
3.1.3.2 Implementacja ekstensji klasy przy użyciu klasy dodatkowej ... 92
3.1.4 Atrybuty ... 93
3.1.4.1 Atrybuty proste ... 93
3.1.4.2 Atrybuty złożone ... 94
3.1.4.3 Atrybuty wymagane oraz opcjonalne ... 94
3.1.4.4 Atrybuty pojedyncze ... 95
3.1.4.5 Atrybuty powtarzalne ... 95
3.1.4.6 Atrybuty obiektu ... 95
3.1.4.7 Atrybuty klasowe ... 95
3.1.4.8 Atrybuty wyliczalne ... 96
3.1.5 Metody ... 97
3.1.5.1 Metoda obiektu ... 97
3.1.5.2 Metoda klasowa ... 97
3.1.5.3 Przeciążenie metody ... 98
3.1.5.4 Przesłonięcie metody ... 98
3.1.6 Trwałość ekstensji ... 98
3.1.6.1 Ręczna implementacja trwałości danych ... 99
3.1.6.2 Implementacja trwałości danych w oparciu o serializację ... 105
3.1.6.3 Inne sposoby uzyskiwania trwałości danych ... 109
3.1.7 Klasa ObjectPlus ... 111

3.2 Asocjacje ... 118
3.2.1 Implementacja asocjacji za pomocą identyfikatorów ... 118
3.2.2 Implementacja asocjacji za pomocą natywnych referencji ... 124
3.2.3 Implementacja różnych rodzajów asocjacji ... 129
3.2.3.1 Asocjacja skierowana ... 130
3.2.3.2 Asocjacja rekurencyjna ... 130
3.2.3.3 Asocjacja z atrybutem ... 131
3.2.3.4 Asocjacja kwalifikowana ... 132
3.2.3.5 Asocjacja n-arna ... 136
3.2.3.6 Implementacja agregacji ... 137
3.2.3.7 Implementacja kompozycji ... 137
3.2.4 Klasa ObjectPlusPlus ... 144

3.3 Dziedziczenie ... 157
3.3.1 Dziedziczenie rozłączne ... 157
3.3.2 Polimorficzne wołanie metod ... 158
3.3.3 Dziedziczenie typu *overlapping* ... 162
3.3.3.1 Obejście dziedziczenia overlapping za pomocą grupowania ... 162

3.3.3.2 Obejście dziedziczenia overlapping za pomocą agregacji lub kompozycji 164
3.3.3.3 Polimorfizm w dziedziczeniu overlapping 168
3.3.4 Dziedziczenie kompletne oraz niekompletne 169
3.3.5 Dziedziczenie wielokrotne (wielodziedziczenie) 170
3.3.6 Dziedziczenie wieloaspektowe 175
3.3.7 Dziedziczenie dynamiczne 178
3.3.8 Dziedziczenie, a ekstensja klasy 182
3.3.9 Podsumowanie 184

3.4 Ograniczenia i inne konstrukcje 185
3.4.1 Implementacja ograniczeń dotyczących atrybutów 185
3.4.2 Implementacja ograniczenia {subset} 187
3.4.3 Implementacja ograniczenia {ordered} 191
3.4.4 Implementacja ograniczenia {bag} oraz {history} 191
3.4.5 Implementacja ograniczenia {XOR} 192
3.4.6 Implementacja innych ograniczeń 194

3.5 Model relacyjny 195
3.5.1 Mapowanie klas 196
3.5.2 Mapowanie asocjacji 199
3.5.2.1 Asocjacje binarne 199
3.5.2.2 Asocjacje z atrybutem 201
3.5.2.3 Asocjacje kwalifikowane 202
3.5.2.4 Asocjacje n-arne 203
3.5.2.5 Agregacja i kompozycja 205
3.5.3 Mapowanie dziedziczenia 206
3.5.4 Relacyjne bazy danych w obiektowych językach programowania 208
3.5.4.1 Wykorzystanie JDBC 209
3.5.4.2 Wykorzystanie gotowej biblioteki do pracy z danymi (Hibernate) ... 211

