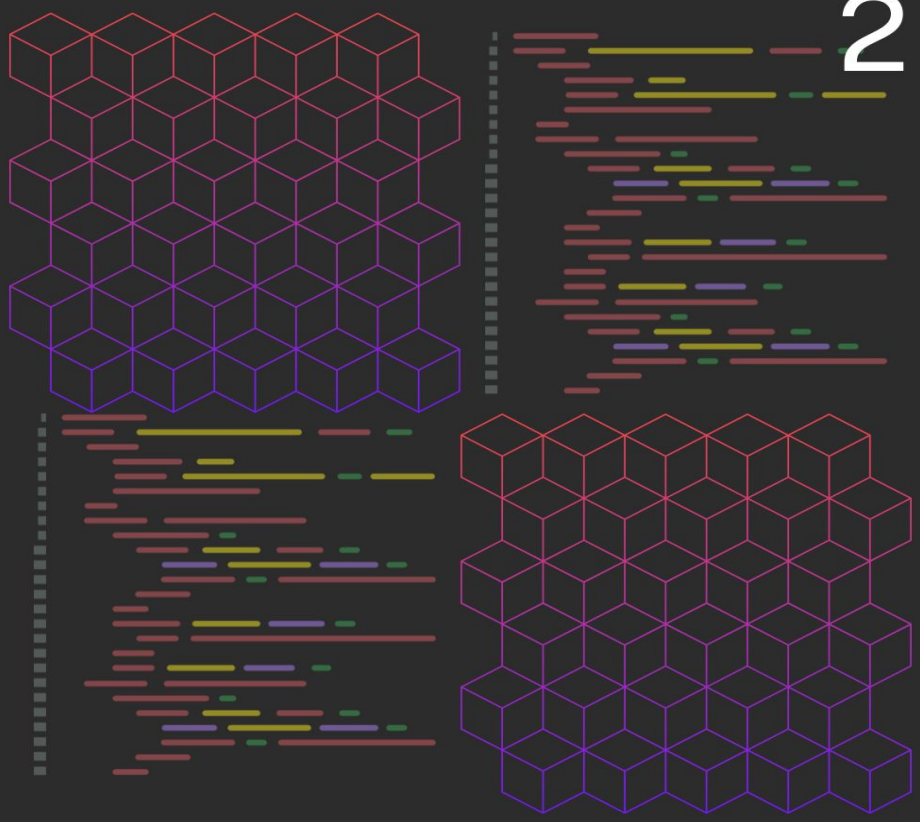
MARIUSZ TRZASKA

MODELOWANIE I IMPLEMENTACJA SYSTEMÓW INFORMATYCZNYCH 2.0



UML • MODELOWANIE • PROGRAMOWANIE

JAVA SPRING • TESTOWANIE

Fragment książki

https://www.mtrzaska.com/

ISBN: 978-83-976442-0-5

e-mail: *mtrzaska@mtrzaska.com*
WWW: *https://www.mtrzaska.com/*

Wydanie 2026-4-8

Spis treści

1. Wprowadzenie (aktualne)

Długo się zastanawiałem, czy i w jakiej formie tworzyć nową wersję oryginalnego wydania tej książki z roku 2008. Od tego czasu minęło sporo lat, wiele rzeczy się zmieniło, włączając w to rynek książek i skłonność do ich czytania. Zachęcające były pytania potencjalnych czytelników o nową wersję i zgłaszane problemy z dostępnością tej oryginalnej. Rozważałem różne scenariusze, począwszy od napisania wszystkiego od nowa, poprzez modyfikacje istniejącej wersji, a kończąc na wariancie w którym w ogóle nic nie piszę.

Ostatecznie założyłem optymistycznie, że znajdą się jacyś zainteresowani i zaczynam prace. Uznałem, że większość zasadniczych pomysłów z oryginalnej książki wytrzymała próbę czasu i nie ma potrzeby ich wymyślać całkiem od nowa. Oczywiście trzeba było je zaktualizować - choćby o nowsze rozwiązanie programistyczne. W efekcie wyszło całkiem sporo zmian, które sprawiają, że rezultat moich prac jest bardziej zbliżony do nowej publikacji niż tylko kolejnego wydania.

Zmieniłem też tematykę wymagań użytkownika, wokół których jest budowana duża część niniejszej książki (analiza, projektowanie, implementacja). Oryginalnie była to wypożyczalnia wideo, o której niektórzy z potencjalnych czytelników może nigdy nie słyszeli. Zamiast tego mamy również wypożyczalnię, ale książek, czyli bibliotekę. Nie była to tylko prosta zamiana nazw, ponieważ w trakcie prac okazało się, że pewne koncepcje nie mają bezpośrednich odpowiedników.

Oprócz aktualizacji wszystkich diagramów, kodów programów, dodałem również całkiem nowe rozdziały/sekcje, których w ogóle nie było pierwotnie,

m.in.:

- implementacja różnych rodzajów aplikacji we frameworku Spring (Data REST, kontrolery Spring, MVC z Thymeleaf),
- testy integracyjne we frameworku Spring,
- testy E2E we frameworku Spring,
- konserwacja oprogramowania.

Sporych aktualizacji doczekał się też rozdział o maperze obiektowo-relacyjnym Hibernate.

O skali zmian może też świadczyć fakt, iż niniejsza edycja książki ma ponad 200 stron więcej niż ta z roku 2008.

Znacząco zaktualizowana i rozbudowana została też bibliografia - teraz ma ponad 100 pozycji. Starałem się znaleźć nowe, interesujące publikacje dotyczące omawianych zagadnień. W niektórych przypadkach (szczególnie tych bardziej teoretycznych jak, np. modelowanie UML, czy dobre praktyki programistyczne) okazało się, że nadal popularne są wydania książek sprzed wielu lat. Z tego powodu niektóre z nich pozostawiłem. Być może są aż tak dobre i/lub w teorii zmiany następują zdecydowanie wolniej, a szybsze są w konkretnych technologiach.

Zastanawiałem się również jak rozwiązać kwestię zwracania się do czytelników oraz feminatywów, np. w nazwach zawodów. Ostatecznie uznałem, że pisanie za każdym razem ”programista/programistka”, ”analityk/analityczka” będzie utrudniało czytanie i z tego zrezygnowałem. Mam nadzieję, że nikt nie poczuje się urażony.

Zmienił się również podział na rozdziały, sekcje itp. Nie zawsze jest idealnym odzwierciedleniem znaczenia poszczególnych koncepcji, ale zbyt dokładne trzymanie się takich reguł prowadziło do zbyt wielu poziomów zagnieżdżeń. Czasami granice są też dość płynne. W efekcie, np. model relacyjny jest poza projektowaniem.

Zastosowałem też pewnie zmiany natury edytorskiej w postaci tzw. uwag. Opisują one różne zagadnienia, które mogą być istotne dla wszystkich zainteresowanych wytwarzaniem systemów informatycznych.

Na koniec tego wprowadzenia, chciałbym prosić o przysyłanie znalezionych błędów (w tym literówek), uwag, komentarzy, pomysłów, sugestii na adres: *mtrzaska@mtrzaska.com*. Postaram się odpowiedzieć na każdego maila i odnieść się do każdej z uwag, a pomysły w miarę możliwości zrealizować w aktualizacjach do niniejszego wydania.

Wprowadzenie (z pierwszego wydania)

Poniżej znajduje się wprowadzenie z pierwotnej wersji książki z roku 2008. Jako, że w dużej części jest aktualne, to zdecydowałem się je zostawić.

Ponad dziesięć lat temu przeczytałem książkę o programowaniu, która mnie urzekła: „Symfonia C++" napisana przez Jerzego Grębosza [Greb96] . 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 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 skrótowo (nie dotyczy to diagramu klas, który jest omówiony bardzo szczegółowo). Czytelnik, który potrzebuje poszerzyć swoją wiedzę na temat tych zagadnień, powinien sięgnąć po którąś z książek traktujących o modelowaniu, UML itp. (lista proponowanych tytułów znajduje się w ostatnim rozdziale: Bibliografia). Większość przykładów w książce oparta jest na konkretnych wymaganiach biznesowych (wypożyczalnia wideo). Implementacja została wykonana dla języka programowania Java SE 6. Czasami też zamieszczam odnośniki do Microsoft C# czy C++. Na początku książki mamy jakiś biznes do skomputeryzowania (wypożyczalnia wideo), przeprowadzamy jego analizę, robimy projekt, a na koniec częściową implementację w postaci prototypu systemu komputerowego.

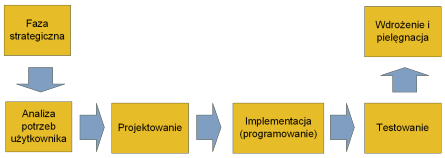
Proces analizy potrzeb użytkownika i projektowania oprogramowania oraz zagadnienia z tym związane (szeroko pojmowana obiektowość) są czasami postrzegane (szczególnie przez niektórych studentów) jako zbędny balast teoretyczny. Można spotkać się z opinią, że należy usiąść i zacząć pisać program (programować),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

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

2. Analiza

Wytwarzanie współczesnego oprogramowania to proces bardzo skomplikowany. Bierze w nim udział cały sztab ludzi, z których każdy ma konkretne zadanie do wykonania. Aby te osoby mogły się wzajemnie porozumieć, muszą mówić wspólnym językiem. W przypadku projektowania systemu informatycznego do tego celu najczęściej używa się notacji UML (*Unified Modeling Language*) [54], [32], [118]. Umożliwia ona w miarę precyzyjne opisanie wszystkich elementów składających się na projekt nowoczesnego oprogramowania. Więcej informacji na temat samego procesu pozyskiwania wymagań można znaleźć np. w książkach [15], [27], [55].