3.6 Użyteczność graficznych interfejsów użytkownika 224
3.6.1 Co to jest użyteczność? 224
3.6.2 Kształtowanie użyteczności 225
3.6.3 Testowanie użyteczności 225
3.6.4 Użyteczność niestety kosztuje 227
3.6.5 Zalecenia dotyczące Graficznego Interfejsu Użytkownika 228
3.6.5.1 Wymagania dotyczące funkcjonalności 228
3.6.5.2 Wymagania związane z wykorzystywaną platformą 229
3.6.5.3 Wymagania dotyczące okien 229
3.6.5.4 Wymagania dotyczące zarządzania oknami dialogowymi 230
3.6.5.5 Wymagania dotyczące kontrolek 230
3.6.5.6 Wymagania dotyczące menu 231
3.6.5.7 Wymagania dotyczące podpisów 232
3.6.5.8 Wymagania dotyczące pracy z klawiaturą 232
3.6.6 Jakość interfejsu graficznego 232

3.7 Projekt dla Wypożyczalni wideo ... *234*
3.7.1 Diagram klas dla wypożyczalni wideo ... 234
3.7.2 Projekt działania systemu ... 245
3.7.3 Projekt interfejsu użytkownika ... 248

4 Implementacja i testowanie ... 252

4.1 Wprowadzenie ... *252*
4.1.1 Nazewnictwo i formatowanie kodu źródłowego ... 252
4.1.2 Zintegrowane środowisko programistyczne (IDE) ... 255
4.1.3 Wykorzystanie narzędzi CASE ... 256
4.1.4 Użyteczne biblioteki pomocnicze ... 257

4.2 Zarządzanie danymi ... *258*

4.3 Logika biznesowa ... *263*

4.4 Implementacja Graficznego Interfejsu Użytkownika ... *266*

4.5 Testowanie ... *269*

5 Uwagi końcowe ... 272

Bibliografia ... 274

Ważniejsze informacje związane z Wypożyczalnią wideo ... 276

Indeks ... 278

Spis ilustracji ... 280

Spis listingów ... 286

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 chciałem się zająć przełożeniem efektów analizy na projektowanie oraz programowanie. W związku z tym, czynności z tej pierwszej fazy są potraktowane nieco skróto-

---

[1] Na stronie 275 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/.

wo (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.

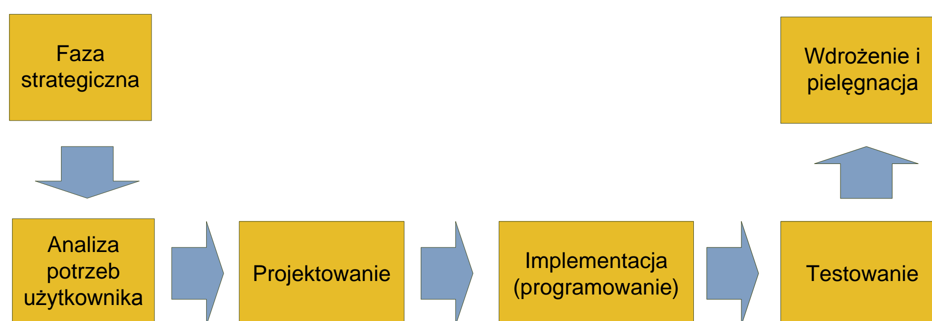
Proces analizy potrzeb użytkownika i projektowania oprogramowania oraz zagadnienia z tym związane (szeroko pojmowana obiektowość) są czasami postrzegane (szczególnie przez niektórych studentów) jako zbędny balast teoretyczny. Można spotkać się z opinią, że należy usiąść i zacząć pisać program (*programować*), 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 przeprowadzeniu 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 256). 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).

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 23 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 24) 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>();

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

            /** Wyswietla ekstensje. Metoda klasowa */
            public static void pokazEkstensje() {
(5)               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? Oczy-

wiście – 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 90), 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 bezpośredniego 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 111.

3.1.4 Atrybuty

W podrozdziale 2.4.2.1 (strona 26 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 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#

---

[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.

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 31). 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 158).

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 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:

---

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

- 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 90).

- (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 stream)
(5)           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.
- (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),

[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.

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 3-1. 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).