Istnieje wiele różnych metodyk definiujących proces wytwarzania oprogramowania, zaczynając od klasycznych (zwanych często wodospadowymi (*waterfall*)), a kończąc na lekkich (*Agile*). 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. Główna różnica polega na tym, że w podejściu przyrostowym fazy te są powtarzane wielokrotnie (w każdej iteracji), a w klasycznym realizujemy je jednorazowo.



Rysunek 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 jego każdy aspekt, 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 pierwsza (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 7.1.2). 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 section 2.2) opisującą biznes naszego klienta. Celowo wybraliśmy bibliotekę [1], ponieważ w zaproponowanym kształcie posiada większość elementów występujących podczas modelowania systemów komputerowych.

[1] W pierwszej edycji książki zajmowaliśmy się wypożyczalnią wideo, ale ten rodzaj biznesu praktycznie już nie występuje i mógłby być zbyt egzotyczny dla współczesnych czytelników.

2.2 Wymagania dla biblioteki

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 (ulica, numer domu, ewentualnie numer mieszkania, miasto, kod pocztowy),
   d. Dane kontaktowe (nr telefonu, adres e-mail, opcjonalny adres internetowy).
2. Dla klienta firmowego przechowujemy następujące informacje:
   a. Nazwa,
   b. Adres (ulica, numer domu, ewentualnie numer mieszkania, miasto, kod pocztowy),,
   c. NIP,
   d. Dane kontaktowe (nr telefonu, adres e-mail, opcjonalny adres internetowy).
3. W przypadku klientów prywatnych, klientem biblioteki może zostać osoba, która ukończyła 16 lat.
4. System ma przechowywać informacje o wszystkich wypożyczanych podręcznikach, biografiach, ich autorach, tytułach, wydaniach itp.
5. Informacja o publikacji dotyczy:
   a. Tytułu,
   b. Roku stworzenia,
   c. Liczby stron,
   d. Autorów (może być ich więcej niż jeden),
   e. Cyklu tematycznego razem z jego nazwą; nie każda publikacja musi być przypisana do cyklu,
   f. Opłaty pobieranej za wypożyczenie egzemplarza (takiej samej dla wszystkich publikacji).
6. Publikacja powinna zawierać informacje o postaciach w niej występujących razem z ich opcjonalną rolą w konkretnej publikacji, np. postać negatywna, postać pozytywna itd. Postacie mogą być:
   a. Osobami ludzkimi (pamiętamy imię i nazwisko),
   b. jakimiś istotami (przechowujemy informację o ewentualnych cechach szczególnych).
7. Może istnieć wiele egzemplarzy z tą samą publikacją. Każdy egzemplarz posiada unikalny numer identyfikacyjny oraz datę zakupu.
8. Egzemplarz jest zawsze wystąpieniem konkretnego wydania publikacji. Przechowujemy następujące informacje:
   a. numer wydania,

   b. wydawca,
   c. język,
   d. uwagi,
   e. numer ISBN.
9. Publikacje podzielone są według rodzaju na np. biografia, podręcznik, beletrystyka itd. System powinien być dostosowany do przechowywania informacji specyficznych dla poszczególnych rodzajów (niektóre z nich zostaną doprecyzowane w przyszłości):
   a. Biografia – postać, której dotyczy biografia wybrana spośród postaci występujących w danej publikacji.
   b. Podręcznik – informacje o tematyce, dla której jest przeznaczony.
   c. Beletrystyka – informacje o kategoriach.
10. Innym kryterium podziału publikacji 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, wulgaryzmy).
11. Informacja o wypożyczeniu dotyczy daty wypożyczenia oraz opłaty za wypożyczenie.
12. Do jednego wypożyczenia może być przypisane kilka egzemplarzy, jednak nie więcej niż trzy. Każdy z pobranych egzemplarzy może być oddany w innym terminie.
13. Jednocześnie można mieć wypożyczonych maks. 5 egzemplarzy.
14. Egzemplarze wypożycza się standardowo na jeden dzień, płatne z góry. W przypadku przetrzymania egzemplarza opłata za każdy dzień przetrzymania zostaje zwiększona o 10% w stosunku do opłaty standardowej.
15. Jeśli fakt przetrzymania powtórzy się trzykrotnie, klient traci na zawsze prawo do korzystania z biblioteki.
16. Jeśli klient oddał uszkodzony egzemplarz, jest zobowiązany do zwrócenia kosztów nowego.
17. Publikacje przeznaczone wyłącznie dla osób dorosłych może wypożyczyć osoba, która ukończyła 18 lat.
18. Klient musi mieć możliwość samodzielnego przeglądania informacji o publikacjach oraz stanu swojego konta.
19. Codziennie opracowuje się raport dzienny o wydarzeniach w bibliotece, tzn. o:
   a. Liczbie nowych wypożyczeń,
   b. Liczbie zwrotów,
   c. Liczbie dzisiaj wypożyczonych egzemplarzy,

   d. Dziennym utargu.
20. Co jakiś czas opracowuje się raport okresowy (za zadany okres - okresy mogą się nakładać), który zawiera informacje o:
   a. Najczęściej wypożyczanej publikacji,
   b. Najpopularniejszej kategorii,
   c. Najpopularniejszej postaci.
21. Raporty są uporządkowane chronologicznie.

Jak można się zorientować, powyższe wymagania odpowiadają mniej więcej typowej bibliotece (może oprócz postaci występujących w publikacji). 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 15).

Warto również zwrócić uwagę na to, że wymagania mogą się zmieniać w trakcie realizacji projektu. W takim przypadku należy zastosować odpowiednie procedury zmiany wymagań, które zazwyczaj są opisane w umowie zawartej z klientem.

2.3 Przypadki użycia

Diagramy przypadków użycia są bardzo ważnym elementem analizy. Ich głównym zadaniem jest zdefiniowanie funkcjonalności tworzonego systemu. Pojęcie funkcjonalności określa, co system ma robić (jakie funkcje może realizować). Ich odzwierciedleniem będą odpowiednie komendy w menu, wyświetlające dedykowane elementy GUI (*Graphical User Interface* – graficznego interfejsu użytkownika). Ewentualnie, w przypadku tzw. Aplikacji konsolowych (pozbawionych graficznego interfejsu użytkownika) będą miały przełożenie na odpowiednie parametry tekstowe, np. „/remove” lub „-add”. Jednakże na tym etapie tworzenia oprogramowania nie interesuje nas dokładne mapowanie przypadków użycia (funkcji) na konkretne pozycje menu czy przyciski. Nie zajmujemy się też określaniem, jak te funkcje mają być dokładnie wykonywane – to jest przedmiotem fazy projektowania.

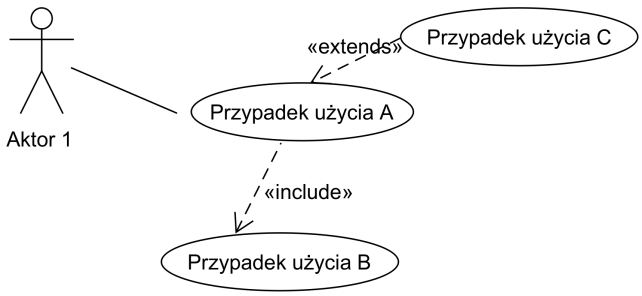
Przeprowadzenie dokładnej analizy jest o tyle istotne, że rzutuje na dalsze prace projektowe. Wyobraźmy sobie sytuację, że na tym etapie zapomniano o uwzględnieniu jakiejś funkcji. Oznacza to, że nie zostanie ona uwzględniona przy projektowaniu, implementacji oraz testowaniu. Po prostu jej nie będzie

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

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

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

Diagramy przypadków użycia są stosunkowo proste. Ich najważniejsze elementy to (zobacz rysunek 2.2):



Rysunek 2.2: Przykładowy diagram przypadków użycia służący jako ilustracja notacji.

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

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

Przypadki użycia mogą być powiązane ze sobą na dwa sposoby:

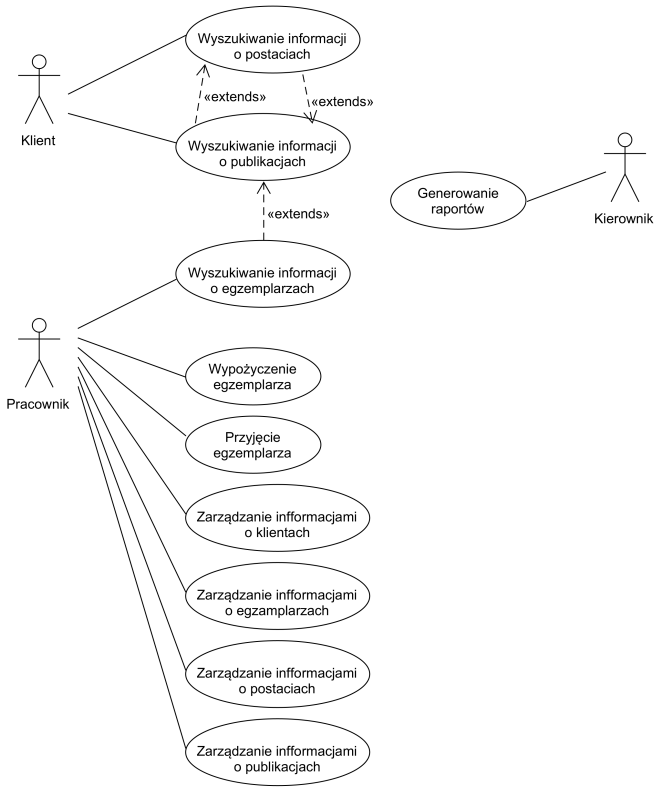
- *«include»* oznacza, że przypadek użycia *A* zawsze korzysta (wywołuje, uruchamia) przypadek użycia *B*,

- *«extend»* formalnie oznacza, że przypadek użycia *A* jest rozszerzany przez przypadek użycia *C*. Mówiąc po polsku: *A* czasami używa *C*. Nie mamy informacji jak często. Jeżeli dodatkowo wyspecyfikowaliśmy punkt rozszerzalności (*extension point*), to wiemy, kiedy to ewentualne użycie zachodzi. Warto zwrócić uwagę na zwrot strzałek: mówiąc, że *A* czasami używa *C*, pokazujemy z *C* do *A* (wzięło się to z tego, że C „rozszerza" A).

2.3.1 Ogólny diagram przypadków użycia dla Biblioteki

Diagramy przypadków użycia można rysować na różnym poziomie szczegółowości. Zwykle robi się jeden (lub więcej dla bardzo rozbudowanych systemów) ogólny diagram i uszczegóławia się go na oddzielnych diagramach. Nasuwa się pytanie: jak bardzo należy wchodzić w szczegóły? Nie ma jednej precyzyjnej odpowiedzi. Przeważnie robimy to tak, aby dało się z niego odczytać dość ogólne informacje. Do przedstawiania bardziej szczegółowych i precyzyjnych danych służą inne diagramy (np. aktywności – podrozdział 2.5, stanów – podrozdział 2.6; warto także zajrzeć do książki [25]).

Rysunek 2.3 zawiera przykładowy, ogólny (zawierający wszystkie funkcje) diagram dla naszej biblioteki. Słowo przykładowy nie jest przypadkowe. Inny analityk (np. ty, Czytelniku) mógłby narysować (trochę) inny diagram. Jego kształt oraz zakres pokazanych funkcji zależy od dwóch głównych czynników:



Rysunek 2.3: Diagram przypadków użycia dla biblioteki.

- Uwarunkowania biznesowe, czyli co ten system ma robić (jakie ma mieć funkcje). Część z nich staje się jasna dopiero po zadaniu wielu pytań naszemu zleceniodawcy. Niektórzy analitycy je zadadzą, inni nie. Może to sugerować po prostu błędny diagram. Owszem może, ale nie musi – patrz następny punkt (ktoś mógł uznać, że pewne informacje są zbyt szczegółowe).
- Sposobu modelowania (postrzegania pewnych zjawisk) konkretnego analityka. Nawet mając te same dane wejściowe (sytuację biznesową), różni analitycy mogą stworzyć różne diagramy. Naturalnie część z nich może być błędna (mniej lub bardziej), ale też może istnieć wiele poprawnych (też mniej lub bardziej) i zarazem różnych rozwiązań. Te różnice są szczególnie widoczne w przypadku bardziej skomplikowanych diagramów (np. klas – patrz rozdział 2.4). Można w takim razie zapytać; jakie są kryteria poprawności – który z nich jest lepszy,

a który gorszy? i znowu nie ma jednoznacznej odpowiedzi – po prostu różni ludzie widzą te same sprawy w różny sposób (i jak pewnie nieraz mogłeś się przekonać, drogi Czytelniku, nie dotyczy to tylko modelowania). Na pewno błędnym diagramem będzie taki, który nie przykrywa pewnej funkcjonalności, tworzy nową, o której zleceniodawca nie wspominał (tak – i w tym przypadku nadgorliwość jest wadą) albo po prostu utrudnia jej zrealizowanie.

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

- Aktorzy. Jak łatwo się zorientować, ich liczba oraz zakres „obsługiwanych" funkcji nie wynika wprost z wymagań na system (patrz podrozdział 2.2, 16). Przykładowo, funkcje systemu związane z dodawaniem informacji o nowych publikacjach oraz egzemplarzach mógłby pełnić dedykowany aktor. W prawdziwym systemie tego typu decyzje są uzależnione od czynników biznesowych (np. wielkości firmy). Na potrzeby tej książki chcieliśmy ograniczyć ich liczbę. Załóżmy, że w toku długich rozmów z klientem doszliśmy do wniosku, że właśnie tacy aktorzy będą obsługiwać nasz system.

- Zwróćmy też uwagę na umieszczenie aktorów na diagramie. Zwykle są oni zlokalizowani na krańcach diagramu, aby podkreślić fakt, iż nie są częścią analizowanego systemu; czasami przypadki użycia dodatkowo oddziela się graficznie, otaczając je prostokątem, a aktorzy pozostają na zewnątrz tego prostokąta.

- Przypadki *„Wyszukanie informacji o postaciach"* oraz *„Wyszukanie informacji o publikacjach"* są wzajemnie powiązane przy pomocy *«extend»*. Sens tego jest taki, że szukając informacji o publikacjach (a właściwie konkretnej publikacji), możemy chcieć zobaczyć postacie, które tam występują. Możliwa jest też sytuacja odwrotna: mając konkretną postać, możemy chcieć zobaczyć „jej" publikacje. Ponieważ możemy to zrobić, ale nie zawsze będziemy korzystali z tej funkcji, zastosowaliśmy *«extend»*, który jak wcześniej powiedzieliśmy, oznacza opcjonalne wykorzystanie.

- Przypadek użycia *„Wyszukiwanie informacji o egzemplarzach"* rozszerza „Wyszukiwanie informacji o publikacjach". Oznacza to, że czasami, wyświetlając informacje o publikacji, chcemy zobaczyć listę egzemplarzy, na których się znajduje.

- *„Wypożyczenie egzemplarza"* oraz *„Przyjęcie egzemplarza* są przypi-

sane do aktora Pracownik. Jest tak dlatego, że to właśnie pracownik obsługuje system w momencie wypożyczania/zwracania. Gdybyśmy chcieli zrobić system dla biblioteki samoobsługowej, to właściwym aktorem byłby klient.

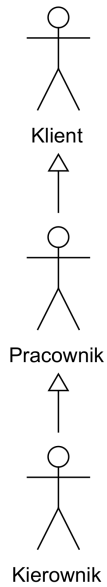
- Grupa przypadków użycia *„Zarządzanie informacjami o ...”*. Bierzemy tu pod uwagę dodawanie, usuwanie i edycję. Oczywiście można dyskutować, czy nie powinniśmy do tego włączyć też wyszukiwania: raczej nie, ponieważ wyszukiwanie jest dostępne dla klienta, a dodawanie/usuwanie pewnie nie powinno.

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

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

- Uważny czytelnik (mam nadzieję, że to jesteś właśnie Ty), mógłby się zdziwić: chwileczkę, ale jak to – *Kierownik* nie ma dostępu do funkcji zwykłego pracownika czy wręcz klienta (analogicznie z funkcjami pracownika oraz klienta)? Faktycznie trochę dziwne. Zastanówmy się, jakie mamy możliwości, aby to zmienić:
  - Pierwsza to połączenie wszystkich przypadków użycia klienta z aktorem *Pracownik*. Analogicznie łączymy wszystkie przypadki pracownika z *Kierownikiem*. W rezultacie mamy straszną gmatwaninę linii na diagramie (a diagram z definicji powinien być czytelny).
  - Narysować oddzielny diagram pokazujący zależności pomię-

dzy aktorami (lub nanieść je na istniejący diagram – tak też można). I taki właśnie diagram jest na rysunku 2.4. Specjalny znaczek (strzałka z trójkątnym grotem) oznacza dziedziczenie. Czyli wszystkie przypadki użycia nad-aktora są dostępne dla pod-aktora, odpowiednio: *Klient* i *Pracownik* oraz *Pracownik* i *Kierownik*. Rozwiązanie z dziedziczeniem jest czytelniejsze oraz umożliwia łatwe wprowadzanie zmian (modyfikujemy tylko hierarchię dziedziczenia aktorów, co oznacza zmiany dostępnych przypadków użycia dla wszystkich zaangażowanych aktorów).

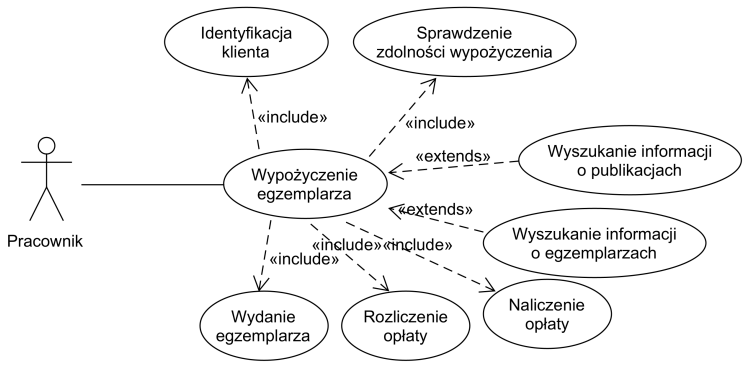


Rysunek 2.4: Diagram dziedziczenia dla aktorów.

2.3.2 Szczegółowy diagram przypadków użycia

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

Rysunek 2.5 zawiera uszczegółowioną wersję jednego z przypadków użycia („Wypożyczenie egzemplarza”) przedstawionych ogólnie na rysunku



Rysunek 2.5: Przykładowy, uszczegółowiony diagram przypadków użycia.