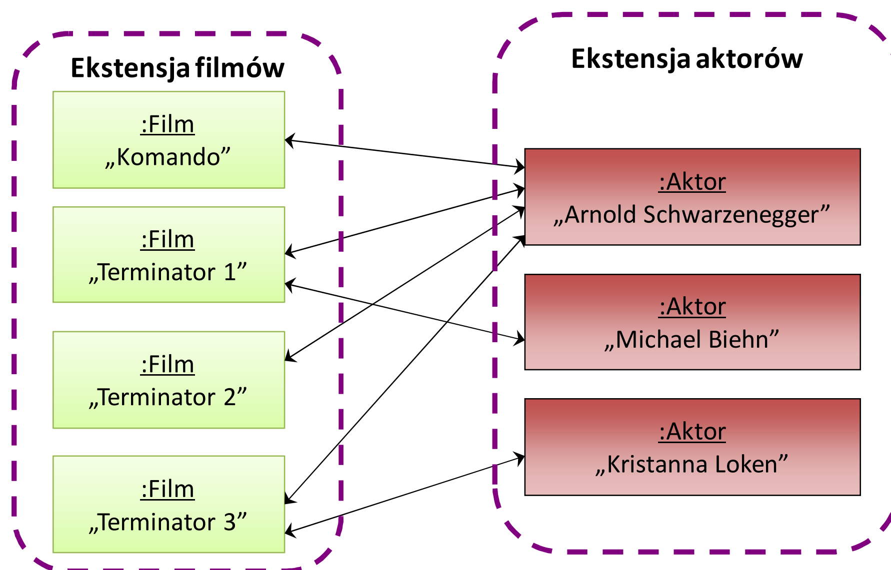
- Możemy spróbować zapisywać w ten sposób:

    - Tak jak dotychczas zapisujemy poszczególne atrybuty,

    - Gdy natrafimy na obiekt, to wywołujemy na jego rzecz metodę `write()` (z listingu 3-11, strona 101). 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 „Arnold 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 118).



3-1 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 3-1 (strona 105). 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 zaimplementowany 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 101), 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 102). 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 stream)
(1)          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 po-

[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.

święconym modelowi relacyjnemu (podrozdział 3.5, strona 195).

- 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/).

- 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 43).

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

---

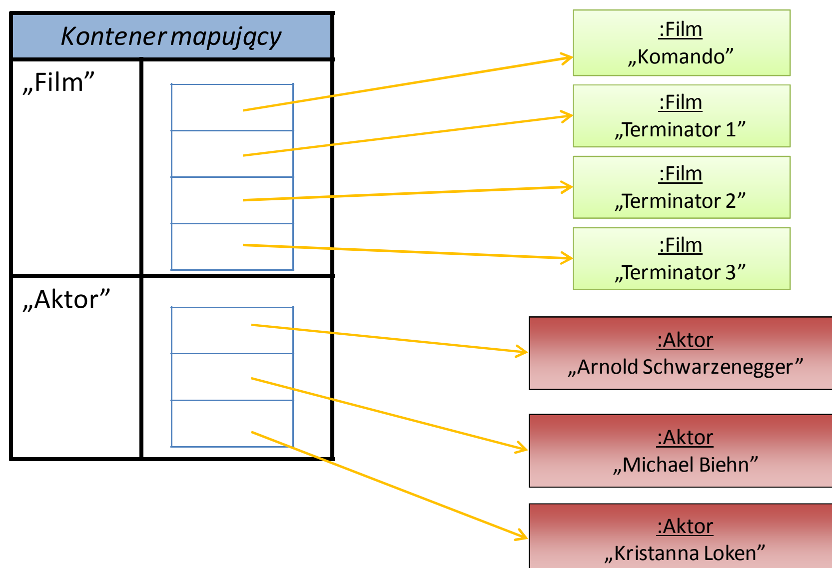
[12] Jako że wszystkie klasy w języku Java, te dostarczane przez producenta i te stworzone przez programistę dziedziczą z klasy `Object`, to nasza ulepszona klasa może nazywać się `ObjectPlus`. W przypadku MS C# jest zresztą podobnie – również istnieje tam jedna wspólna nadklasa.

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:

- 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 3-2 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.



3-2 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.

---

[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.

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

- (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 105).

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 rozwią-

zać 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`).

3.7 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.

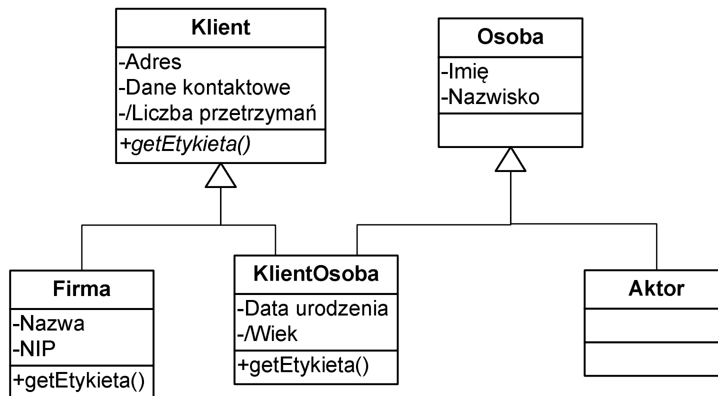
3.7.1 Diagram klas dla wypożyczalni wideo

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 111,
- `ObjectPlusPlus`: obsługa powiązań i częściowo kompozycji – podrozdział 3.2.4 na stronie 144,
- `ObjectPlus4`: wsparcie dla niektórych rodzajów ograniczeń - podrozdział 3.4.2 (i dalsze) na stronie 187.

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 82). Jest on dla nas podstawą do dalszych prac. Analogicznie jak przy okazji jego tworzenia (podrozdział 2.4.6, strona 54), 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.



3-43 Tworzenie projektowego diagramu klas – krok 1

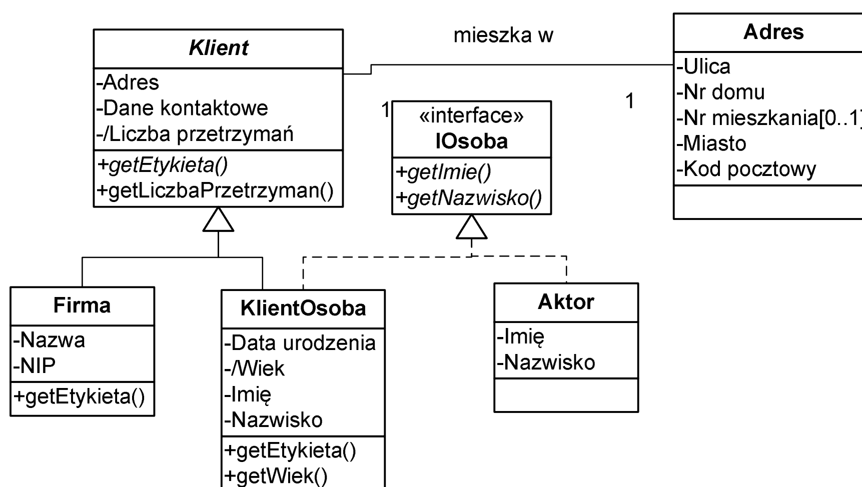
- Rysunek 3-43

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 170) takowe w Javie nie występuje. Co z tym możemy zrobić? Możliwości jest kilka (patrz podrozdział 3.3.5 na stronie 170). 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 atry-

butu 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 94).

Czyli po naszych modyfikacjach odpowiedni fragment diagramu wyglądałby tak jak na rysunku 3-44.

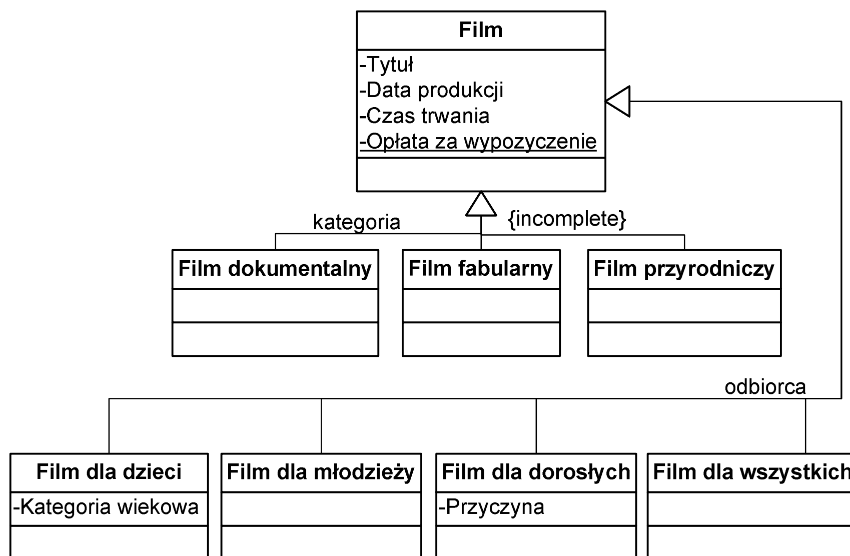


3-44 Tworzenie projektowego diagramu klas – krok 2

- Rysunek 3-45

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 175. 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 razie 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`).