2.3 (strona 21). Poniżej zamieszczono jego skrótową analizę:

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

- Następnie musimy być pewni, że dany klient (uprzednio zidentyfikowany) może coś wypożyczać. W wymaganiach na system, które z takim trudem ustaliliśmy, są podane pewne warunki (patrz m.in. punkt 12 i dalsze, strona 17), które muszą być spełnione.

- Opisując powyższe przypadki użycia zastosowaliśmy sformułowania takie jak „następnie”, „uprzednio”. Jest to naturalne, ponieważ chcemy zachować pewien porządek, który zwiększy nasze szanse na niepominięcie niczego istotnego. W związku z tym należy się zastanowić, w jakiej kolejności umieszczamy przypadki użycia na diagramie? Od prawej do lewej, z góry na dół? Pytanie jest podchwytliwe – otóż diagramy przypadków użycia nie uwzględniają kolejności. Dlatego

kolejność dołączania poszczególnych elementów („pod-przypadków" użycia) jest dowolna. Zwykle robimy to zgodnie z kierunkiem ruchu wskazówek zegara, ale jest to tylko zdroworozsądkowy zwyczaj.

- Na pewno zwróciłeś uwagę na to, że część elementów jest dołączona korzystając z *«include»*, a inne przy użyciu *«extend»*. Jak mówiliśmy wcześniej, użycie *«include»* oznacza, że coś jest zawsze wykonywane, a *«extend»*, że tylko czasami. Tutaj jest tak samo. Zastanowienie może budzić tylko kwestia, dlaczego, w świetle powyższego rozumowania, *„Wyszukanie informacji o publikacjach*" oraz *„Wyszukanie informacji o egzemplarzach"* są wykonywane czasami. Przecież witalną częścią biblioteki jest wyszukiwanie powyższych informacji. Oczywiście, że tak, ale może wystąpić kilka scenariuszy:
  - Klient przychodzi i mówi, że chce książkę „Wiedźmin tom I". W takiej sytuacji musimy odnaleźć egzemplarz, który ją zawiera.
  - Klient przychodzi i mówi, że chce po prostu fajną książkę fanstasy. System przedstawia kilka propozycji (ponieważ beletrystyka należy do jakieś kategorii – patrz punkt 9, strona 17). Ale klientowi żaden z nich nie odpowiadał, więc nic nie wypożyczył – czyli nie musieliśmy szukać egzemplarza.
  - Wreszcie, może być tak, że klient chce wypożyczyć egzemplarz o numerze 6341. Wtedy nie musimy wyszukiwać publikacji.

  Jak widać, możliwości jest dość dużo. I dlatego wybraliśmy najbardziej uniwersalne rozwiązanie: obydwa przypadki użycia są wykorzystywane opcjonalnie. Rysując diagram przypadków użycia, należy założyć pewien scenariusz, który określa, co się stanie, jakie decyzje podejmie klient itp. W przeciwnym wypadku większość naszych przypadków użycia wyglądałaby podobnie: najpierw identyfikacja, którą przeprowadzamy zawsze (*«include»*), a później same połączenia z *«extend»'ami* (ponieważ zawsze może nie dojść do wykonania przypadku użycia).
- *„Naliczenie opłaty"* [2] polega po prostu na wyliczeniu należności za wypożyczane egzemplarze. Ponieważ wykorzystaliśmy *«include>*, więc łatwo się domyślić, że zakładamy, iż klient wybrał coś do wypożyczenia.
- *„Rozliczenie opłaty"* jest nazwą dość ogólną. Analogicznie jak przy *„Identyfikacji klienta"*, specjalnie nie chcemy wnikać w szczegóły, np.

[2] Oczywiście biblioteka, która wymaga opłat jest dość rzadkim zjawiskiem, ale pewnie można takie znaleźć.

rozliczenie może być za pomocą karty, gotówki czy po prostu dopisania do rachunku opłacanego przelewem raz w miesiącu.

- *„Wydanie egzemplarza”* oznacza, że cały proces wypożyczania został szczęśliwie zakończony i powinien zostać zapamiętany w systemie. Celowo nie piszemy czegoś w rodzaju *„zatwierdzenie transakcji w bazie danych”*, ponieważ przypadki użycia z definicji unikają wszelkiego „technicyzowania” (odwołań do technicznych aspektów implementowanego systemu).

2.4 Diagram klas

Diagram klas jest uznawany chyba za najważniejszy z diagramów wykorzystywanych w czasie tworzenia oprogramowania. I nie jest tak bez przyczyny: informacje, które się na nim znajdują, są bezpośrednio wykorzystywane przy implementacji (naturalnie, jeżeli korzystamy z obiektowego języka programowania takiego jak Java, C# czy C++). Definiują sposób przechowywania informacji w naszym systemie, a poprzez umieszczenie metod pełnią rolę swoistego szkieletu umożliwiającego zdefiniowanie zachowania się tworzonej aplikacji (patrz uwaga 2.1). Tak naprawdę możemy mówić przynajmniej o dwóch rodzajach diagramów klas:

Uwaga 2.1: Metody na diagramie klas?

Wiemy, że notacja UML pozwala na umieszczanie metod na diagramie klas. Czy zawsze powinniśmy to robić? Wbrew pozorom odpowiedź nie jest jednoznacznie twierdząca:

- aby to właściwie zrealizować na etapie analizy, czy nawet projektowania, trzeba mieć dobre rozeznanie programistyczne, co nie zawsze jest oczywiste,
- w przypadku dość prostych projektów, funkcjonalności mogą być umieszczane bezpośrednio w metodach modelu i w takim przypadku, faktycznie diagram klas nam to umożliwi,
- w zależności od zastosowanej technologi, metody realizujące wymagania funkcjonalne nie muszą być implementowane w klasach modelu; co więcej w wielu współczesnych sytuacjach tak się nie robi, np. wzorzec MVC gdzie stosujemy np. kontrolery i serwisy (patrz rozdział 7.3)

- ewentualnie można umieścić tam proste metody, np. obliczające wiek osoby.

Możemy też traktować metody z diagramu klas jako ogólne ”przypomnienie” wymagań funkcjonalnych, które muszą być zrealizowane. Nie jestem jednak przekonany, czy to rzeczywiście będzie bardzo pomocne.

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

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

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

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

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

Spotyka się też podejście, polegające na niewykorzystywaniu tych „trudnych" konstrukcji. Pewną korzyścią będzie łatwiejsze projektowanie oraz implementacja, ale bardzo istotną wadą takiego postępowania może być nadmierne skomplikowanie fazy analizy (m.in. poprzez trudniejsze „odczytywanie" diagramu). Czyli, innymi słowy, to co przeanalizowaliśmy, łatwiej zaprojektujemy, ale zwiększamy prawdopodobieństwo popełnienia błędów we wcześniejszej fazie (analizy). A jak pamiętamy z poprzednich rozdziałów, im błąd wcześniej popełniony, tym większe problemy może sprawić. Dlatego, ja osobiście przestrzegam przed unikaniem tych konstrukcji na etapie analizy.

Myślę, że teraz jest właściwy moment, abyśmy przeanalizowali poszczególne elementy występujące na diagramie klas. Jedna drobna uwaga: jak pisaliśmy wcześniej, książka ta jest poświęcona głównie przejściu od fazy analizy, poprzez projektowanie do implementacji. Nie będziemy się koncentrowali na bardzo dokładnym i szczegółowym omówieniu pojęć związanych z obiektowością jako taką. Więcej informacji na ten temat można znaleźć w [32], [118] czy [25].

2.4.1 Obiekt

Każda z osób, które zetknęły się z programowaniem w jednym ze współczesnych języków programowania (np. Java, C# czy C++), ma jakieś własne, intuicyjne wyobrażenie obiektu. Spróbujmy je doprecyzować, na razie zapominając o językach programowania (bez obawy - wrócimy do nich w rozdziale 3 „Projektowanie"). Nasza definicja będzie oparta na tej pochodzącej z [25] (z niewielkimi zmianami):

Definicja 2.1 Obiekt

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

Obiekt jest również nazywany *instancją klasy* - patrz dalej.

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

2.4.2 Klasa

Jak sama nazwa wskazuje, podstawowym elementem znajdującym się na diagramie klas są klasy. Cóż to takiego jest klasa? Naukowcy zajmujący się problemami obiektowości nie są do końca zgodni, ale myślę, że jedną z

najlepszych definicji może być definicja pochodząca z [25]:

Definicja 2.2 Klasa
nazwany opis grupy obiektów o podobnych własnościach.

Zgodnie z tą definicją, klasa nie jest zbiorem obiektów (do tego jest oddzielne pojęcie). Jest to bardzo ważne stwierdzenie, ponieważ często jest to mylone. Innymi słowy, cytując z pamięci [36]: klasa jest jakby przepisem na obiekt. Opisuje jego cechy (atrybuty), zachowanie (metody) oraz budowę (w kontekście języków programowania).

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

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

Definicja 2.3 Ekstensja klasy
zbiór aktualnie istniejących wszystkich wystąpień (innymi słowy: instancji lub jak kto woli: obiektów) danej klasy.

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

Tak naprawdę powyższa definicja nie jest do końca precyzyjna (ktoś wie dlaczego?), ponieważ nie określa zależności w stosunku do obiektów z podklas oraz nadklas (wrócimy do tego przy okazji dziedziczenia – patrz podrozdział 2.4.5.2).