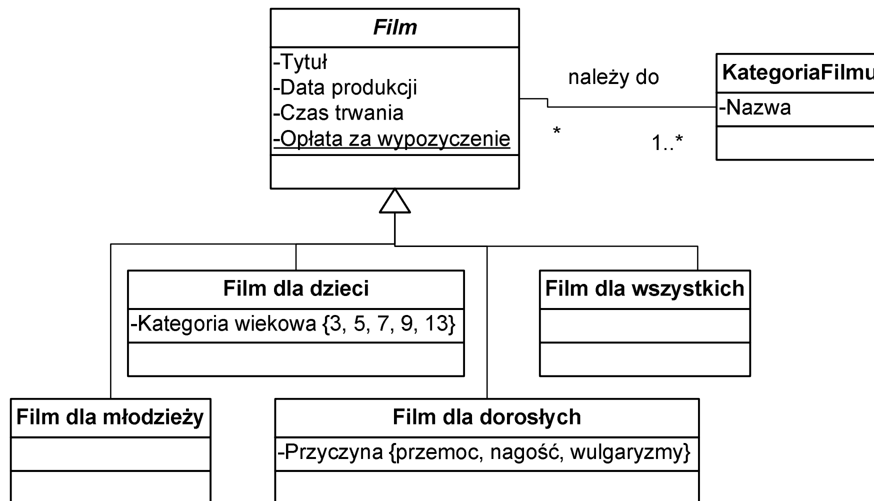
3-45 Tworzenie projektowego diagramu klas – krok 3

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 „* - *”.

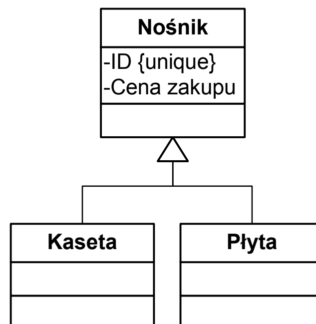
Odpowiedni fragment diagramu po modyfikacjach (przy okazji też oznaczyliśmy klasę Film jako abstrakcyjną) przedstawia rysunek 3-46.



3-46 Tworzenie projektowego diagramu klas – krok 4

- Rysunek 3-47

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.

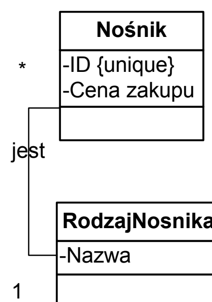


3-47 Tworzenie projektowego diagramu klas – krok 5

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 3-48.

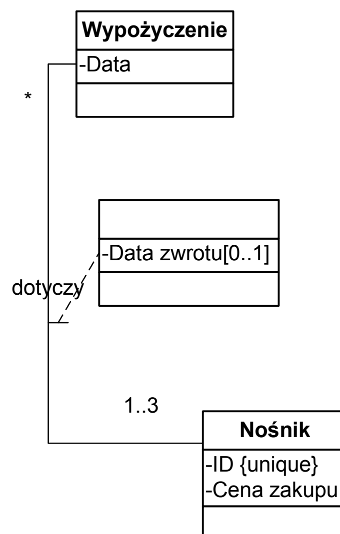


3-48 Tworzenie projektowego diagramu klas – krok 6

- Rysunek 3-49

Informacje dotyczące wypożyczenia opisywane za pomocą asocjacji z atrybutem. Jak ustaliliśmy w podrozdziale 3.2.3.3 (strona 131), taka konstrukcja nie występuje w języku Java. W związku z tym wprowadza-

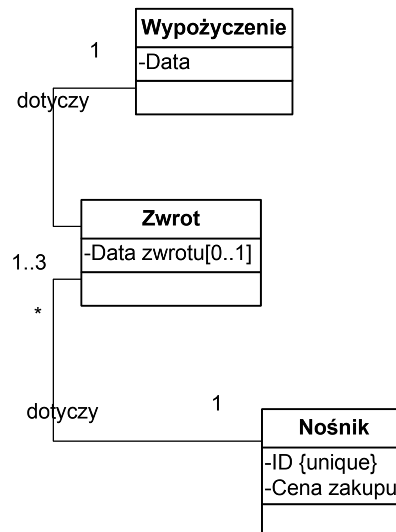
my klasę pośredniczącą. Nazwijmy ją zwrot. Co będzie zawierać? Oczywiście datę zwrotu. Ale czy w momencie tworzenia powiązania będziemy ją znali? Ponieważ mówimy o faktycznej dacie, a nie o zakładanej, to odpowiedź brzmi: nie. Czyli, podobnie jak na diagramie pierwotnym, atrybut będzie opcjonalny. Teoretycznie 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).



3-49 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 3-50.



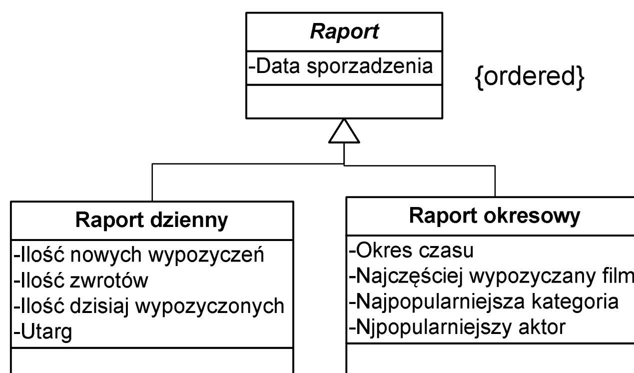
3-50 Tworzenie projektowego diagramu klas – krok 8

- Rysunek 3-51

Ostatnim elementem diagramu, któremu powinniśmy się przyjrzeć, jest część dotycząca raportów. Jak wspomnieliśmy w komentarzu do rysunku 2-66 (strona 82), 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.



3-51 Tworzenie projektowego diagramu klas – krok 9

I tym oto sposobem prawie dobrnęliśmy do końca naszego projektu dotyczącego struktury przechowywanych danych – rysunek 3-52.

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 3-53.

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.

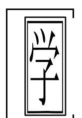
W kolejnym rozdziale poświęconym projektowaniu dyskutowaliśmy, jak mają się poszczególne pojęcia obiektowości oraz analizy do współczesnych języków programowania: głównie Javy oraz trochę MS C# i C++. Omówiliśmy niektóre aspekty projektu systemu informatycznego, ze szczególnym uwzględnieniem ich wpływu 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*.

Wydawnictwo Polsko-Japońskiej

Wyższej Szkoły Technik Komputerowych



ul. Koszykowa 86, 02-008 Warszawa
tel. (22) 58 44 526
faks (22) 58 44 503
e-mail:oficyna@pjwstk.edu.pl
www.pjwstk.edu.pl

W sytuacji, gdy na polskim rynku wydawniczym wciąż brakuje podstawowych publikacji dla studentów informatyki szkół wyższych, istnieje pilna potrzeba wydawania szybko i profesjonalnie podręczników akademickich na wysokim poziomie merytorycznym i edytorskim.

Autorami publikacji Wydawnictwa PJWSTK są przede wszystkim - lecz nie tylko - pracownicy naukowo-dydaktyczni w dziedzinie informatyki, którzy są wykładowcami naszej Uczelni. Wydawane podręczniki prezentują najwyższy poziom wiedzy w zakresie poszczególnych przedmiotów informatycznych wykładanych obecnie w szkołach wyższych w Polsce.

Wydawnictwo publikuje również monografie niezbędne dla rozwoju naukowego społeczności informatycznej, w tym również prace w celu uzyskania przez ich autorów stopnia naukowego doktora, doktora habilitowanego czy też tytułu naukowego profesora z zakresu informatyki.

Nasze publikacje zyskały duże uznanie również u znanych zagranicznych wydawców. Springer-Verlag, wydawnictwo o zasięgu międzynarodowym, zadeklarowało chęć wydania niektórych naszych tytułów w języku angielskim, udostępniając tym samym nasze podręczniki szerokiej rzeszy studentów i pracowników naukowych poza granicami naszego kraju.

Publikacje nasze są adresowane nie tylko do studentów informatyki, lecz również do wszystkich, którzy zainteresowani są pogłębieniem swojej wiedzy i rozwinięciem własnych zainteresowań zawodowych i naukowych.

Mamy nadzieję, że nasza inicjatywa wydawnicza przyczyni się do uzupełnienia wykazu dobrych książek niezbędnych głównie dla wykładowców i studentów informatyki w kraju i za granicą.

Zainteresowanych tą inicjatywą wydawniczą zapraszamy do współpracy.