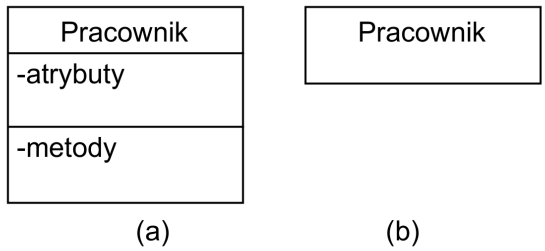
Klasy możemy podzielić na:

- Konkretne,
- Abstrakcyjne.

Aby dobrze zrozumieć ten podział, konieczne jest wprowadzenie pojęcia dziedziczenia. Dlatego wrócimy do tego tematu w podrozdziale 2.4.5.

Rysunek 2.6 zawiera różne warianty notacji służącej do przedstawiania klasy na diagramie:

- Część (a) pokazuje kompletną klasę (razem z atrybutami oraz metodami),
- Część (b) przedstawia klasę bez pokazanych metod oraz atrybutów. Notacja zezwala na ukrywanie niepotrzebnych (brakujących) elementów.



Rysunek 2.6: Notacja dla klas.

2.4.2.1 Atrybuty

Atrybuty służą do opisu własności obiektów należących do pewnej klasy. Przyjmuje się, że ich wartości znajdują się wewnątrz konkretnych obiektów i nie są współdzielone z innymi obiektami. Oznacza to, że każdy obiekt ma swój indywidualny zestaw, nawet jeżeli w wielu z nich znajdują się te same wartości (szczególnym przypadkiem są atrybuty klasowe - patrz dalej). Oczywiście wartości atrybutów mogą być zmieniane w trakcie życia obiektu.

Poniżej znajdują się różne rodzaju atrybutów z uwzględnieniem kryteriów podziału:

- budowa:
  - **prosty**; przechowuje atomowe (niepodzielne) wartości, m. in.: liczba lub napis. Przykłady: pensja, nazwisko, nazwa koloru, waga;
  - **złożony**; Składa się z atrybutów prostych lub innych atrybutów złożonych. Przykłady: adres (może się składać z ulicy, numeru domu, miasta, kodu pocztowego), współrzędne położenia w przestrzeni 3D (składają się z wartości dla osi x, y, z).
- liczność:
  - **pojedynczy**; posiada jedną wartość. Przykłady: data urodzenia, płeć;
  - **powtarzalny**; posiada jedną lub więcej wartości. Przykłady: imię (bo ktoś może mieć wiele imion).

- przynależność:
  - **obiektu**; każdy obiekt w klasie może mieć własną wartość. Przykłady: dla klasy *Pracownik* przykładami mogą być: nazwisko, imię, data urodzenia.
  - **klasowy**; Ma tę samą wartość dla wszystkich obiektów danej klasy. Przykłady: Dla klasy *Pracownik* przykładami mogą być: minimalny wiek, maksymalna pensja.
- ustalanie wartości:
  - **konkretny**; wartość atrybutu jest przechowywana bezpośrednio w obiekcie. Przykłady: Data urodzenia.
  - **wyliczalny**; Wartość atrybutu może być wyliczona na podstawie innych danych (np. atrybutów). Czasami przechowujemy tę wyliczoną kopię. Przykłady: wiek (obliczony na podstawie daty urodzenia oraz aktualnej daty systemowej).
- obowiązkowość:
  - **wymagany**; Jeżeli nie oznaczono inaczej to atrybut musi mieć wartość. Przykłady: nazwisko w klasie pracownik.
  - **opcjonalny**; Atrybut może nie mieć wartości. Przykłady: Nazwisko panieńskie w klasie pracownik (mężczyźni zwykle nie mają nazwiska panieńskiego).

Spójrzmy jeszcze na dodatkowy komentarz dotyczący różnych rodzajów atrybutów w klasie:

- Atrybut powtarzalny powinien mieć nazwę w liczbie pojedynczej, np. Imię. Jeżeli nazwiemy go imiona, to będzie oznaczało, że mamy wiele imion (wnioskujemy to z jego nazwy, a zatem znaczenia). Jeżeli dodatkowo oznaczymy go jako powtarzalny, to całość będzie oznaczała jakąś listę imion, z której każdy element będzie miał wiele wartości (w tym przypadku imion). Czyli będzie to lista list, a raczej nie o to nam chodziło.
- Jak widać z przykładu dotyczącego atrybutu obiektu, nie ma przeszkód, aby wiele (lub wszystkie) obiekty danej klasy miały taką samą wartość danego atrybutu (w przykładzie jest data urodzenia, która dla odpowiednio dużej liczby osób na pewno się powtórzy).
- Zaletą korzystania z atrybutu klasowego (ta sama wartość dla wszyst-

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

- Atrybut wyliczalny budzi, przynajmniej na początku, sporo wątpliwości. Po co przechowywać dane dwa razy? Dla jasności: jego istnienie niekoniecznie musi oznaczać dosłowne przechowywanie. Często możemy wyliczać jego wartość w razie potrzeby (pewnie zrobilibyśmy tak z wiekiem). W innych sytuacjach, rozsądniej będzie zapamiętać w systemie tę obliczoną wartość (czyli wystąpi jakaś forma redundancji danych[3]). Kryterium decyzyjnym jest oczywiście koszt obliczenia oraz częstość dostępu:
  - ustalenie wieku nie kosztuje zbyt dużo czasu procesora,
  - gdybyśmy jednak mieli atrybut, np. średnia wartość towarów sprzedanych przez handlowca, to w przypadku dużej firmy jego obliczenie (na podstawie faktur) może zająć trochę czasu. W takiej sytuacji rozsądne wydaje się przechowywanie jego kopii. W efekcie pewne informacje są zapisane w systemie wielokrotnie i w przypadku aktualizacji należy zadbać o zmianę wszystkich „egzemplarzy". Dlatego stosujemy specjalną notację, która nas ostrzega: „uwaga – redundancja danych"!

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

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

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

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

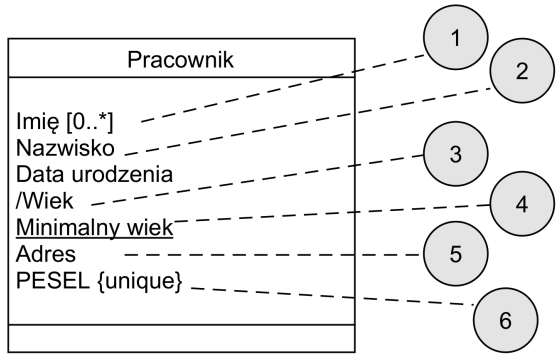
- Atrybut opcjonalny może nie mieć wartości. Jest to użyteczne co najmniej w dwóch przypadkach:
  - Wartość nie ma sensu z biznesowego punktu widzenia, np. w firmie mamy pracowników będących mężczyznami oraz kobietami, ale z jakichś powodów nie chcemy dla nich tworzyć oddzielnych klas. Dlatego wystąpienia (obiekty) klasy Pracownik opisujące mężczyzn nie będą przechowywały wartości dla atrybutu *Nazwisko panieńskie* (ewentualnie dla obu płci możemy przechowywać *nazwisko rodowe*). Oczywiście można to obejść i zapamiętywać tam np. pusty ciąg tekstowy, ale z punktu widzenia systemu będzie tam wartość, tylko że specyficzna.

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

Rysunek 2.7 zawiera notację UML służącą do oznaczania różnych rodzajów atrybutów. Pominięto sposób oznaczania typu atrybutu – będzie o tym przy okazji projektowania. Kilka uwag:

1. Atrybut powtarzalny oznaczamy pokazując jego liczności. Zapis [0…*] oznacza dowolną liczność, a np. [1…4] mówi, że atrybut może mieć od 1 do 4 wartości. Tak naprawdę, brak informacji o liczności oznacza po prostu liczność [1].
2. Brak specjalnych symboli zakłada, że mamy atrybut z licznością [1], czyli wymagany, obiektu i konkretny.
3. Oznaczenie atrybutu wyliczalnego.
4. Atrybut klasowy. Tutaj mamy na myśli minimalny wiek, powyżej którego osoba może zostać naszym pracownikiem.
5. Adres jest przykładem atrybutu złożonego. Jak widać, ten rodzaj nie ma żadnych specjalnych oznaczeń.

[4]Ograniczenia są jednym z mechanizmów rozszerzalności UML i ogólnie rzecz biorąc, umożliwiają umieszczenie na diagramie informacji, których nie jesteśmy w stanie wyrazić przy pomocy np. klas czy atrybutów. Mogą być zapisywane zwykłym tekstem, korzystając ze specjalnego języka OCL czy wyrażeń matematycznych. Umieszczamy je w nawiasach klamrowych. Więcej w rozdziale 2.4.6.



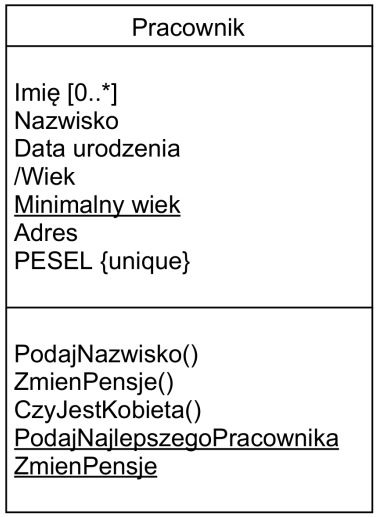
Rysunek 2.7: Notacja do oznaczania atrybutów.