Nasze książki są do nabycia bezpośrednio w Wydawnictwie lub mogą być przesłane, po wpłacie należności na konto, za pośrednictwem poczty. Dostępne są również w księgarniach, głównie informatycznych i technicznych na terenie całego kraju. W niedalekiej przyszłości przewidujemy prowadzenie sprzedaży w organizowanym przez nas sklepie internetowym.

Dotychczas ukazały się:

1. Kazimierz Subieta „Wprowadzenie do inżynierii oprogramowania”;
2. Krzysztof Barteczko „Podstawy programowania w Javie”;
3. Michał Lentner „Oracle 9i. Kompletny podręcznik użytkownika”;
4. Lech Banachowski, Agnieszka Chądzyńska, Elżbieta Mrówka-Matejewska, Krzysztof Matejewski, Krzysztof Stencel „Bazy danych. Wykłady i ćwiczenia”;
5. Andrzej Bernacki, Tomasz Piątek „Rachunkowość i finanse podmiotów gospodarczych”;
6. Ewa Krassowska-Mackiewicz „Pismo japońskie. Metody transkrypcji”;
7. Grażyna Mirkowska-Salwicka „Elementy matematyki dyskretnej”,
8. Jacek Płodzień, Ewa Stemposz „Analiza i projektowanie systemów informatycznych”;
9. Piotr Ciesielski, Jacek Sawoniewicz, Adam Szmigielski „Elementy robotyki mobilnej”;
10. Tomasz R. Werner „Podstawy C++”;
11. Ewa Krassowska-Mackiewicz „Język japoński dla początkujących”
12. Lech Banachowki, Agnieszka Chądzyńska, Krzysztof Matejewski „Relacyjne bazy danych. Wykłady i ćwiczenia”;
13. Krzysztof Stencel „Systemy operacyjne”;
14. Lech Banachowski, Elżbieta Mrówka-Matejewska, Krzysztof Stencel „Systemy baz danych. Wykłady i ćwiczenia”;
15. Krzysztof Barteczko, Wojciech Drabik, Bartłomiej Starosta „Nowe metody programowania. Tom I”;
16. Krzysztof Barteczko „Programowanie obiektowe i zdarzeniowe w Javie”;
17. Ewa Krassowska-Mackiewicz „Kaligrafia japońska”
18. Piotr Kaźmierczak,, Krzysztof Luks, Lech Polkowski „Elementy robotyki humanoidalnej. Projekt głowy humanoidalnej PALADYN”;
19. Jacek Płodzień, Ewa Stemposz „Analiza i projektowanie systemów informatycznych. Wydanie drugie rozszerzone”
20. Kazimierz Subieta „Teoria i konstrukcja obiektowych języków zapytań”;

21. Matthew Michalewicz, Zbigniew Michalewicz „Wiarygodność: klucz do sukcesu w biznesie”
22. Włodzimierz Dąbrowski, Kazimierz Subieta „Podstawy inżynierii oprogramowania”
23. Krzysztof Barteczko, Wojciech Drabik, Bartłomiej Starosta „Nowe metody programowania. Tom II”
24. Bartłomiej Starosta „JMX. Zarządzanie aplikacjami w języku Java”
25. Anna Korzyńska, Małgorzata Przytulska „Przetwarzanie obrazów – ćwiczenia”
26. Krzysztof Stencel „Półmocna kontrola typów w językach programowania baz danych”
27. Elżbieta Bielecka „Systemy informacji geograficznej. Teoria i zastosowania”
28. Ewa Krassowska Mackiewicz :Podstawowe znaki japońskie”
29. Anna Dańko, Truong Lan Le, Grażyna Mirkowska, Paweł Rembelski, Adam Smyk, Marcin Sydow Algorytmy i struktury danych – zadania”
30. Tadeusz Łagowski „ Systemy informacyjne zarządzania organizacjami gospodarczymi w procesie globalizacji z wspomaganiem informatycznym”
31. Agnieszka Mykowiecka „Inżynieria lingwistyczna”
32. Lech Banachowski, Krzysztof Stencel „Systemy zarządzania bazami danych”
33. Magdalena Kaliszewska, Tomasz Pieciukiewicz, Aneta Sobczak, Krzysztof Stencel „Technologie internetowe”
34. Włodzimierz Pastuszak, „Grafika wydawnicza. Vademecum projektanta”
35. Alicja Wieczorkowska, „Multimedia. Podstawy teoretyczne i zastosowania praktyczne”