6. W ten sposób mówimy systemowi, że ten atrybut ma mieć unikalną wartość wśród obiektów danej klasy.

2.4.3 Metody

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

- Metody obiektu. Operują na konkretnym obiekcie. Mają dostęp do jego atrybutów oraz innych metod tego obiektu (a dokładniej: innych metod zdefiniowanych w klasie, do której należy ten obiekt). Nie mamy możliwości bezpośredniego odwołania się do innego obiektu (w tym jego zawartości) – nawet z tej samej klasy. Można powiedzieć w uproszczeniu, że pojedynczy obiekt nie jest świadomy istnienia innych obiektów danej klasy (chyba że dostanie się do nich poprzez ekstensję). Przykłady to (z dokładnością do nazewnictwa):
  - `PodajNazwisko():String`, zwracamy nazwisko konkretnego pracownika (tego, na rzecz którego wywołaliśmy tę metodę).
  - `ZmienPensje(NowaPensja):void`, zmieniamy pensję konkretnego pracownika.
  - `CzyJestKobieta():boolean`, pytamy czy konkretny pracownik jest kobietą.
- Metody klasowe. Mają dostęp do całej ekstensji, a zatem do wszystkich

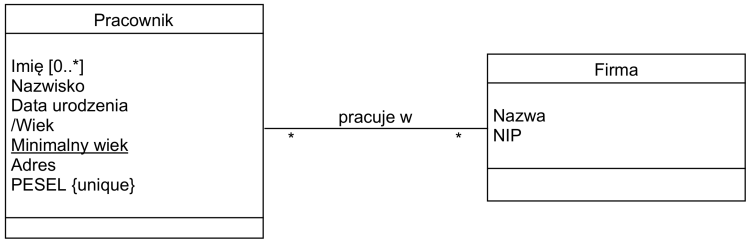


Rysunek 2.8: Notacja do oznaczania metod.

obiektów należących do danej klasy. Mogą na nich operować, ale nie muszą: może być np. metoda klasowa, która zwróci jakąś wartość bez „zaglądania" do ekstensji (dlatego stwierdzenie, że metody klasowe operują na całej ekstensji, nie jest do końca precyzyjne). Analogicznie jak w przypadku atrybutów, metody klasowe wywołujemy na rzecz klasy, a nie konkretnego obiektu. Dzięki temu można z nich korzystać nawet, gdy nie ma jeszcze obiektów danej klasy. Przykłady:

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

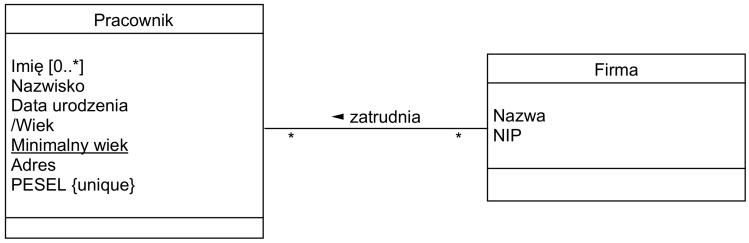
Rysunek 2.8 pokazuje notację służącą do umieszczania metod na diagramie (pominięto informacje o parametrach oraz zwracanym typie, pokazano również atrybuty). Jak widać, oznaczenie metody klasowej jest analogiczne do oznaczenia atrybutu klasowego: podkreślona nazwa.



Rysunek 2.9: Przykład ilustrujący wykorzystanie asocjacji.

2.4.4 Asocjacje

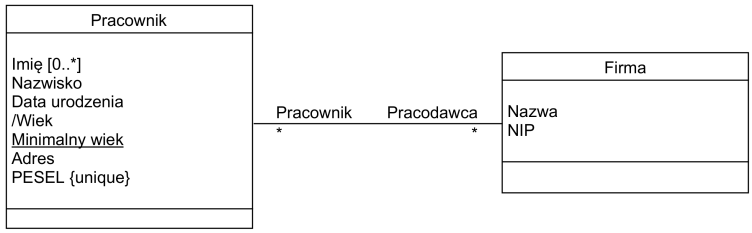
Asocjacje służą do łączenia klas, które mają jakieś biznesowe zależności. Dzięki temu możemy np. zapamiętać, że *Pracownik* (klasa) *pracuje w* (asocjacja) *Firmie* (klasa). Sytuację tę ilustruje rysunek 2.9. Nazwa asocjacji powinna być tak dobrana, aby w połączeniu z nazwami klas utworzyła sensowne wyrażenie, np. Pracownik pracuje w Firmie. Domyślnie nazwa asocjacji ma poprawne znaczenie przy czytaniu od lewej do prawej. Jeżeli w powyższym przykładzie wolimy utworzyć nazwę z punktu widzenia firmy, wtedy brzmiałaby ona zatrudnia i należałoby ją czytać od prawej do lewej. Sytuacje tę ilustruje rysunek 2.10. Warto zwrócić uwagę na mały trójkącik przy nazwie pokazujący właściwy kierunek czytania. Podkreślmy: kierunek czytania nazwy, a nie kierunek asocjacji. Jeżeli nie zaznaczono inaczej, to asocjacja jest dwukierunkowa: w naszym przykładzie możemy przejść od firmy do pracownika oraz od pracownika do firmy.



Rysunek 2.10: Przykład ilustrujący wykorzystanie asocjacji. W stosunku do rysunku 2.9 zmieniono nazwę oraz jej kierunek czytania

Umieszczenie nazwy to nie jedyny sposób opisania asocjacji. Zamiast niej można wykorzystać nazwy ról (patrz rysunek 2.11). Należy pamiętać, że na diagramie są umiejscowione przy klasie docelowej, np. rola *Pracodawca*

w klasie *Pracownik*. Przy takim podejściu mamy kompleksowy opis: z punktu widzenia każdej z klas. Jest to szczególnie użyteczne, gdy na dalszych etapach będziemy przekształcali diagram do wersji implementacyjnej, korzystając z narzędzia CASE. Więcej o tym będzie w kolejnych rozdziałach.

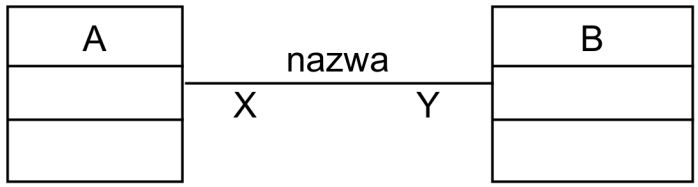


Rysunek 2.11: Ilustracja wykorzystania ról asocjacji

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

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

Można się zastanawiać, po co w ogóle podawać informacje o licznościach. Przecież można by się umówić, że wszędzie jest „*” i możemy zapamiętać



Rysunek 2.12: Liczności asocjacji

każdą sytuację biznesową. Głównym powodem jest chęć pilnowania pewnych reguł, np. jeżeli gdzieś występuje liczność „1”, to nie będziemy mogli stworzyć obiektu („system” na to nie pozwoli), który nie będzie powiązany z dokładnie jednym obiektem. Dodatkowym powodem są kwestie implementacyjne, ponieważ inaczej się zapamiętuje liczności typu „jeden”, a inaczej „wiele”.

Jak, mam nadzieję, pamiętamy, wystąpieniem (instancją) klasy jest obiekt. Podobna zależność jest też dla asocjacji - spójrzmy na definicję:

Definicja 2.4 Powiązanie
wystąpienie asocjacji pomiędzy konkretnymi obiektami, np. informacja, że pracownik ”Kowalski” pracuje w firmie ”Microsoft”.

2.4.4.1 Asocjacja binarna

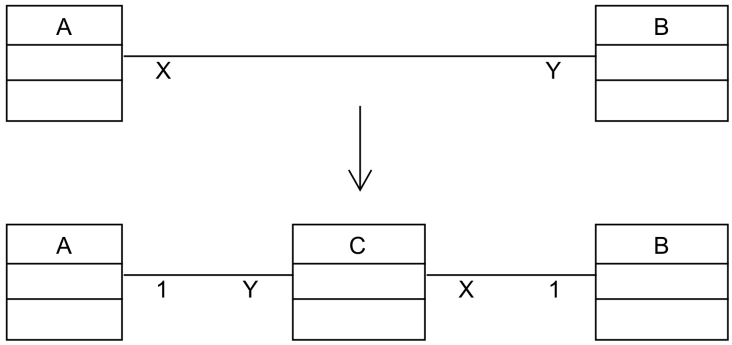
Asocjacja binarna jest to taka asocjacja, która występuje pomiędzy dokładnie dwoma klasami. Tak naprawdę już znamy ten rodzaj asocjacji, ponieważ to, co do tej pory napisaliśmy o asocjacjach, charakteryzuje właśnie ją.

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

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

Czasami (np. gdybyśmy chcieli przejść na model relacyjny) ta ostatnia kategoria może sprawić nam spory problem. Dlatego może zastosować przejście pokazane na rysunku 2.13. Wprowadzamy klasę pośredniczącą C i zamieniamy jedną asocjację „wiele-do-wielu” (pomiędzy klasami A i B) na dwie „jeden-do-wielu” (pomiędzy A-C oraz C-B). Całe przejście jest automa-

tyczne i zawsze wykonywane według tego samego schematu. Warto zwrócić uwagę na pozorny błąd w licznościach: na nowym diagramie widzimy, że liczności X oraz Y zamieniły się miejscami. Sprawdźmy: z punktu widzenia klasy A występowała liczność Y, a z punktu widzenia klasy B była liczność X. Okazuje się, że nadal tak jest (proszę sprawdzić!).



Rysunek 2.13: Redukcja liczności dla asocjacji

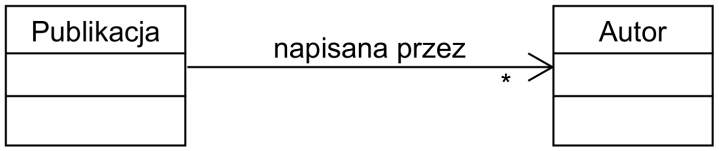
Nie zawsze da się wymyślić sensowną nazwę dla klasy C, dlatego czasami jej nazwę stanowi zlepek nazw sąsiadujących klas (np. zamiast C mogliśmy napisać AB).

2.4.4.2 Asocjacja skierowana

Asocjacja skierowana to taka asocjacja, która ma określony kierunek. W poprzednich przykładach asocjacje były dwukierunkowe, co oznacza, że można było przejść od jednej klasy do drugiej i odwrotnie. W przypadku asocjacji skierowanej nie jest to możliwe. Zwykle chcielibyśmy móc przejść w obu kierunkach. W rzadkich wypadkach, gdy chcemy zaznaczyć, że przejście jest możliwe tylko w jedną stronę, stosujemy asocjację skierowaną. Innym powodem mogą być kwestie wydajnościowe, gdy zależy nam na oszczędzaniu pamięci. Niemniej, biorąc pod uwagę, moc współczesnych komputerów, ten powód zwykle nie ma zastosowania. W notacji UML, asocjację skierowaną oznaczamy strzałką, która wskazuje kierunek przejścia (rysunek 2.14). Zauważmy, że zdefiniowaliśmy tylko jedną liczność (ponieważ nie ma połączenia w drugą stronę).

2.4.4.3 Asocjacja n-arna

Ten rodzaj asocjacji sprawia sporo problemów i z tego powodu w praktyce jest rzadko wykorzystywany. Powiem więcej, raczej odradzam korzystanie z tej

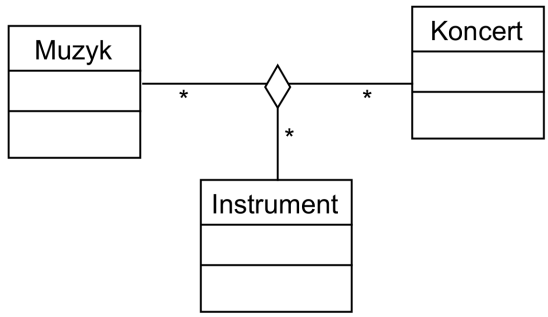


Rysunek 2.14: Przykład asocjacji skierowanej

konstrukcji, tym bardziej, że bez problemu można ją zastąpić rozwiązaniem równoważnym (patrz podrozdział 3.7.8 na stronie 168).

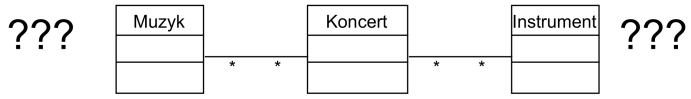
Problemy biorą się ze sposobu określania liczności oraz biznesowego wyobrażenia sobie danej sytuacji. Najczęściej zakłada się, że asocjacja n-arna powinna mieć wszędzie liczność „wiele", a procedura ustalania tych liczności nawiązuje do asocjacji binarnych. Szacowany koniec asocjacji „konfrontujemy" z pozostałymi końcami połączonymi w jedną, wirtualną całość. Rysunek 2.15 przedstawia przykładową asocjację n-arną opisującą zależność pomiędzy muzykiem, koncertem oraz instrumentem. Nazwijmy ją (zależność oraz asocjacje) „granie". Zakładamy, że:

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



Rysunek 2.15: Przykładowa asocjacja n-arna

Pozornie mogłoby się wydawać, że wystarczy połączyć koncert z muzykiem oraz koncert z instrumentem za pomocą dwóch asocjacji binarnych. W rzeczywistości tak nie jest (patrz rysunek 2.16. Przy takim podejściu co prawda wiedzielibyśmy, którzy muzycy grali na danym koncercie i jakie instrumenty wykorzystano, ale nie wiedzielibyśmy, który muzyk grał na którym instrumencie (w ramach tego koncertu). Jak widać z powyższego wywodu, musimy zastosować asocjację o trzech krańcach, ponieważ angażuje ona więcej niż dwie klasy (i dlatego asocjacja binarna nie wystarczy). Na szczęście istnieje prawidłowy sposób zastąpienia asocjacji n-arnej – poznamy go w rozdziale 3.7.8 (168). Jeżeli jest to biznesowo uzasadnione, stopień asocjacji (n) może być wyższy i równy 3, 4, 5 itd.



Rysunek 2.16: Ilustracja problemu z asocjacją n-arną

2.4.4.4 Asocjacja kwalifikowana

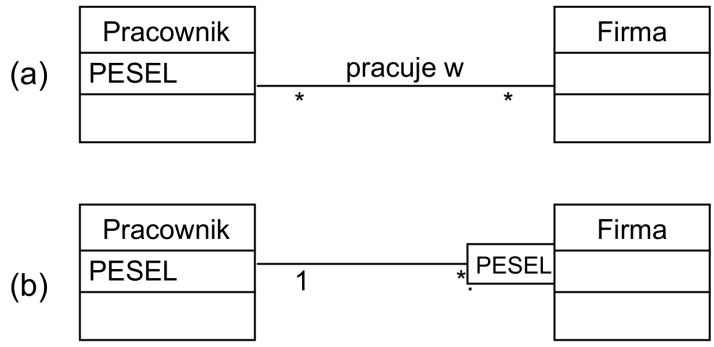
W odróżnieniu od asocjacji n-arnej asocjacja kwalifikowana jest dość chętnie stosowana – czasami nawet zbyt często i bez zrozumienia (a szczególnie jej implementacyjna inkarnacja – patrz podrozdział 3.7.7). W klasycznej asocjacji binarnej, aby znaleźć konkretny obiekt znajdujący się „po drugiej stronie", musimy przeglądać wszystkie powiązania, aż znajdziemy obiekt spełniający nasze kryteria (np. pracownik z określonym numerem PESEL). I tutaj właśnie z pomocą przychodzi nam asocjacja kwalifikowana, a szczególnie kwalifikator. Spójrzmy na definicję:

Definicja 2.5 Kwalifikator
atrybut (lub kombinacja atrybutów) umożliwiający jednoznaczne zidentyfikowanie obiektu docelowego.

Rysunek 2.17 składa się z dwóch części:

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

- (b) również opisuje związek pracownika z firmą, ale za pomocą asocjacji kwalifikowanej. Dzięki temu rozwiązaniu, mając „w ręku" kon-



Rysunek 2.17: Asocjacja binarna (a) oraz kwalifikowana (b)

kretną wartość kwalifikatora (numer PESEL), otrzymujemy odpowiadający mu obiekt (pracownika). Warto zwrócić uwagę na dwie rzeczy:

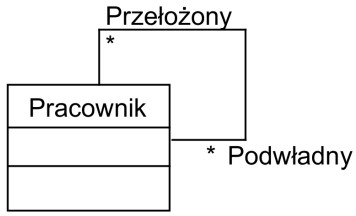
- Atrybut będący kwalifikatorem jest usuwany z pola atrybutów „swojej" klasy (na diagramie) i przenoszony obok klasy docelowej,
- Ulega zmianie liczność (często też się o tym mówi redukcja liczności). Jest to spowodowane tym, że myślimy w kategoriach pary (w tym przypadku *PESEL* i *Firma*): dla konkretnej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

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

2.4.4.5 Asocjacja rekurencyjna (zwrotna)

Asocjacja rekurencyjna (zwana też zwrotną) polega na tym, że występuje w ramach tej samej klasy. Aby móc prawidłowo zidentyfikować znaczenie każdego z końców asocjacji, obowiązkowo musimy zastosować nazwy ról. Rysunek 2.18 pokazuje klasyczny przykład umożliwiający przechowywanie informacji o zależnościach służbowych w przedsiębiorstwie (w analogiczny sposób można modelować np. informacje o stosunkach własnościowych: firma jest właścicielem innych firm).

> **Uwaga 2.2: Nazwy asocjacji, a role**
>
> Zastanówmy się jeszcze przez chwilę, dlaczego asocjacja rekurencyjna wymaga dwóch ról, a w zwykłej wystarczy nam jedna nazwa? Odpowiedź jest prosta: w zwykłej asocjacji mamy do czynienia z dwiema różnymi klasami, a więc pełnione role możemy wywnioskować na podstawie przynależności obiektów do tych klas. W przypadku rekurencyjnej asocjacji mamy do czynienia z jedną klasą, a więc nie mamy takiej możliwości. Dlatego musimy ręcznie określić, co oznacza każdy z końców asocjacji.



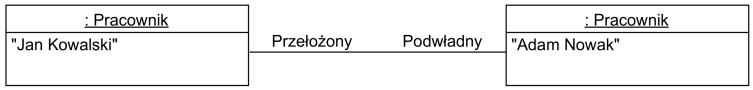
Rysunek 2.18: Przykładowa asocjacja rekurencyjna (zwrotna)

Ten rodzaj asocjacji zwykle nie sprawia problemów koncepcyjnych. Należy tylko zadbać o właściwe liczności (dopuszczające „0”): w naszym przykładzie, gdzieś na szczycie hierarchii, będzie pracownik, który nie ma szefa (ponieważ szef wszystkich szefów nie ma już szefa). Analogicznie, na samym dole będzie pracownik, który nie ma podwładnych.

Warto jeszcze podkreślić, że asocjacja rekurencyjna, co prawda występuje w ramach tej samej klasy, ale zwykle łączy dwa różne obiekty - patrz przykładowy diagram obiektów na rysunku 2.19).

Uwaga 2.3: Diagram obiektów, a diagram klas

Diagram obiektów, w odróżnieniu od diagramu klas, zawiera już konkretne wystąpienia – obiekty; notacja jest wzorowana na notacji dla diagramu klas i może pokazywać wartości atrybutów oraz powiązania, a nie asocjacje.



Rysunek 2.19: Przykładowy diagram obiektów dla asocjacji rekurencyjnej z rysunku 2.18

2.4.4.6 Asocjacja z atrybutem (klasa asocjacji)

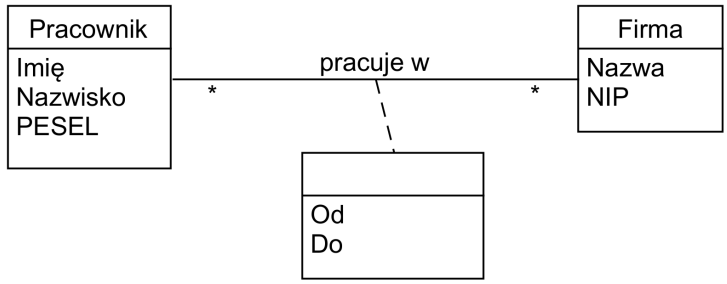
W niektórych sytuacjach zachodzi potrzeba przechowania dodatkowych informacji na temat konkretnego powiązania (wystąpienia asocjacji). Przypomnijmy sobie przykład z rysunku 2.17 część (a) (strona 43). Jak widzimy, ten diagram umożliwia nam zapamiętanie informacji o miejscach pracy poszczególnych pracowników. Załóżmy, że chcielibyśmy mieć informacje o okresie czasu, który spędzili pracując w poszczególnych firmach. Jak możemy to zrobić? Najprostszym pomysłem, jaki przychodzi do głowy, jest wstawienie odpowiednich atrybutów do którejś z klas:

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

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

Czyżby nie dało się tego zrobić? Oczywiście, że się da, ale musimy zastosować nową konstrukcję: *klasę asocjacji* (rysunek 2.20; dodaliśmy też atrybuty do istniejących klas). Cóż to takiego jest? Jest to zwykła klasa (ma takie same cechy jak każda inna), ale „podłączona" do asocjacji. Wystąpienia klasy asocjacji niosą dodatkowe informacje dotyczące konkretnego powiązania (wystąpienia asocjacji). Można o niej myśleć jak o „karteczce" przyczepionej do nitki obrazującej powiązanie.

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



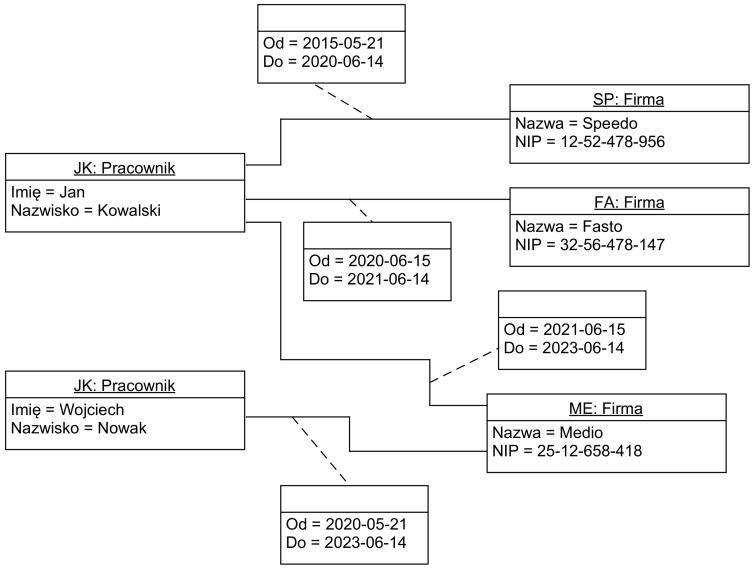
Rysunek 2.20: Zastosowanie klasy asocjacji

Uwaga 2.4: Stosowanie klasy asocjacji

Co właściwie spowodowało, że musieliśmy użyć tej nowej konstrukcji? Głównym powodem są liczności "wiele" występujące po obu stronach asocjacji. Gdyby gdzieś było "1" lub "0..1", to moglibyśmy zastosować atrybuty w jednej z klas. W naszym przypadku, aby zapamiętać informacje o okresie pracy, musieliśmy zastosować klasę asocjacji.
W związku z tym, umieszczając klasę asocjacji na diagramie, upewnij się, że liczności są odpowiednie: "* - *. W naszym przypadku mamy do czynienia z taką sytuacją, więc wszystko jest OK.

Aby ułatwić zrozumienie mechanizmu klasy asocjacji spójrzmy na rysunek 2.21, który zawiera diagram obiektów (a nie klas). Widzimy dwa wystąpienia klasy *Pracownik*, trzy wystąpienia klasy *Firma* oraz cztery instancje klas asocjacji. Na podstawie diagramu możemy powiedzieć, że:

- Jan Kowalski pracował w trzech firmach:
  - „Speedo" od 2015-05-21 do 2020-06-14,
  - „Fasto" od 2020-06-15 do 2021-06-14,
  - „Medio" od 2021-06-15 do 2023-06-14
- Wojciech Nowak pracował tylko w firmie „Medio" od 2020-05-21 do 2023-06-14. Zwróćmy też uwagę, że jest to ta sama firma w której pracował też Jan Kowalski. Innymi słowy, konkretny obiekt może być połączony z wieloma innymi obiektami. Naturalnie tylko wtedy gdy liczności na to pozwalają – w naszym przykładzie mamy „wiele do wielu" (rysunek 2.20), więc wszystko jest OK.

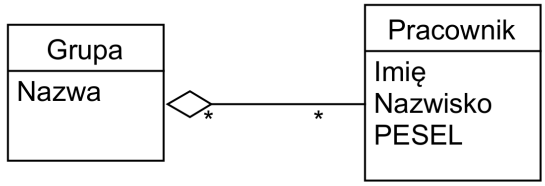


Rysunek 2.21: Diagram obiektów ilustrujący wykorzystanie klasy asocjacji

2.4.4.7 Agregacja i kompozycja

Agregacja i kompozycja są szczególnymi rodzajami asocjacji (głównie binarnych). Ich dodatkowym zadaniem, oprócz modelowania jakiejś zależności,

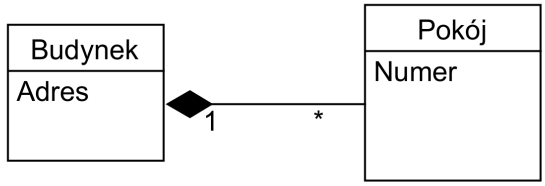
jest podkreślanie związku typu część-całość. Rysunek 2.22 przedstawia przykładową agregację opisującą grupę składająca się z pracowników. Symbol rombu, oznaczający właśnie agregację, umieszczamy przy „całości". Możemy również podać informację dotyczącą liczności. Jeżeli chodzi o nazwę, to zwykle ją pomijamy, ponieważ wykorzystanie agregacji umożliwia odczytywanie tego związku jako: „składa się", „zawiera", „jest częścią" itp.



Rysunek 2.22: Przykładowa agregacja

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

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



Rysunek 2.23: Przykładowa kompozycja

Warto zwrócić uwagę, że powyższe cechy skutkują pewnymi konkretnymi licznościami. Rysunek 2.23 zawiera przykładową kompozycję (zamalowany romb) opisującą budynek oraz jego pokoje. Ten przykład jest o tyle dosłowny, że z fizycznego punktu widzenia nie może istnieć pokój

bez budynku. Nie zawsze tak musi być: można również zastosować kompozycję w przykładzie z rysunku 2.22, jeżeli tylko mamy do tego biznesowe uzasadnienie:

- Pracownik należy tylko do jednej grupy. Warto zwrócić uwagę, że jeżeli chcielibyśmy trzymać historię przydziałów do grup, to docelowa liczność „1" nam na to nie pozwoli. Innymi słowy: nie będzie mógł istnieć obiekt klasy *Pracownik* połączony z wieloma obiektami klasy *Grupa*.
- W modelowanej firmie nie może być pracownika nie przypisanego do grupy,
- Usunięcie grupy oznacza też usunięcie jej pracowników.

Uwaga 2.5: Stuprocentowa kompozycja

Niewiele jest przykładów takich bezdyskusyjnych kompozycji (*Budynek-Pokój*), które w każdym systemie powinny być tak przedstawiane. Przykładowo *samochód-silnik* może być kompozycją, gdy sprzedajemy tylko kompletne samochody z silnikami, a agregacją w sytuacji, gdy można kupić również sam silnik (bo silnik mógłby istnieć bez samochodu).

2.4.5 Dziedziczenie

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

Następne podrozdziały zawierają krótkie omówienie poszczególnych rodzajów dziedziczenia i tematów z tym związanych. Bardziej szczegółowe informacje można znaleźć w [25], [32] czy [118].

2.4.5.1 Dziedziczenie pojedyncze

Dziedziczenie pojedyncze jest najprostszym rodzajem dziedziczenia. Rysunek 2.24 zawiera typowy przykład opisujący hierarchię dziedziczenia dla klasy *Osoba*. Przeanalizujmy ten diagram: