helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

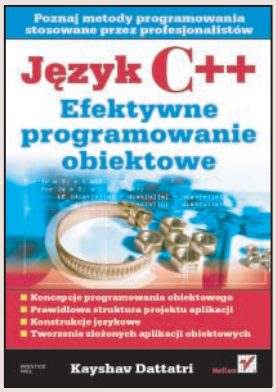
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Język C++. Efektywne programowanie obiektowe



Autor: Kayshav Dattatri
Tłumaczenie: Michał Grzegorczyk, Jaromir Senczyk, Przemysław Steć, Przemysław Szeremiota, Tomasz Walczak
ISBN: 83-7361-812-0
Tytuł oryginału: C++: Effective Object-Oriented Software Construction
Format: B5, stron: 800

Programowanie obiektowe jest nierozerwalnie związane z językiem C++. Koncepcje i metody programowania obiektowego, niezbędne do swobodnego posługiwania się tą techniką, pomimo pozornej prostoty są stosunkowo trudne do opanowania. Projektowanie aplikacji w języku C++ wymaga jednak nie tylko znajomości podstawowych zasad programowania obiektowego, ale również wielu innych technik programistycznych. Należy prawidłowo zaplanować strukturę aplikacji, poznać zasady pisania poprawnego kodu i nauczyć się korzystać z notacji UML do modelowania zależności pomiędzy elementami aplikacji.

„C++. Efektywne programowanie obiektowe" to podręcznik przeznaczony zarówno dla początkujących, jak i zaawansowanych programistów C++. Przedstawia metody programowania obiektowego stosowane przez profesjonalistów. Opisuje techniki obiektowe w kontekście rzeczywistych problemów, przed jakimi stają twórcy oprogramowania podczas codziennej pracy.

- Podstawowe pojęcia i koncepcje programowania obiektowego
- Abstrakcja danych
- Notacja UML
- Zarządzanie pamięcią w programowaniu obiektowym
- Dziedziczenie
- Zasady programowania generycznego
- Obsługa wyjątków
- Zaawansowane aplikacje obiektowe

Dzięki zawartym w tej książce wiadomościom wykonasz nawet najtrudniejsze zadania programistyczne, wykorzystując techniki obiektowe.

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

Spis treści

Rekomendacje **13**

Przedmowa **15**

Wstęp **17**

Część I Pojęcia, techniki i aplikacje **21**

Rozdział 1. Czym jest programowanie obiektowe? **23**

Pochodzenie 23
- Przykład programowania proceduralnego 24
- Reprezentacja konta bankowego 25
- Bezpieczeństwo konta bankowego 26
- Rozwiązywanie problemów w programowaniu obiektowym 27

Wprowadzenie do modelu obiektowego 28
Terminologia 30
Poznajemy komunikaty, metody i zmienne egzemplarza 30
- Z czego składa się obiekt? 31
- Tworzenie obiektów 32

Co można uznać za klasę? 33
Co nie jest klasą? 34
Cel klasy 35
Więcej o obiektach 36
- Stan obiektu 36
- Dlaczego stan obiektu jest ważny? 37
- Kto kontroluje stan obiektu? 38
- Zachowanie obiektu 39

Etapy w konstruowaniu oprogramowania obiektowego 40
- Analiza obiektowa 40
- Projektowanie obiektowe 41

Programowanie obiektowe 43
Kluczowe elementy modelu obiektowego 43
Paradygmaty i języki programowania obiektowego 45
Jakie wymagania musi spełniać język obiektowy? 46
Zalety modelu obiektowego 47
Podsumowanie 48

Rozdział 2. Czym jest abstrakcja danych? 49
Analiza projektu odtwarzacza 51
Oddzielanie interfejsu od implementacji 52
Znaczenie interfejsu 52
Dlaczego interfejs obiektu jest tak ważny? 53
Jaki interfejs jest wystarczający? 53
Znaczenie implementacji 54
Ochrona implementacji 54
Jakie są korzyści ukrywania danych? 56
Relacje między interfejsem, implementacją i hermetyzacją danych 57
Środki ostrożności przy hermetyzacji danych 58
Co i kiedy ukrywać? 58
Abstrakcyjne typy danych 59
Implementacja abstrakcyjnego typu danych — stosu 60
Abstrakcja danych w języku C++ 62
Regiony dostępu klasy 63
Niektóre pojęcia związane z klasami 69
Kto jest implementatorem klasy? 70
Implementowanie funkcji składowych 70
Identyfikacja obiektu docelowego w funkcjach składowych 71
Przykładowy program 73
Uwaga skoncentrowana jest na obiekcie 74
Drugi rzut oka na interfejs 75
Czym są bezpieczne klasy wielowątkowe? 76
Zapewnianie niezawodności abstrakcji — niezmienniki i asercje klasy 78
Niezmienniki klasy 79
Warunki wstępne i warunki końcowe 79
Używanie asercji do implementowania niezmienników i warunków 80
Efektywne korzystanie z asercji 81
Sposoby reprezentacji projektów obiektowych 82
Notacja Boocha 83
Relacje między klasami 83
Asocjacja 84
Agregacja (ma-coś) 84
Relacja korzystania 86
Relacja dziedziczenia (jest-czymś) 87
Kategorie klas 88
UML 88
Relacje między klasami 90
Asocjacja 90
Asocjacja jako agregacja 92
Asocjacja typu OR 93
Kompozycja 93
Relacja uogólniania (jest-czymś) 94
Znaczenie relacji ma-coś 95
Podsumowanie 97

Rozdział 3. Abstrakcja danych w języku C++ 99
Podstawowe informacje o klasie 99
Elementy klasy 100
Regiony dostępu 100
Konstruktor kopiujący 103
Dostęp do danych składowych obiektu — model języka C++ 106
Operacja przypisania 111
Więcej o wskaźniku this i dekorowaniu nazw 116
Metoda stała (const) 118

Jak kompilator implementuje metody stałe? ... 120
Różnice między klasą a strukturą w języku C++ ... 120
Co może zawierać klasa? ... 121
W czasie projektowania najważniejszy jest interfejs klasy ... 122
Nazwy klas, nazwy metod, typy argumentów i dokumentacja ... 123
Sposoby przekazywania argumentów z perspektywy klienta ... 124
Semantyka własności ... 128
Wybór odpowiedniego sposobu przekazywania argumentu ... 130
Wartości zwracane przez funkcję ... 131
Zwracanie referencji ... 133
Jak napisać bezpieczną pod względem pamięci klasę? ... 134
Optymalizacja kodu ... 134
Obowiązki klienta w pracy z klasami i funkcjami ... 134
Podsumowanie ... 136

Rozdział 4. Inicjalizacja i zwalnianie pamięci w programowaniu obiektowym ... 137
Co to jest inicjalizacja? ... 137
Inicjalizacja za pomocą konstruktora ... 139
Reguły pracy z obiektami zagnieżdżonymi ... 146
Zagadnienia związane z przywracaniem pamięci ... 146
Śmieci ... 146
Wiszące referencje ... 147
Jak zapobiegać powstawaniu śmieci i wiszących referencji? ... 148
Przywracanie pamięci a projektowanie języka ... 149
Kiedy powstają śmieci w języku C++? ... 151
Kiedy obiekt zajmuje zasoby? ... 152
Przywracanie pamięci w języku C++ ... 152
Tożsamość obiektów ... 154
Semantyka kopiowania obiektów ... 157
Semantyka prostej operacji kopiowania ... 158
Semantyka przypisywania obiektów ... 163
Przypisanie jako operacja na l-wartości ... 166
Semantyka porównywania obiektów ... 166
Równość obiektów a ekwiwalencja ... 168
Dlaczego potrzebna jest kontrola kopiowania? ... 170
Przykład semafora ... 171
Przykład — serwer licencji ... 173
Przykład — klasa String ... 174
Analiza ... 180
Kopiowanie przy zapisie ... 182
Kiedy używać zliczania referencji? ... 188
Podsumowanie kopiowania przy zapisywaniu ... 188
Klasy i typy ... 189
Podsumowanie ... 190

Rozdział 5. Dziedziczenie ... 191
Podstawy dziedziczenia ... 191
Znaczenie relacji dziedziczenia ... 205
Skutki relacji dziedziczenia ... 205
Bezpośrednie i pośrednie klasy bazowe ... 206
Zasada podstawiania polimorficznego ... 207
Inicjalizacja obiektów klasy bazowej ... 210
Rozszerzanie hierarchii klas za pomocą dziedziczenia ... 213
Podstawowe zalety dziedziczenia ... 215
Wiązanie dynamiczne, funkcje wirtualne i polimorfizm ... 216
Co oznacza wiązanie dynamiczne? ... 219
Obsługa wiązania dynamicznego — funkcje wirtualne ... 220

Wpływ dziedziczenia na hermetyzację danych ... 222
Co oznacza polimorfizm? ... 224
Efektywne stosowanie funkcji wirtualnych ... 225
Przesłanianie ... 225
Kiedy potrzeba wirtualnego destruktora? ... 228
Konstruktory i funkcje wirtualne ... 231
Uogólnianie-uszczegóławianie ... 233
Klasy abstrakcyjne ... 233
Zastosowania klas abstrakcyjnych ... 237
Zaawansowany przykład klasy abstrakcyjnej — gra w szachy ... 241
Waga dziedziczenia ... 249
Efektywne wielokrotne używanie kodu ... 250
Klient abstrakcyjnej klasy bazowej ... 253
Podsumowanie zalet dziedziczenia ... 254
Zagrożenia związane z dziedziczeniem i wiązaniem dynamicznym ... 256
Jak implementowane są funkcje wirtualne w języku C++? ... 257
Koszty funkcji wirtualnych ... 258
Dynamiczne wiązanie i sprawdzanie typu ... 259
Zbędne używanie dziedziczenia i wiązania dynamicznego ... 259
Wypożyczanie zbiorów bibliotecznych ... 259
Różne sposoby używania funkcji wirtualnych ... 270
Podsumowanie ... 272

Rozdział 6. Dziedziczenie wielokrotne ... 273

Prosta definicja dziedziczenia wielokrotnego ... 273
Abstrakcja uniwersytetu ... 274
Powtórne wykorzystanie kodu z ulepszeniami ... 278
Znaczenie wielokrotnego dziedziczenia ... 280
Przykład dziedziczenia wielokrotnego ... 281
Rozwiązywanie konfliktów nazw w języku C++ ... 282
Problem z wieloznacznością klas bazowych ... 285
Podstawowe zalety dziedziczenia wielokrotnego ... 287
Rozwiązania alternatywne dla dziedziczenia wielokrotnego ... 287
Pierwsze rozwiązanie alternatywne ... 287
Drugie rozwiązanie alternatywne ... 290
Problem powtórnego dziedziczenia ... 291
Rozwiązanie problemu powtórnego dziedziczenia ... 295
Dzielenie obiektów za pomocą wirtualnych klas bazowych ... 295
Zalety wirtualnych klas bazowych ... 297
Nowe problemy wynikające z użycia wirtualnych klas bazowych ... 297
Porównanie dziedziczenia wielokrotnego w językach Eiffel i C++ ... 302
Ogólne problemy z dziedziczeniem ... 304
Dodawanie statycznych możliwości za pomocą klas mieszanych ... 306
Definicja klasy mieszanej ... 306
Kiedy należy używać klas mieszanych? ... 310
Dynamicznie zmieniająca się sytuacja ... 311
Elastyczność projektu z klasami pełniącymi różne role ... 316
Jak używać klas pełniących różne role? ... 316
Inne rozwiązanie zarządzania rolami ... 324
Polimorficzne używanie obiektów TUniversityMember ... 326
Wprowadzanie zmian w istniejących klasach ... 326
Klasy mieszane a obiekty pełniące role — możliwości zastosowań ... 328
Wyprowadzenie prywatne w języku C++ ... 330
Kiedy używać wyprowadzenia prywatnego? ... 332
Ponowne eksportowanie składowych prywatnej klasy bazowej ... 334
Alternatywne rozwiązanie dla wyprowadzenia prywatnego — zawieranie ... 335
Kiedy potrzebne jest prywatne wyprowadzenie? ... 337

Bardzo użyteczny przykład dotyczący klas mieszanych i wyprowadzenia prywatnego 340
Dziedziczenie a zawieranie 345
Podsumowanie 347

Rozdział 7. Selektywny eksport z klas (funkcje zaprzyjaźnione) 349
Czego nam potrzeba? 349
Eksport selektywny w języku C++ 350
Konsekwencje związku przyjaźni 353
Zastosowania funkcji niebędących składowymi oraz funkcji zaprzyjaźnionych 357
Przypadek 1. Minimalizowanie silnych interakcji pomiędzy klasami 357
Przypadek 2. Przezwyciężanie problemów składniowych 363
Przypadek 3. Funkcje, które wymagają komunikacji z więcej niż jedną klasą 374
Przewaga funkcji niebędących składowymi 376
Wybór pomiędzy funkcjami zaprzyjaźnionymi a funkcjami składowymi 379
Podsumowanie 380

Rozdział 8. Przeciążanie operatorów 383
Różnica pomiędzy typami standardowymi a typami definiowanymi przez programistę 383
Czym jest operator przeciążony? 386
Przeciążać czy nie przeciążać — plusy i minusy przeciążania operatorów 388
Bardziej eleganckie abstrakcyjne typy danych 388
Zagmatwane przeciążanie operatorów 389
Brak zrozumienia zasad pierwszeństwa i łączności 389
Operatory przeciążone w języku C++ 392
Inne zastosowanie operatorów ++ oraz -- 397
Operator indeksowania — operator [ ] 399
Rzecz bardziej wyrafinowana — operator dostępu do składowych, czyli –> 405
Operatory jako funkcje składowe lub jako funkcje niebędące składowymi 414
Operatory będące funkcjami składowymi 415
Operatory implementowane w postaci funkcji niebędących składowymi 416
Dlaczego potrzebujemy konwersji? 420
Funkcje konwersji 421
Interakcje pomiędzy konstruktorami konwertującymi a funkcjami konwersji 424
Eliminacja potrzeby tworzenia obiektów tymczasowych 428
Zwracanie wyników z funkcji operatorowych 430
Operator przypisania 435
Podsumowanie 435

Rozdział 9. Typy generyczne 437
Problem powtarzania kodu 437
Eleganckie rozwiązanie — programowanie generyczne 444
Podstawy typu generycznego (klasy generycznej) 446
Co się dzieje podczas konkretyzacji nowej klasy szablonowej w języku C++? 448
Typy generyczne a powielanie kodu 453
Kontrakt pomiędzy implementatorem klasy generycznej a jej klientami 454
Czy można to uznać za „dobry projekt”? 459
Operatory a funkcje składowe w implementacji klas generycznych 462
Rozwiązanie alternatywne — specjalizacja klas generycznych 464
Specjalizacje szablonów 464
Specjalizacja funkcji składowej szablonu 465
Inne rozwiązanie — wydzielenie operacji porównania obiektów 467
A jeśli nie można zdefiniować specjalizacji określonej funkcji składowej szablonu? 469
Specjalizacja klas szablonowych 470
Koncepcja funkcji generycznych 472

Konkretyzacja klas szablonowych i funkcji składowych w języku C++ ... 476
Typy generyczne a kontrola typów ... 482
Generyczność z ograniczeniami i bez ograniczeń ... 484
Ograniczenia parametrów szablonu w języku C++ ... 488
Konkretne typy jako parametry szablonu w języku C++ ... 488
Wartości domyślne parametrów szablonu ... 490
Nakładanie ograniczeń na parametry szablonu w języku C++ ... 491
Klasy generyczne a eksport selektywny ... 494
Dziedziczenie a klasy generyczne ... 497
Polimorfizm a klasy generyczne ... 501
Przydatne zastosowania dziedziczenia wraz z klasami generycznymi ... 504
Podejście typu singleton ... 504
Ogólna technika kontrolowania tworzenia obiektów ... 506
Implementacja wskaźników zliczanych ... 508
Minimalizowanie powielania kodu w przypadku obiektów szablonowych ... 517
Zapotrzebowanie programu na pamięć ... 519
Sposoby redukcji rozmiaru kodu szablonowego ... 519
Klasy szablonowe a ochrona kodu źródłowego ... 531
Klasy szablonowe w bibliotekach współdzielonych (dynamicznych) ... 531
Klasy szablonowe w bibliotekach współdzielonych — problem wielokrotnych egzemplarzy ... 534
Eliminacja wielokrotnie występujących egzemplarzy w bibliotekach współdzielonych ... 535
Konsolidacja z istniejącymi bibliotekami współdzielonymi ... 537
Klasy kontenerowe ... 538
Porównanie klas generycznych i mechanizmu dziedziczenia ... 539
Podsumowanie ... 540

Rozdział 10. W oczekiwaniu nieoczekiwanego ... 543

Po co obsługiwać sytuacje wyjątkowe? ... 543
Dlaczego kody błędów nie są wystarczające? ... 544
Jakie mamy możliwości? ... 545
Model obsługi wyjątków języka C++ ... 546
Działanie mechanizmu wyjątków w języku C++ ... 547
Wpływ bloku kodu chronionego na program ... 549
Wpływ zgłaszanego wyjątku na program ... 550
Dynamiczny łańcuch wywołań ... 551
Obsługa wielu wyjątków ... 554
Odpowiedzialność klauzuli catch ... 555
Model wyjątków w języku Eiffel ... 556
Porównanie modeli wyjątków w językach C++ i Eiffel ... 560
Efektywne stosowanie wyjątków w C++ ... 563
Tworzenie hierarchii wyjątków ... 563
Kolejność klauzul przechwytujących ... 566
Odporność funkcji na wyjątki ... 568
Zagadnienia projektowe dotyczące obsługi wyjątków ... 570
Kiedy zgłaszać wyjątki? ... 570
Strategie skutecznego zarządzania błędami w projekcie ... 573
Funkcje nie są zaporami ... 574
Projektowanie hierarchii wyjątków ... 575
Zarządzanie zasobami w środowisku wyjątków ... 578
Automatyzacja zarządzania zasobami ... 579
Uogólnianie rozwiązania zarządzania zasobami ... 582

Wyjątki a konstruktory ... 584
Zwracanie zabezpieczonych zasobów z funkcji ... 585
Klasa pomocna w zarządzaniu tablicami obiektów ... 588
Koszt automatycznego zarządzania zasobami ... 594
Częściowa skuteczność konstruktora ... 594
Bezpieczne tworzenie tablic z wykorzystaniem wyjątków ... 595
Podsumowanie ... 601

Część II Tworzenie zaawansowanych aplikacji obiektowych ... 603

Rozdział 11. Poznawanie abstrakcji danych ... 605
Ukrywanie szczegółów implementacji abstrakcji ... 605
Zalety używania uchwytów ... 609
Wady używania uchwytów ... 609
Wskaźniki w roli danych składowych (opóźnianie ewaluacji) ... 613
Kontrolowanie sposobu tworzenia obiektów ... 616
Wymuszanie stosowania operatora new() ... 616
Blokowanie stosowania operatora new() ... 619
Używanie wskaźników i referencji zamiast obiektów zagnieżdżonych ... 619
Wady stosowania dużych tablic jako zmiennych automatycznych (lub jako danych składowych) ... 620
Używanie tablic obiektów oraz tablic wskaźników do obiektów ... 621
Obiekty zamiast wskaźników typów prostych (w danych składowych i wartościach zwracanych przez funkcje składowe) ... 624
Zgodność z językiem C ... 627
Rozmiar obiektu a efektywność kodu — szukanie implementacji alternatywnych ... 629
Unikanie obiektów tymczasowych ... 632
Inicjalizowanie obiektów konstruktorem kopiującym ... 633
Efektywne używanie obiektów proxy (lub obiektów zastępczych) ... 634
Obiekty proxy ułatwiające bezpieczne współdzielenie obiektów ... 634
Łatwość używania obiektów proxy ... 637
Obiekty proxy będące zastępcami obiektów zdalnych ... 638
Inteligentne obiekty proxy dostarczające dodatkowej funkcjonalności ... 638
Klasy proxy do rozwiązywania problemów składniowych i semantycznych ... 639
Technika ogólnego indeksowego obiektu proxy ... 642
Używanie prostych abstrakcji do budowy bardziej złożonych ... 644
Abstrakcje dające użytkownikom wybór sposobu stosowania klas ... 645
Podsumowanie ... 647

Rozdział 12. Efektywne korzystanie z dziedziczenia ... 649
Wykorzystanie dziedziczenia w eleganckich implementacjach menu ... 649
Obsługa menu różnych typów ... 655
Hermetyzacja szczegółów fazy tworzenia obiektu ... 656
Koncepcja konstruktorów wirtualnych ... 658
Funkcje wirtualne i niewirtualne w kontroli protokołu ... 661
Koncepcja podwójnego rozgłaszania wywołań ... 670
Projektowanie i implementacja klas kontenerów ... 677
Projektowanie obsługi różnych kontenerów ... 679
Implementacja klas kontenerów homogenicznych z użyciem programowania generycznego ... 691
Cele projektowe ... 692
Zalety kontenerów homogenicznych opartych na szablonach ... 697
Wady kontenerów szablonowych ... 698
Implementacja heterogenicznych kontenerów na podstawie homogenicznych kontenerów wskaźników ... 698

Przeglądanie kontenerów ... 701
Iteratory pasywne ... 702
Iteratory aktywne ... 705
Iteratory jako obiekty ... 708
Zarządzanie kolekcjami i iteratorami z punktu widzenia użytkownika ... 715
Metoda 1. Utworzenie iteratora w kontenerze i przekazanie go użytkownikowi ... 715
Metoda 2. Zwracanie kopii kontenera, aby użytkownik swobodnie nim manipulował ... 716
Biblioteka standardowa języka C++ (STL) ... 718
Kontenery STL ... 719
Iteratory ... 720
Algorytmy biblioteki STL ... 721
Podsumowanie ... 723
Kod implementujący kontener TArray ... 724

Rozdział 13. Wewnętrzny model obiektowy w C++ ... 735
Efektywna implementacja ... 735
Reprezentacja obiektów w C++ ... 735
Obiekty klas pozbawionych funkcji wirtualnych ... 736
Metody ... 736
Statyczne dane składowe ... 738
Konstruktory ... 738
Klasy z metodami wirtualnymi ... 739
Rozmieszczenie wskaźnika tablicy vtbl ... 740
Współużytkowanie tablic funkcji wirtualnych przez biblioteki współdzielone ... 743
Metody wirtualne a dziedziczenie wielokrotne (bez wirtualnych klas bazowych) ... 743
Wirtualne klasy bazowe ... 749
Odwołania do składowych z wirtualnymi klasami bazowymi ... 750
Wirtualne klasy bazowe z metodami wirtualnymi ... 752
Implementacja mechanizmu RTTI (informacja o typie w czasie wykonania) ... 754
Programowanie obiektowe a programowanie z wykorzystaniem obiektów ... 756
Referencje, wskaźniki i wartości ... 756
Przypisania referencji i wskaźników ... 756
Konstruktor kopiujący ... 758
Zadania konstruktora ... 758
Zadania konstruktora kopiującego ... 761
Optymalizacja przekazywania i zwracania obiektów przez wartość ... 763
Przekazywanie przez wartość ... 763
Zwracanie przez wartość ... 765
Inicjalizacja czasu wykonania ... 768
Podsumowanie ... 768

Dodatki ... 769

Dodatek A Przestrzenie nazw ... 771
Deklaracja using ... 772
Dyrektywa using ... 773
Przestrzeń nazw std ... 773

Dodatek B Bibliografia i lista lektur zalecanych ... 775

Skorowidz ... 779

Rozdział 1.

Czym jest programowanie obiektowe?

Ostatnio niemal wszyscy pracujący w przemyśle związanym z produkcją oprogramowania zachwycają się paradygmatem programowania obiektowego. Nawet menadżerowie, dyrektorzy i pracownicy marketingu zakochali się w technologii obiektowej. Można odnieść wrażenie, że nie istnieje nic lepszego od podejścia obiektowego. Wygląda na to, że programy obiektowe stały się *Świętym Graalem*, którego wszyscy poszukują. Niektórzy mogą się zastanawiać, czym jest ten nowy paradygmat i czym różni się od standardów, które obowiązywały przez dziesięciolecia. Programiści mogą poczuć się odstawieni na boczny tor wraz z całym doświadczeniem i umiejętnościami, które w obliczu obiektowego potwora nie są już potrzebne. Jeśli weźmie się to wszystko pod uwagę, warto zapoznać się z odpowiedziami na poniższe pytania:

- O co chodzi w tym całym konstruowaniu oprogramowania obiektowego?
- Jakie są z tego korzyści?
- W czym różni się ono od tradycyjnego podejścia do konstruowania oprogramowania?
- Jaki wpływ ma programowanie obiektowe na tradycyjne umiejętności związane z konstruowaniem oprogramowania?
- Jak można stać się „obiektowym”?

Pochodzenie

Programiści konstruowali oprogramowanie od dziesięcioleci i zwykle stosowali bardzo małe programy do wielkich systemów. Używali przy tym rozmaitych języków programowania, takich jak Algol, COBOL, Lisp, C czy Pascal. *Bardzo mały program* oznacza na przykład rozwiązanie problemu wież Hanoi, pasjansa, prostą implementację sortowania quicksort i inne programy, które pisze się w ramach zadania domowego na

kursie lub po prostu dla nauki. Programy te nie mają żadnej wartości komercyjnej. Pomagają za to w nauce nowego pojęcia lub języka. Dla odmiany termin *wielkie systemy* odnosi się do systemów oprogramowania, które rozwiązują poważne problemy, takie jak kontrola rezerw czy zarządzanie finansami. Przy ich implementacji współpracują zespoły programistów i projektantów. Następnie producenci puszczają je w obieg. Nauka wyniesiona z projektowania i implementowania małych programów pomaga w rozwiązywaniu dużych problemów. Na co dzień używa się systemów zaimplementowanych za pomocą wszystkich podanych języków. Na podstawie doświadczeń z tymi językami i systemami zebrana została także szeroka wiedza. Dlaczego więc powinno zmieniać się paradygmat programowania? Czytaj dalej. Odpowiedź stanie się oczywista po przeczytaniu kolejnych kilku akapitów.

Przykład programowania proceduralnego

Kiedy programista staje przed koniecznością rozwiązania jakiegoś problemu, który podaje mu się na przykład w postaci ustnego lub pisemnego opisu, w jaki sposób powinien zabrać się za projektowanie i implementowanie rozwiązania przy użyciu języka takiego jak C? Programista rozbija problem na mniejsze, bardziej znaczące fragmenty, zwane modułami. Następnie projektuje struktury do przechowywania danych i implementuje potrzebne funkcje, nazywane także procedurami, które działają na tych danych. Funkcje mogą zmieniać wartość struktur danych, zapisywać je do pliku lub wyświetlać. Cała wiedza systemu wbudowana jest w zestaw funkcji. Główny nacisk położony jest na te funkcje, ponieważ bez nich nie można wykonać żadnej pożytecznej operacji. Styl programowania, gdy w centrum zainteresowania są funkcje, nazywa się **programowaniem proceduralnym**. Nazwa wzięła się stąd, że najważniejszym elementem są tu procedury. Ich waga bierze się z myślenia o problemie w kategoriach jego funkcji. Taki sposób myślenia nazywa się funkcjonalną analizą problemu.



Podział na procedury, funkcje i podprogramy nie istnieje w językach C i C++. Jednak w językach Pascal, Modula-2 i Eiffel używane jest pojęcie **funkcji**, które określa podprogram zwracający obliczoną wartość, oraz pojęcie **procedury**, które określa podprogram pobierający argumenty i wykonujący operacje, ale niezwracający wartości. W tej książce pojęcia *procedura* i *funkcja* używane są zamiennie. Mają one identyczne znaczenie. Języki takie jak Algol, Fortran, Pascal i C nazywane są językami proceduralnymi.

Jednak kiedy przyjrzymy się dokładniej implementacjom, możemy zauważyć, że najważniejsze informacje znajdują się w strukturach danych. Tym, co nas najbardziej interesuje, są wartości przechowywane w tych strukturach — te, które w programowaniu proceduralnym zostały odstawione na boczny tor. Procedury, które implementujemy, to proste narzędzia do operowania strukturami danych. Bez tych struktur procedury nie miałyby na czym działać. Większość czasu spędzamy na projektowaniu procedur i poświęcamy im całą uwagę mimo tego, że najważniejsze elementy znajdują się gdzie indziej. Ponadto kod w tych procedurach nigdy nie zmienia się w czasie działania programu. To dane w strukturach danych podlegają takim zmianom. Pod tym względem procedury są dość nieciekawe, ponieważ zachowują się statycznie. Wyobraź sobie na przykład system bankowy, w którym klienci mogą otwierać różne rodzaje kont, takie

jak rachunki oszczędnościowe, rachunki bieżące i kredytowe. Klienci mogą lokować pieniądze na koncie, wycofywać wkład i przesyłać pieniądze między rachunkami. Jeśli system implementowany jest w języku C, prawdopodobnie zobaczysz taki zestaw procedur[1]:

```
typedef unsigned long AccountNum;
typedef int bool;

bool MakeDeposit(AccountNum whichAccount, float amount);
float WithDraw(AccountNum whichAccount, float howmuch);
bool Transfer(AccountNum from, AccountNum to, float howmuch);
```

`AccountNum` może być dowolną dodatnią liczbą całkowitą. Możemy stworzyć prostą strukturę danych do zarządzania rachunkami:

```
// Prosty typ strukturalny do przechowywania informacji o koncie
struct Account {
   char* name;          /* nazwa właściciela konta */
   AccountNum accountId;
   float balance;
   float interestYTD;    /* wysokość odsetek */
   char accountType;     /* Oszczędnościowy, Bieżący, Kredytowy itd. */
   /* i wiele innych szczegółów */
};
```

Możemy utworzyć rachunek klienta, tworząc egzemplarz struktury `Account` za pomocą funkcji:

```
AccountNum CreateNewAccount(const char name[], char typeOfAccount);
```

Reprezentacja konta bankowego

Funkcja ta zwraca numer rachunku nowego konta. Kiedy popatrzymy na to rozwiązanie, staje się jasne, że klient może być dużo bardziej zainteresowany tym, ile pieniędzy jest na jego koncie i jakie są zyski z odsetek, niż funkcjami, które pozwalają na wpłacanie i wypłacanie pieniędzy. Klient postrzega rachunek jako bezpieczne miejsce dla swoich ciężko zarobionych pieniędzy. Nie obchodzi go, jak dokonywane są wypłaty lub wpłaty. Jedyne, czego potrzebuje, to prosty sposób na dokonywanie tych żmudnych operacji. Ale programista skoncentrował się na pisaniu tych właśnie nieciekawych fragmentów kodu w postaci funkcji `MakeDeposit` i innych funkcji, a do zarządzania danymi utworzył skromne struktury. Mówiąc inaczej, skupił uwagę na tych elementach, które mało klienta obchodzą. Co więcej, nie istnieje specjalny związek między klientem a jego rachunkami bankowymi. Klient jest traktowany jak seria liter i cyfr, a operacje na danych odbywają się bez uwzględniania informacji o właścicielu konta i o stanie rachunku. Dla funkcji rachunek bankowy to po prostu numer — numer konta.

[1] Składnia nie jest w tym momencie istotna. Ważne, aby podkreślić użycie procedur i argumentów. Każdy język posiada własną składnię do ich definiowania.

Bezpieczeństwo konta bankowego

Następnie możemy zauważyć, że dowolny inny program lub programista może utworzyć rachunek i dokonywać na nim operacji. Ponieważ rachunek jest przechowywany jako fragment danych, każdy, kto uzyska dostęp do rekordu z kontem, może zmienić jego wartość — nawet nielegalnie — i wypłacić pieniądze. Jest to konsekwencją faktu, że rachunek reprezentowany jest jako ciąg liter i cyfr. Nie istnieje zabezpieczenie różnych wartości w nim przechowywanych. Nie istnieje reguła, która określa, że rachunek może zostać zmieniony tylko przez zaufanego pracownika banku — a nawet jeśli istnieje, kto wymusi jej stosowanie? Języki takie, jak C lub Pascal, nie mogą tego zrobić, ponieważ nie widzą różnicy między rachunkiem bankowym a zwykłą liczbą całkowitą.

Jeśli program ma wyświetlać dane z rachunku klientowi, dodawana jest nowa funkcja:

```
PrintAccount(Account thisAccount);
```

Funkcja ta wyświetli dane rachunku. Ale powinna ona mieć informacje o tym, jakiego rodzaju jest to rachunek — oszczędnościowy, bieżący czy inny. To proste — wystarczy sprawdzić wartość pola `accountType`. Niech w banku istnieją wymienione wcześniej trzy rodzaje kont — bieżące, oszczędnościowe i kredytowe. Funkcja `PrintAccount()` zna te typy, ponieważ zostały na sztywno umieszczone w jej kodzie. Jak dotąd wszystko idzie dobrze. Teraz załóżmy, że dodawany jest nowy typ konta — konto emerytalne `retirement_account`. Co się stanie, jeśli przekażemy do funkcji `PrintAccount()` rachunek typu „konto emerytalne”? Funkcja nie zadziała. Możemy zobaczyć komunikat o błędzie:

```
"Nieznany typ rachunku - wyświetlenie niemożliwe"
```

Albo nawet gorzej:

```
"Niepoprawny typ rachunku - skontaktuj się z administratorem"
```

Dzieje się tak, ponieważ typ rachunku jest informacją na sztywno umieszczoną w systemie i nie może zostać zmieniony, dopóki nie zmieni się kod źródłowy, a program nie zostanie ponownie skompilowany i skonsolidowany. Jeśli więc dodamy nowy typ rachunku, trzeba zmienić wszystkie funkcje, w których występuje ta informacja, i przejść przez etap kompiluj-konsoliduj-testuj. Proces ten jest żmudny i narażony na błędy. Jak więc można sobie poradzić z tym problemem? Odpowiedź znów wynika z faktu, że funkcje i struktury danych traktowane są jako dwa odrębne byty, które nie mają ze sobą nic wspólnego. Z tego powodu zmiany w strukturze danych nie są łatwo rozumiane przez funkcje. Pożądaną sytuacją jest istnienie systemu, w którym dodanie nowego typu rachunku nie wpływa na inne typy i nie powoduje potrzeby globalnej rekompilacji kodu. Rozszerzanie istniejącej już implementacji jest bardzo kłopotliwe.

O wszystkie te problemy można obwiniać złe rozłożenie akcentów w pierwotnym rozwiązaniu. Programista jest w nim skoncentrowany bardziej na funkcjach, o których błędnie myśli, że są najważniejsze. Z kolei prawie zupełnie lekceważone są struktury danych, czyli elementy najbardziej istotne dla klienta. Mówiąc inaczej, główny nacisk kładzie się na to, **jak** coś zrobić, podczas gdy trzeba skupić się na tym, **co** robić. To właśnie miejsce, w którym programowanie obiektowe różni się od proceduralnego.

Rozwiązywanie problemów w programowaniu obiektowym

Jeśli rozwiązujemy problem rachunku bankowego za pomocą technik programowania obiektowego, trzeba zająć się głównie samym rachunkiem. Na pierwszym miejscu programista skupia się na tym, czego oczekuje klient, co jest dla niego ważne. **W programowaniu obiektowym najważniejsze są dane, na których operujemy, a nie procedury, które implementują te operacje**. Programista stara się zrozumieć, co użytkownik chciałby robić z danymi, a następnie implementuje te istotne operacje. Dodatkowo dane i operacje nie są traktowane jako odrębne elementy, jak miało to miejsce wcześniej. Postrzegane są jako integralna całość. Dane udostępniane są wraz z zestawem operacji, które pozwalają użytkownikowi pracować z nimi. Jednocześnie dane same w sobie nie są dostępne dla żadnego zewnętrznego programu lub procedury. Jedynym sposobem ich zmiany jest korzystanie z operacji dostarczonych specjalnie dla tego celu. Możliwe operacje na rachunku dokonywane są w imieniu użytkownika. Można teraz powiedzieć, że rachunek bankowy jest klasą i da się utworzyć wiele jego egzemplarzy, a każdy z nich będzie obiektem. Dlatego **programowanie obiektowe jest metodą programowania, w której programy składają się ze współpracujących ze sobą obiektów będących egzemplarzami klas. Klasy najczęściej związane są ze sobą poprzez dziedziczenie**[2].

Kluczowe są tu pojęcia **klasa** i **obiekt**. Klasa to byt, który zbiera wspólne właściwości grupy obiektów. Obiekt to egzemplarz klasy. Wszystkie obiekty jednej klasy posiadają tę samą strukturę i funkcje zgodne z deklaracją tej klasy. Można patrzeć na klasę jak na foremkę do ciastek — wówczas ciasteczka byłyby egzemplarzami tworzonymi za jej pomocą. Analogia ta jest bardzo zgrubna, ale przypomina proces tworzenia obiektów na podstawie klasy. Foremka określa rozmiar i kształt ciastka — choć już nie jego smak. Podobnie klasa określa rozmiar i funkcje obiektów. W prawdziwym obiektowym rozwiązaniu problemu wszystkie elementy są obiektami utworzonymi na podstawie klasy[3]. W programowaniu obiektowym myślimy w kategoriach klas, obiektów i relacji między nimi[4]. Shlaer i Mellor opisują różnice między programowaniem proceduralnym i obiektowym. **Analiza funkcjonalna odnosi się do trzech elementów projektowania systemu, w kolejności: algorytmy, przepływ danych, opis danych. Programowanie obiektowe odwraca tę kolejność.**

Wracamy do problemu rachunku bankowego. Teraz w centrum zainteresowania jest rachunek bankowy, który staje się klasą. Poniżej znajduje się szkielet klasy `BankAccount` w języku C++. Podobnie jak w poprzednim przypadku, nie przejmuj się składnią.

```
class BankAccount {
   public:
      // Wiele szczegółów pominięto dla czytelności
```

[2] Pełny opis dziedziczenia znajduje się w dalszych rozdziałach.

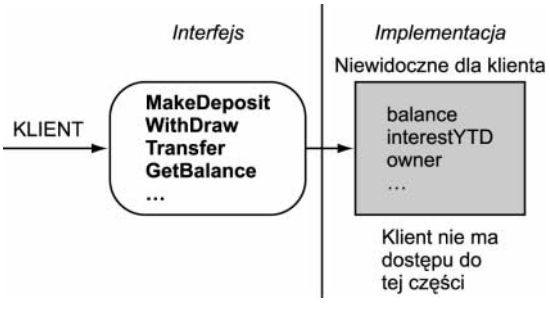
[3] Sytuacja taka prawie zachodzi w języku C++, jeśli założymy, że funkcja `main()` jest głównym obiektem, od którego wszystko się zaczyna.

[4] Istnieją stosunkowo nieznane języki delegowania (ang. *delegation languages*), w których obiekty mogą tworzyć inne klasy.

```
        void MakeDeposit(float amount);
        float WithDraw();
        bool Transfer(BankAccount& to, float amount);
        float GetBalance() const;
    private:
        // Implementacja danych do użytku w obrębie klasy BankAccount
        float balance;
        float interestYTD;
        char* owner;
        int account_number;
};
```

Dla klienta istotna jest część publiczna — zawiera ona możliwe do wykonania operacje na klasie. Na rysunku 1.1 zaznaczono je pogrubioną czcionką. Deklaracje w regionie prywatnym nie są dostępne dla klienta. Są one przeznaczone do wyłącznego wewnętrznego użytku operacji. Na przykład operacja `MakeDeposit` może być zaimplementowana w podany sposób:

```
// Implementacja operacji klasy BankAccount
void BankAccount::MakeDeposit(float amount)
{
    if (amount > 0.0)
        balance = balance + amount;
}
```



Rysunek 1.1. *Podział klasy na interfejs i implementację*

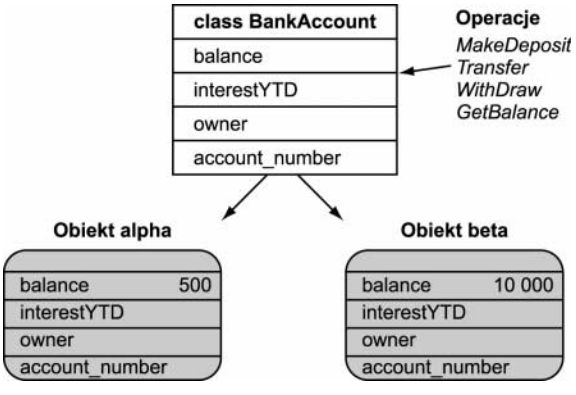
Dostęp do prywatnej składowej `balance` i innych składowych prywatnych istnieje wyłącznie z poziomu operacji zadeklarowanych w klasie `BankAccount`. Składowe te nie są dostępne do normalnego użytku z zewnątrz klasy. Takie prywatne dane określamy jako dane **ukryte**. Ten sposób ukrywania danych wewnątrz klasy nazywany jest *hermetyzacją danych*. Więcej szczegółów na temat tego zagadnienia znajdziesz w rozdziale 2.

Wprowadzenie do modelu obiektowego

Jedną z przeszkód w zrozumieniu paradygmatu programowania obiektowego jest brak zrozumienia pojęć *klasa* i *obiekt*. Aby skutecznie programować w paradygmacie obiektowym, należy dogłębnie zapoznać się z tymi pojęciami.

Podstawowym bytem w programowaniu obiektowym jest klasa. Wróćmy do przykładu rachunku bankowego — każdy obiekt typu `BankAccount` posiada tę samą strukturę

i funkcje. Dlatego na wszystkich obiektach `BankAccount` można wykonać operację `MakeDeposit` i inne operacje tej klasy. Wszystkie obiekty posiadają też własny zestaw prywatnych danych (`balance`, `account_number` itd.), różnią się natomiast wartościami, które są w nich przechowywane. Na przykład obiekt `alpha` typu `BankAccount` może posiadać wartość `500` w polu `balance`, a obiekt `beta` może w tym samym polu posiadać wartość `10000`. Zostało to pokazane na rysunku 1.2.



Rysunek 1.2. *Przykładowe obiekty klasy BankAccount*

Obiekty są egzemplarzami („ciasteczkami") klasy. Klasa widoczna jest tylko w kodzie źródłowym programu, podczas gdy obiekty uczestniczą w działaniu programu. To one zajmują miejsce w pamięci komputera. Można „dotknąć" i „poczuć" obiekt. Proces tworzenia obiektu jest także nazywany **konkretyzacją**.

W programowaniu proceduralnym zawsze mówi się o wywoływaniu procedur. Używane są wyrażenia typu „wywoływanie procedury X" czy „wywoływanie Y", gdzie X i Y to nazwy procedur. W programowaniu obiektowym nigdy nie mówimy: „Wywołaj X i wywołaj Y". Zamiast tego używamy wyrażeń „Wykonaj X dla pewnego obiektu". Jeśli istnieje obiekt `myAccount` klasy `BankAccount`, można powiedzieć: „Wykonaj procedurę `MakeDeposit` dla obiektu `myAccount`". Procedury są wykonywane (wywoływane) dla obiektów. Bez obiektu po prostu nie możemy wywołać procedury. Nigdy nie używa się tu sformułowania „Zrób to". Poprawnym zwrotem jest „Zrób to dla tego obiektu". Każda operacja dotyczy obiektu i każda spełnia jakąś użyteczną rolę, na przykład obsługuje dokonywanie wpłaty. Możliwość zmiany cudzych danych jest tu całkowicie wykluczona.

Kiedy programista patrzy na strukturę danych w modelu programowania proceduralnego, ciężko mu zorientować się, co dana struktura robi i do czego może być używana. Zdarza się, że nie jest jasne, po co ona w ogóle istnieje. Cel, zastosowanie i ograniczenia struktury danych są trudne do zrozumienia, ponieważ nie jest ona inteligentna. Cała inteligencja potrzebna do poprawnego jej stosowania znajduje się gdzieś w zestawie funkcji. Nie istnieją niezależne struktury danych. W programowaniu obiektowym nie ma tego typu problemów, ponieważ wszystko, co można zrobić z klasą, jest jasno określone w niej samej, w postaci operacji publicznych. W rzeczywistości nie istnieją struktury danych niezależnie dostępne dla klienta. Wszystko, z czym on się kontaktuje, to jasno zdefiniowany zbiór operacji. Struktury danych ukryte są wewnątrz klasy stanowią jedną całość z operacjami, które można na nich wykonać.

Terminologia

Przyszła pora, aby przedstawić poprawną terminologię dotyczącą operacji i danych w klasie (patrz też tabela 1.1).

Tabela 1.1. *Pojęcia używane w różnych językach obiektowych*

Definicja	Język C++	Język Smalltalk	Język Eiffel
Opis grupy podobnych obiektów	Klasa	Klasa	Klasa
Zestaw prywatnych danych i funkcji	Obiekt	Obiekt lub egzemplarz	Obiekt lub egzemplarz
Ogólna klasa wyższego poziomu	Klasa bazowa	Nadklasa	Klasa macierzysta
Klasa niższego poziomu	Klasa pochodna	Podklasa	Klasa potomna
Sposób wielokrotnego użycia projektu i kodu	Dziedziczenie	Dziedziczenie	Dziedziczenie
Wywołanie niezależne od typu	Polimorfizm	Polimorfizm	Polimorfizm
Żądanie wykonania przez obiekt jednej z jego operacji	Wywołanie metody	Wywołanie komunikatu	Wywołanie operacji
Sposób wykonania jednej z operacji	Implementacja metody	Metoda	Podprogram — procedura lub funkcja
Prywatne dane	Dane składowe	Zmienne egzemplarza	Pola, atrybuty

C++ W języku C++ funkcje klasy nazywane są *metodami*, a zmienne nazywane są *danymi składowymi*. Funkcje zachowują się jak normalne funkcje, ale należą do określonej klasy, dlatego są jej metodami. Tak samo zmienne składają się na dane, które należą do obiektu i dlatego są danymi składowymi.

EIFFEL Języki Eiffel i Ada nazywają funkcje *operacjami*, a zmienne *atrybutami*. W języku Eiffel rozróżnia się zmienne w klasach (nazywane *atrybutami*) oraz zmienne w obiektach (zwane *polami*). Funkcje są operacjami, ponieważ używane są przez klientów do manipulowania obiektem. Zmienne nazywane są polami z powodu podobieństwa obiektów do *rekordów* w języku Pascal.

SMALLTALK W języku Smalltalk funkcje są nazywane *komunikatami*, a zmienne *zmiennymi egzemplarza*.

Poznajemy komunikaty, metody i zmienne egzemplarza

Dowolny użytkownik (zwykle inny program lub nawet inna klasa) klasy nazywany jest *klientem* tej klasy. Klient używa metod (komunikatów) do wykonywania potrzebnych operacji na obiekcie. Jak się później przekonamy, klienty mogą tworzyć obiekty i używać ich; mogą też utworzyć nową klasę, *dziedzicząc* po istniejącej klasie.

SMALLTALK W języku Smalltalk wywoływanie funkcji interfejsu — komunikatów — postrzegane jest jako *wysyłanie komunikatów do obiektu*. Podejście takie jest uzasadnione. Wysyłamy komunikat `MakeDeposit` do obiektu typu `BankAccount`, prosząc o przyjęcie wpłaty. To powoduje uruchomienie w obiekcie określonej *metody*. W ten sposób *odpowiada* on na wysłany mu komunikat. Komunikat jest tylko nazwą widzianą przez klienta, zaś ona z kolei związana jest z poprawną implementacją komunikatu w obiekcie, który go otrzymuje. Powiązanie nazwy z implementacją zachodzi podczas wykonywania programu, a poprawna implementacja to po prostu metoda. Każdy egzemplarz klasy, czyli każdy obiekt, zawiera odrębną kopię zmiennych egzemplarza, co przedstawiono na rysunku 1.2.



Pojęcia *metoda*, *funkcja składowa*, *komunikat* i *operacja* używane są w tej książce zamiennie. Ich znaczenie jest praktycznie takie samo.

Z czego składa się obiekt?

Każdy utworzony obiekt otrzymuje własną kopię atrybutów. Atrybuty, za wyjątkiem statycznych, nie są dzielone. W języku C++ tylko atrybuty statyczne mogą być dzielone pomiędzy wszystkie obiekty klasy w trakcie działania programu. Język Smalltalk także obsługuje atrybuty dzielone[5]. A co z metodami? Czy każdy obiekt otrzymuje własną kopię kodu każdej metody? Zdecydowanie nie. Każdy obiekt może stosować wszystkie metody deklarowane w klasie, ale nie zawiera kopii kodu implementacji. W niektórych przypadkach istnieje tylko jedna kopia kodu implementacji metody w jednym programie. Niezależnie od ilości obiektów utworzonych w programie kod metod nie jest powielany. Kod współdzielony jest pomiędzy wszystkimi obiektami klasy. Dla ułatwienia można wyobrazić sobie, że kod implementacji znajduje się w bibliotece. Wiele implementacji może go ulepszać i mogą one przechowywać tylko jedną kopię kodu implementacji w całym systemie. Zwykle odbywa się to za pomocą bibliotek współdzielonych. Wszystkie te szczegóły są w dużym stopniu zależne od systemu operacyjnego. Na przykład możemy użyć klasy `Card` do opisu karty w talii kart:

```
enum Suit { Clubs, Diamond, Heart, Spade, Unknown };
enum Rank { Two, Three, Four, Five, Six, Seven, Eight,
                        Nine, Ten, Jack, Queen, King, Ace, Invalid };
enum Color { Red, Black };

class Card {
   public:
      void FaceUp();       // Pokazuje kartę rysunkiem do góry
      void FaceDown();     // Pokazuje kartę rysunkiem do dołu
            // Wiele innych metod
      Card(Rank r, Suit s);    // Tworzy kartę na podstawie argumentów
   private:
      Rank cardRank;
      Suit cardSuit;
      Color cardColor;
};
```

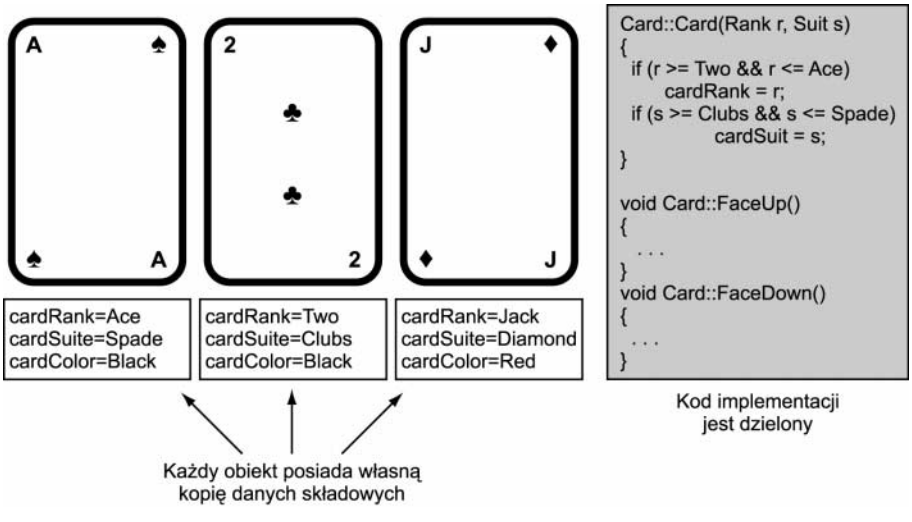
[5] Szerzej składowe dzielone opisane zostały przy okazji wyjaśniania analizy mechanizmu współdzielenia przez obiekty danej klasy.

Jeśli utworzy się talię 52 kart, powstaną 52 obiekty klasy `Card`. Każdy obiekt będzie miał własną kopię danych składowych `cardRank`, `cardSuit` i `cardColor`.

```
Card myDeck[52];    // Utwórz standardową talię 52 kart
```

Można manipulować każdą kartą w talii oddzielnie. Możliwe jest też utworzenie kilku obiektów typu `Card`. Efekty pokazano na rysunku 1.3.

```
Card spade_Ace(Ace, Spade);    // As pik
Card clubs_2(Two, Clubs);      // Dwójka trefl
Card diamond_Jack(Jack, Diamond)    // Walet karo
```



Rysunek 1.3. *Egzemplarze klasy Card*

Tworzenie obiektów

Kiedy klasa jest już zaprojektowana i zaimplementowana, programista, który chce używać obiektu tej klasy, musi utworzyć w kodzie jej egzemplarz. W poszczególnych językach odbywa się to w różny sposób. W języku C++ tworzenie obiektu wygląda jak prosta deklaracja, na przykład:

```
BankAccount myAccount;
```

W języku Smalltalk programista musi wysłać do klasy predefiniowany komunikat `new`, aby utworzyć jej nowy obiekt. Wygląda to następująco:

```
BankAccount new.
```

W języku Eiffel do tworzenia obiektu służy predefiniowana operacja `make`:

```
MyAccount : BankAccount; -- to tylko deklaracja, obiekt nie jest tworzony
MyAccount.!!make; -- w tym miejscu tworzony jest obiekt
```

Co można uznać za klasę?

Łatwo mówić o klasach i obiektach na podstawie prostych przykładów, jednak celem jest określenie odpowiedniego zestawu klas dla danego problemu. Należy zrozumieć, co reprezentuje klasa i kiedy problem powinien zostać przekształcony w klasę, a kiedy wystarczą proste dane. Według naszej definicji klasa powinna reprezentować wspólne właściwości grupy obiektów. Jak wspólne muszą być te *wspólne* właściwości? Kiedy powinniśmy stwierdzić, że coś jest klasą, a nie tylko obiektem składowym innej klasy? Są to istotne pytania, które pojawiają się, gdy poznajemy programowanie obiektowe.

Kiedy zapada decyzja, że powstanie klasa, należy przede wszystkim zadać sobie pytanie: „Czy na pewno istnieje potrzeba tworzenia więcej niż jednego egzemplarza klasy?". Jeśli odpowiedź jest pozytywna, wszystko jest na dobrej drodze — przynajmniej na początku. Jeśli nie ma żadnych różnic między egzemplarzami klasy — każdy egzemplarz jest odpowiednikiem wszystkich pozostałych i zachowują się one tak samo — prawdopodobnie utworzyliśmy klasę, która powinna być wartością. Kiedy na przykład utworzymy klasę `Kolor`, która dotyczy kwiatów, a potem okaże się, że wszystko, co ona reprezentuje, to zwykła liczba, nasze działanie nie ma sensu. Jeśli jednak działamy w systemach graficznych, które wykonują złożone obliczenia kolorów, `Kolor` powinno być klasą, ponieważ posiada wiele elementów składowych i odcieni opartych na kolorach podstawowych — czerwonym, zielonym i niebieskim. Taka klasa `Kolor` może być kontrolowana za pomocą operowania jej składowymi. Uwidacznia to, że `Kolor` w systemach graficznych nie jest prostą wartością — do tej klasy dodano wiele możliwości.

Następny przykład to klasa `Adres`. Może ona określać adres domowy, który traktujemy jako ciąg znaków. Ale adres może też być klasą w systemie poczty elektronicznej i zawierać nazwy domen, serwerów i inne składowe. Taki adres może też określać sposób przekazywania komunikatów. Nie należy traktować adresu w takiej formie jako ciągu znaków, ale jak prawdziwą klasę.

Warto pamiętać, że za pierwszym razem prawie zawsze zdarzają się jakieś błędy. Elementy, które określone zostały jako klasy przy pierwszym podejściu do projektowania, mogą stać się prostymi danymi w kolejnym podejściu, i na odwrót. Rozwiązania problemów zmieniają się z czasem. Bardzo rzadko zdarza się, że ostateczne rozwiązanie jest takie samo jak początkowy projekt.

Wszystko to podkreśla jedną ważną właściwość klasy. Nie jest ona tylko pojemnikiem na dane, które mogą być modyfikowane przez funkcje. Klasa to coś, co udostępnia klientowi uproszczony obraz złożonego bytu i pozwala mu operować na swoim obiekcie w celu osiągnięcia pożądanych rezultatów. Klasa określa też, jak czynności są wykonywane, i jasno stwierdza, co można zrobić z obiektem. Jest ona czymś znacznie większym niż tylko sumą swych części. Wracając do przykładu z klasą `Kolor` — w systemie graficznym składowe (czerwony, zielony i niebieski) nie znaczą zbyt wiele dla klienta. Ważne jest, jak klasa łączy te składowe i wyświetla je. Podobnie rachunek bankowy to nie tylko znaki i liczby. Klasa rachunku bankowego widziana jest jako coś, co pozwala klientom zarządzać ich pieniędzmi łatwo i bezpiecznie.

Co nie jest klasą?

Bardzo ważne jest też, aby wiedzieć, kiedy dane i funkcje nie powinny tworzyć klasy. Klasa to nie kilka zgrupowanych funkcji. Taki efekt występuje, kiedy przekształcimy w nią moduł lub prosty plik nagłówkowy języka C. Wystarczy wziąć wszystkie funkcje modułu, zrobić z nich publiczne metody i klasa gotowa! W rzeczywistości klasa to nie zbiór funkcji. To coś o wiele więcej.

Wyobraź sobie moduł, który implementuje zestaw funkcji matematycznych, takich jak pierwiastek kwadratowy, potęga, odwrotność. Można spróbować — niepoprawnie — uczynić z takiego modułu klasę o nazwie `MathHelper`.

```
class MathHelper{
   public:
      double Sqrt(double aNumber);
      double Power(double aNumber, int raiseTo);
      double Inverse(double aNumber);
   private:
      // dowolne prywatne dane, prawdopodobnie nic!
};
```

Problem polega na tym, że nie ma żadnych danych do zarządzania w obrębie klasy `MathHelper`. Klient wywołuje jedną z metod i przekazuje argumenty. Metody wykonują na argumentach potrzebne operacje. Ale w trakcie obliczeń metody nie potrzebują pomocy od klasy. Nie są w niej przechowywane żadne dane, które byłyby przydatne metodom. Dzieje się tak w przypadku wszystkich metod tej klasy. Funkcje są po prostu niepotrzebnie zgrupowane, podczas gdy nie mają ze sobą nic wspólnego. Wszystko, co zawiera taka klasa, to zbiór funkcji bez żadnych danych. Lepszym podejściem byłoby utworzenie klasy `Number` i dostarczenie dla niej operacji.

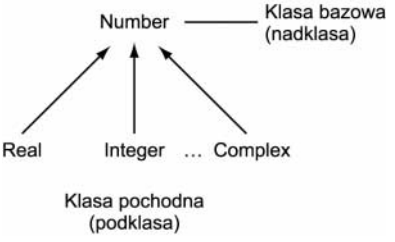
```
class Number {
   public:
      Number Sqrt();
      Number Power(Number raiseTo);
      Number Inverse();
      Number Absolute(); // wartość absolutna liczby
   private:
      // tutaj znajdzie się wewnętrzna reprezentacja do przechowywanie liczby
};
```

W tym przypadku klasa `Number` kontroluje wewnętrzną reprezentację liczby. Ponieważ klient nie ma żadnej wiedzy o tej reprezentacji, logiczne jest, że klasa dostarcza potrzebnych operacji.

Jeśli pójdziemy krok dalej w projekcie, możemy wyobrazić sobie hierarchię dziedziczenia, w której reprezentowane są różne typy liczb — rzeczywiste, całkowite, zespolone. Mogą one stanowić klasy pochodne od klasy `Number`, co pokazano na rysunku 1.4.

Dziedziczenie opisane zostało w rozdziałach 5. i 6.

Podobnie struktura z języka C nie może stać się od razu klasą. Nie można zmienić struktury w klasę i przekształcić wszystkich danych w prywatne dane składowe oraz



Rysunek 1.4. *Hierarchia klasy Number*

dodać zestawu funkcji do pobierania i ustawiania ich wartości. To także nie będzie klasa. Klasa to nie grupa funkcji, która pozwala klientowi pobierać i nadawać wartość. Hermetyzacja danych ukrywa dane w klasie i udostępnia wyższy poziom abstrakcji za pomocą metod. Jeśli tylko dostarczysz funkcji do zapisu i odczytu danych w strukturze, praca z kodem nie stanie się łatwiejsza. Klasy, w których jednymi funkcjami są akcesory `Get` i `Set`, są przeważnie przykładem słabo zaprojektowanych klas[6].

Cel klasy

Klasa projektowana jest w jednym, jasno określonym celu. Jej funkcjonalność nigdy nie powinna poza niego wychodzić. Dobrze zaprojektowana klasa powinna być łatwa do zrozumienia i prosta w użyciu. Jej zastosowanie powinno być oczywiste dla klienta. Nie możemy utworzyć pojedynczej klasy `Kolor` do reprezentowania koloru kwiatów i kolorów w systemie graficznym. Wymagają one zupełnie czego innego. Programista nie powinien dodawać do klasy metod, które są zupełnie niezwiązane z jej zastosowaniem, tylko po to, żeby tylko zaspokoić oczekiwania grupy klientów. Każda klasa zaprojektowana jest w jednym celu w określonej dziedzinie. Zastanów się nad klasą `Person` (osoba):

```
// Przykład złego projektowania
class Person {
   public:
      Person(/*jakieś argumenty*/);
      ~Person();
      char* GetName() const;
      char* GetAddress() const;
      unsigned long GetBankAccountNumber() const;
   //...
};
```

Metoda `GetBankAccountNumber` jest zupełnie niezwiązana z resztą klasy. Skąd wiadomo, że osoba zawsze będzie miała rachunek? W tym celu trzeba najpierw się dowiedzieć, czy jest ona klientem banku.

Czasem projektanci po prostu wrzucają gdzieś metody, ponieważ nie pasują one nigdzie indziej, albo próbują znaleźć zastosowanie dla metody w ograniczonym środowisku. Narusza to zasadę abstrahowania danych i wprowadza klientów w zakłopotanie. Projektant klasy miesza abstrakcję osoby z abstrakcją klienta banku.

[6] Jedna funkcja — `GetXXX()` — do pobierania wartości i druga — `SetXXX()` — do nadawania jej.

Dobrą miarą projektu klasy jest to, czy projektant potrafi w jednym prostym zdaniu zawrzeć jej cel. Jeśli do jej opisu potrzeba kilku akapitów, na pewno zawiera ona zbyt wiele dostępnych funkcji i należy rozbić ją na mniejsze klasy. Inną miarą jest ilość metod w klasie. Dobrze zaprojektowana nie powinna zawierać więcej niż 15 – 25 metod.

Kiedy klient patrzy na klasę, powinien móc łatwo określić, jakie możliwości mu ona udostępnia. Klasa powinna przedstawiać zwięzły i jasny obraz swoich możliwości i ograniczeń. Jeśli po zetknięciu się z nią klient nie jest pewny, do czego ona służy, projekt jest słaby i prawdopodobnie niepoprawny. Czas wtedy zacząć wszystko od nowa.

Klasa, a dokładniej obiekt, ma do wypełnienia określone obowiązki. Kiedy klasa jest projektowana, projektant podejmuje zobowiązanie odnośnie jej możliwości. Takie zobowiązanie zapewnia, że klasa odpowiedzialna jest za zarządzanie określonymi szczegółami, aby klient nie musiał się już o nie troszczyć. Klasa `BankAccount` odpowiedzialna jest za utrzymywanie poprawnego bilansu w wyniku dokonywanych wpłat i wypłat. Jej zobowiązanie polega na tym, że kiedy klient używa jej obiektu do zarządzania rachunkiem i stosuje się do jej publicznych metod, poprawność i bezpieczeństwo rachunku są gwarantowane. Podczas wypełniania zobowiązań klasa `BankAccount` może współpracować z innymi klasami. Ta współpraca może, choć nie musi interesować klientów, ale projektant klasy powinien dobrze rozumieć te zagadnienia. Współpraca między klasami opisana została w dalszej części rozdziału. Więcej o dobrych i złych klasach dowiesz się z dalszej części rozdziału.

Więcej o obiektach

Jak już wspominałem, obiekt to egzemplarz klasy. Obiekt powołuje ją do życia. Wszystkie możliwości klasy uwidaczniają się poprzez utworzenie obiektu i manipulowanie nim. Obiekt jest inteligentny. Wie, co potrafi zrobić, a czego nie. Posiada też wiedzę i zasoby do zmieniania i przechowywania swoich atrybutów.

Następnym logicznym pytaniem jest pytanie o to, co odróżnia klasę od obiektu. Obiekt to „żywy" byt ze stanami i zachowaniem. Zachowanie wszystkich obiektów klasy określane jest właśnie przez nią, a stan przechowywany jest przez poszczególne obiekty. Te dwa elementy — stan i zachowanie — to bardzo proste określenia, które mają swoje głębokie znaczenie w odniesieniu do obiektów.

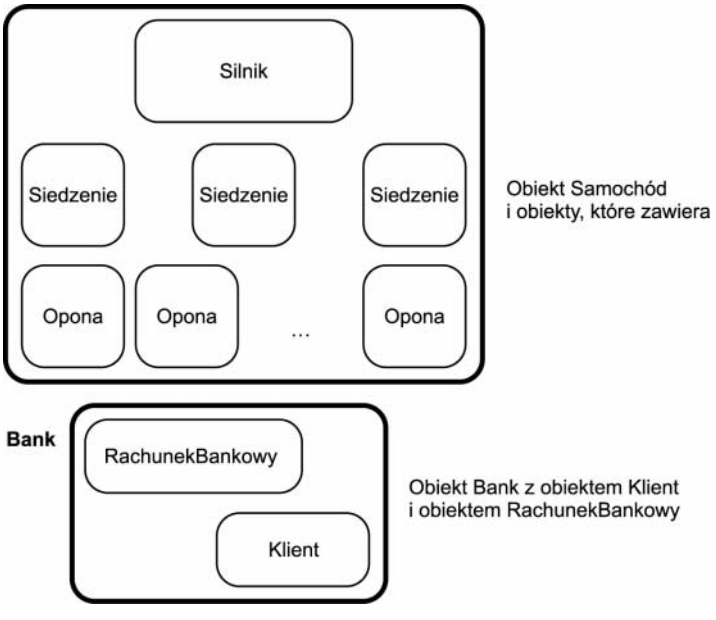
Stan obiektu

Wrócimy teraz znów do problemu rachunku bankowego. Każdy obiekt klasy `BankAccount` posiada składową `balance`. Jeśli założysz, że użytkownik nie może wybrać z rachunku więcej, niż na nim ma, stan składowej `balance` nie może spaść poniżej zera. Stanie się to wtedy znaną właściwością każdego obiektu typu „rachunek bankowy". Nie musimy sprawdzać stanu obiektu, aby się o tym upewnić. Jest to *statyczna* właściwość każdego obiektu tej klasy.

Jednak w dowolnej chwili istnienia obiektu typu `BankAccount` ilość pieniędzy na koncie przechowywana jest przez składową `balance`. Wartość tej składowej zmienia się w momencie, kiedy dokonywane są wpłaty, wypłaty i przelewy. Stan rachunku jest *dynamicznie* zmieniającą się wartością. Innymi słowy, jest to podlegająca zmianom wartość składowa `balance`. Stan obiektu jest sumą wszystkich statycznych i dynamicznych wartości jego właściwości. Właściwość jest niepowtarzalną cechą obiektu. Możemy powiedzieć, że każdy samochód posiada właściwość w postaci numeru rejestracyjnego. Podobnie każda osoba posiada imię i nazwisko, choć ta właściwość nie jest do końca unikatowa.

Stanu obiektu nie muszą określać proste typy danych. I przeważnie nie określają. Wiele obiektów zawiera inne obiekty jako części swego stanu. Na przykład obiekt `Samochód` może zawierać obiekt `Silnik` jako część swojego stanu, a obiekt `Bank` może zawierać obiekty `RachunekBankowy` i `Klient` (patrz rysunek 1.5)[7].

Rysunek 1.5. *Stan obiektu*



Dlaczego stan obiektu jest ważny?

Można się zastanawiać, dlaczego tak wiele uwagi poświęcamy tym danym w obiekcie, które są ukryte. To, jak obiekt reaguje na nasze polecenia i co robi z innymi obiektami, zależy od jego stanu. Wynik działania metody bezpośrednio zależy od stanu obiektu. Kiedy komunikat `WithDraw` (wypłać) zostanie wysłany do obiektu `BankAccount`, zajdzie sekwencja następujących zdarzeń:

[7] Dokładniejsza analiza stanów obiektu znajduje się w rozdziale 2.

1. Sprawdzenie, czy rachunek należy do osoby, która wywołała operację.
2. Jeśli żądana ilość pieniędzy przekracza stan konta, użytkownikowi pokazuje się komunikat o błędzie.
3. W innym przypadku stan konta zmniejszany jest o pobraną kwotę.

Na wszystkich tych etapach niezbędne są informacje o stanie obiektu. Wszystkie metody zakładają, że jest on poprawny. Jeśli stan obiektu jest niepoprawny z bliżej nieokreślonych przyczyn, to jego zachowanie jest nieprzewidywalne.

Inny dobry przykład stanu to stan pralki. Kiedy wciśniemy przycisk START, odpowiednie systemy sprawdzają, czy pojemnik jest zamknięty. Sprawdzanie odbywa się za pomocą właściwości obiektu. Pralka nie zacznie pracować, jeśli drzwiczki są otwarte. Większość pralek posiada czujnik w postaci prostego przełącznika — sprawdza on stan drzwiczek. Użytkownik nie powinien bezpośrednio manipulować przełącznikiem. Czujnik powinien być ukryty. Jeśli jakiś nadmiernie entuzjastyczny użytkownik uzyska dostęp do czujnika, może oszukać pralkę, symulując zamknięcie drzwiczek. Teraz, po wciśnięciu przycisku START pralka zacznie działać i prawdopodobnie porozrzuca ubrania oraz proszek i rozchlapie wodę po całej łazience. Takie nieprzewidziane — lub przewidziane, ale niepożądane — zachowanie jest bezpośrednim rezultatem niekontrolowanej zmiany w stanie obiektu. Dobra implementacja klasy nie powinna pozawalać klientom na bezpośredni dostęp do tego stanu. Stany powinny być modyfikowane tylko poprzez metody, a klient powinien używać wyłącznie tych metod do manipulowania obiektami. Zastanów się, co się stanie, kiedy użytkownik uruchomi kuchenkę mikrofalową z otwartymi drzwiczkami. Nic się nie stanie, ponieważ implementacja kuchenki mikrofalowej sprawdza przed włączeniem magnetronu (urządzenia, które generuje mikrofale), czy drzwiczki są zamknięte. Działa to w bardzo podobny sposób do czujnika w pralce. Jeśli klient uzyska dostęp do przełącznika i zmieni jego stan, kuchenka uruchomi magnetron nawet przy otwartych drzwiczkach, ponieważ czujniki sygnalizują, że zostały one zamknięte. Może to spowodować nieodwracalne zmiany w stanie zdrowia osoby stojącej w pobliżu kuchenki.

Kto kontroluje stan obiektu?

Z poprzedniego paragrafu można łatwo wywnioskować, że stan obiektu modyfikowany jest poprzez metody. Jednak nie wszystkie metody modyfikują stan obiektu; niektóre z nich mogą po prostu używać wartości stanu. Należy do nich na przykład metoda `GetBalance` (podaj stan konta) w klasie `BankAccount`. Każda metoda w klasie zakłada pewne ograniczenia stanu obiektu. Założenia te mogą być udokumentowane, mogą też zostać umieszczone w kodzie. Klasa zakłada, że z zewnątrz nie zostanie dokonana żadna modyfikacja stanu obiektu — wszystkie takie zmiany będą odbywać się za pomocą metod. Metody posiadają konieczną wiedzę na temat tego, co powinny robić z wartościami właściwości obiektu. **Stan obiektu kontrolują metody**[8]. Warto zauważyć, że metody wykonywane są po wywołaniu ich przez klienta. Klient wywołuje operację,

[8] Pewnym wyjątkiem jest funkcja zaprzyjaźniona w języku C++. Można jednak traktować ją, przynajmniej na potrzeby tego opisu, jako część klasy.

wysyłając komunikat do obiektu, a operacja wykonuje właściwe czynności. Zwykle metoda nie wykonuje niczego samodzielnie. Musi zostać wywołana przez klienta[9].

Metody znają także ograniczenia stanów obiektu. W przykładzie dotyczącym rachunku bankowego metody posiadają ograniczenie dotyczące tego, że stan konta nie może spaść poniżej zera. Każda metoda wymusza przestrzeganie tego ograniczenia. Jeśli stan obiektu jest modyfikowany inaczej niż za pomocą metody, zachowanie obiektu staje się nieprzewidywalne. Dzieje się tak w przykładzie z pralką. W innym przykładzie ktoś mógłby nielegalnie usunąć wszystkie pieniądze z cudzego konta. Wtedy właściciel konta nie mógłby wypłacić żadnych pieniędzy, ponieważ stan rachunku wskazywałby zero. Język programowania nie może powstrzymać takich rozmyślnych włamań, ale może zapobiegać przypadkowym błędom. Zabezpieczenie polega na uczynieniu atrybutu `balance` fragmentem prywatnej części implementacji. Wszystko, co nie powinno być dostępne dla normalnego klienta, powinno być *ukrywane* w prywatnym regionie klasy. Takie zabiegi zwane są *hermetyzacją danych*.

Kiedy tworzona jest klasa, zawsze jakiś jej fragment pozostaje ukryty. Są to pewne informacje, których obiekty potrzebują do poprawnego działania. Klasa bez żadnych danych składowych jest wynikiem złego projektowania, ponieważ sygnalizuje obiekt bez stanów. Więcej o hermetyzacji danych można przeczytać w rozdziale 2.

Zachowanie obiektu

Klienci używają metod do wykonywania pożądanych czynności. Obiekt zachowuje się w pewien sposób w odpowiedzi na komunikat, który wywołuje klient. *Zachowanie to sposób, w jaki obiekt działa i reaguje na komunikaty*. Komunikat może spowodować zmianę stanu. Może też spowodować wysłanie kolejnych komunikatów do innych obiektów. Kiedy klient wysyła komunikat do obiektu, może też wysłać inny komunikat do innego obiektu, aby sfinalizować operację. Na przykład obiekt `BankAccount`, kiedy otrzyma komunikat `WithDraw`, może wysłać komunikat do obiektu — nazwijmy go `tl` — klasy `TransactionLogger`, aby zapisać aktualną transakcję. Obiekt `tl` może w odpowiedzi wysłać komunikat do bazy danych, aby zapisała tę transakcję. Wynika z tego, że wysłanie komunikatu do obiektu może zapoczątkować łańcuch kolejnych komunikatów przesyłanych do innych obiektów. Możliwe też, że kolejny komunikat zostanie wysłany do oryginalnego obiektu, nawet rekurencyjnie. Zachowanie to widziane z zewnątrz działanie obiektu w odpowiedzi na komunikat. To właśnie widzi klient.

Niektóre komunikaty mogą spowodować zmianę stanu obiektu, a niektóre nie. W językach C++ i Eiffel można łatwo wyróżnić komunikaty, które nie powodują żadnych zmian w obiekcie[10]. To, co robi komunikat, powinno być jasno udokumentowane dla każdej metody w klasie. Zadaniem projektanta jest dostarczenie klientowi możliwie wielu informacji przy jednoczesnym zatajeniu szczegółów implementacji.

[9] Nie jest to prawdą w przypadku obiektów służących do obsługi przerwań i podobnych zadań.

[10] W języku C++ takie funkcje to metody stałe. Bardziej szczegółowo opisano je w następnym rozdziale.

Etapy w konstruowaniu oprogramowania obiektowego

Analiza obiektowa

Realizacji projektu programu nie rozpoczynamy od zestawu klas lub obiektów. Zaczynamy z prostym opisem problemu, najczęściej niekompletnym. To punkt startowy do dalszej pracy. Teraz należy zaplanować zestaw klas. Warto pamiętać, że większość niebanalnych problemów wymaga nie jednej klasy, a całego ich zestawu. Dopiero wtedy rozwiązanie będzie poprawne. Projektowane klasy powinny komunikować się i współpracować ze sobą, aby osiągnąć zamierzone rezultaty. Jak odkryć lub nawet wymyślić klasy, które potrzebne są do rozwiązania? Jest to prawdopodobnie najtrudniejszy etap konstruowania oprogramowania obiektowego i zajmuje on sporo czasu. Trudność tego zadania bierze się stąd, że opis problemu jest najczęściej niepełny lub dotyczy raczej implementacji niż samego problemu.

Analiza obiektowa zawiera analizę problemu pod kątem klas i obiektów, które znajdują się w dziedzinie problemu. Polega to głównie na tworzeniu modelu dziedziny problemu. Wspomniane klasy nie będą bezpośrednio użyte w końcowej implementacji. Problem jest analizowany, a wymagania jasno doprecyzowane. Powstaje pełny opis tego, czym ma być rozwiązanie w kategoriach klas i obiektów z dziedziny problemu użytkownika. Taki opis problemu korzysta z klas i obiektów, których znaczenie użytkownik potrafi zrozumieć. Te klasy i obiekty pochodzą bezpośrednio z interesującej użytkownika dziedziny.

Następne pytanie dotyczy tego, jakie jest znaczenie pojęcia dziedziny problemu? Każdy problem wymaga rozwiązania, które wiąże się z jedną lub z większą ilością dziedzin. Aby utworzyć rozwiązanie, musimy polegać na wiedzy i doświadczeniu osób pracujących w danej dziedzinie. Na przykład przy projektowaniu rozwiązań do zarządzania bankiem potrzebujemy pomocy osób, które na co dzień zarządzają bankiem. Można powiedzieć, że problem należy do dziedziny bankowej lub finansowej. Mówiąc prosto, dziedzina problemu to obszar lub sektor, do którego problem należy. Dziedziny problemu — lub po prostu dziedziny — mogą być bardzo różne: projektowanie łazienek, produkcja rowerów, projektowanie maszynowe, zarządzanie magazynem, symulacje geofizyczne, sieci, interfejs użytkownika, animacja, finanse, automatyzacja biura, przetwarzanie rozproszone, analiza matematyczna, komunikacja między komputerami, zarządzanie bazą danych i wiele innych. Jedna osoba nie jest w stanie zdobyć odpowiedniej wiedzy we wszystkich tych obszarach w celu tworzenia dobrych projektów rozwiązań w tak różnych dziedzinach.

Nawet jeśli ktoś jest ekspertem w obszarze konstruowania oprogramowania obiektowego, znalezienie rozwiązania dla problemu zarządzania bankiem będzie wymagać pomocy ze strony osób bezpośrednio zaangażowanych w pracę z systemem bankowym. Osoby te znają wymagania, jakie stawiane są takiemu systemowi. Znają też braki obecnego systemu. Mogą dostarczyć wartościowych informacji o funkcjach, które

pomogłyby im w pracy. Takie doświadczone osoby można nazwać ekspertami w dziedzinie. Najczęściej ekspert w danej dziedzinie nie ma pojęcia o programowaniu.

Przy wsparciu ekspertów grupa projektantów specjalizujących się w konstruowaniu oprogramowania obiektowego może zacząć tworzyć rozwiązanie problemu. Bez ich pomocy nie można utworzyć dobrego projektu obiektowego. Każdy rzeczywisty problem wymaga bliskiej współpracy i koordynacji między ludźmi na co dzień zajmującymi się pracą w określonej dziedzinie i ekspertami z obszaru programowania obiektowego.

W tym momencie etap analizy obiektowej jeszcze się nie kończy. Jest to dopiero szkielet rozwiązania, który wymaga mnóstwa pracy, zanim będzie można zacząć implementację. Ale jest to już dobry początek. Warto zauważyć, że na etapie analizy obiektowej uwaga skupia się głównie na klasach używanych w obszarze problemu, a nie na implementacji. Szczegóły implementacji opracowywane są dopiero na etapie projektowania obiektowego.

Zwróć uwagę, że odkrywanie klas jest dla człowieka naturalną rzeczą. W życiu codziennie mamy styczność z klasami, takimi jak poczta, skrzynka na listy, menadżer, kwiatek, gazeta, kuchenka mikrofalowa, odtwarzacz CD, ojciec, matka, samochód itd. Są to obiekty, z którymi stykamy się na każdym kroku. Wiemy także, co można z nimi zrobić i do czego służą. Konstruowanie oprogramowania składa się z tworzenia modelu tych obiektów, które znamy w dziedzinie problemu. To, co już wiemy o prawdziwych obiektach, przetwarzane jest na logiczną wersję rozwiązania problemu. Jeśli istnieje problem zarządzania firmą, pracownicy firmy przedstawiani są jako obiekty. Podobnie ich wypłaty przedstawiane są jako system płac w postaci klas i obiektów. Taki rodzaj symulacji rzeczywistego świata w postaci rozwiązania programowego ułatwia rozwiązywanie problemu, ponieważ pozwala dobrze zrozumieć, jakie klasy i obiekty występują w danej sytuacji. Pomaga to w rozumieniu złożoności problemu i radzeniu sobie z nią. Znane są też ograniczenia obiektów w rzeczywistym świecie. Nie można poprosić listonosza, aby doręczył kwiatki wujkowi. Nie można też odtwarzać płyt w kuchence mikrofalowej. Znane są możliwości i ograniczenia obiektów, które występują w problemie. Wiedza ta jest wykorzystywana w procesie tworzenia rozwiązania. W pewnych sytuacjach wykorzystanie jej jest proste, na przykład w przypadku systemu zarządzania firmą lub systemu zarządzania kontami bankowymi. W innych sytuacjach, szczególnie dotyczących specyficznych problemów oprogramowania, określanie obiektów jest dużo trudniejsze, ponieważ przeważnie nie da się ich zidentyfikować w rzeczywistym świecie. Za przykład niech posłuży określanie odpowiedniego zestawu klas dla systemu zarządzania przerwaniami. Jest to trudne, ponieważ podobne obiekty nie są znane. Trzeba wtedy dogłębnie przeanalizować problem przed rozpoczęciem tworzenia rozwiązania. Podobnie trudne okazują się problemy dotyczące przetwarzania transakcji.

Projektowanie obiektowe

Ten etap następuje po analizie obiektowej i nadaje treść utworzonemu w poprzedniej fazie szkieletowi rozwiązania. Rozparcelowujemy problem na klasy i obiekty, które zostaną wykorzystane w implementacji. Określamy jasne zależności między klasami, powstaje opis dopasowania obiektów do modelu systemu, tworzymy wytyczne dotyczące tego, jak obiekty powinny być rozmieszczone w przestrzeni adresowej, jak system może

się zachowywać itd. Końcowy rezultat tej fazy jest bardziej przejrzysty i łatwiejszy do zrozumienia. Wciąż nie będzie to pełne rozwiązanie, ponieważ brakuje jeszcze implementacji i tak naprawdę nie da się nic powiedzieć o rzeczywistym zachowaniu rozwiązania.

Odkrywanie nieuchwytnych obiektów

Znajdowanie, a raczej wymyślanie i odkrywanie odpowiedniego zestawu klas do implementacji rozwiązania nie jest prostym zadaniem. Jest to jedna z najtrudniejszych rzeczy. Istnieje wiele porad i uznanych metod, które można wykorzystać podczas wykonywania tego zadania. Można za ich pomocą utworzyć początkowy zestaw klas. Wymienione elementy przeważnie przekształcane są w klasy.

- Ludzie, miejsca i przedmioty
- Zdarzenia — wprowadzane dane, narodziny, śmierć itd.
- Transakcje — przyznanie kredytu, sprzedaż samochodu itd.
- Role spełniane przez osoby — matka, ojciec itd.

Przy rozwiązywaniu łatwiejszych problemów projektanci mogą traktować rzeczowniki jako klasy i czasowniki jako metody tych klas. Ale opis problemu w języku naturalnym może być łatwo przekształcony w ten sposób, że rzeczowniki i czasowniki zamieniają się funkcjami. *Dlatego też sposób ten powinien służyć jedynie do wytworzenia pomysłów klas, które posłużą do późniejszej analizy.*

Zarówno w analizie obiektowej, jak i w projektowaniu obiektowym można wykorzystywać istniejące sposoby notacji. Notacja to określony sposób przedstawiania klas, obiektów i relacji między nimi. Umożliwia ona też opis różnych modelów systemu, na przykład logicznego i fizycznego. Daje nam podstawowe narzędzia, które pomagają w procesie projektowania. Używany zapis dostarcza też wspólnego słownictwa dla zespołu projektowego. Jest to jakby wspólny język, rozumiany przez wszystkich członków zespołu. Poniżej przedstawiono nazwy kilku popularnych sposobów notacji.

1. Notacja Boocha[11]
2. Notacja Rumbaugha[12] (zwana też OMT — ang. *Object Modeling Tool*)
3. Notacja Shleara i Mellora[13]

Notacje 1. i 2. zostały połączone w język UML, który opisany został w rozdziale 2.

Warto zauważyć, że nauczenie się notacji nie czyni z ucznia eksperta w dziedzinie projektowania obiektowego. Notacja to tylko narzędzie do takiego przedstawienia projektu, które pozwoli na zrozumienie go. Pomaga to w procesie projektowania, ale nie może pomóc w wynajdywaniu i implementowaniu klas. Niczym nie można zastąpić wiedzy i doświadczenia zespołu projektowego.

[11] Opisana w książce Grady'ego Boocha, *Object-Oriented Analysis and Design*.

[12] Opisana w książce Rumbaugha, *Object-Oriented Modeling and Design*.

[13] S. Shlaer, i S. Mellor, *Object-Oriented Systems Analysis*, Modeling the World in Data.

Analiza obiektowa i projektowanie obiektowe nie są cechami języka C++[14]. Są one nieodłączne przy rozwiązywaniu dowolnego problemu. Analiza obiektowa i projektowanie obiektowe nie zależą od żadnego języka. Jednak często ułatwia pracę świadomość, jaki język będzie wykorzystywany do implementacji. Na etapie projektowania czasem tworzy się taką relację między zestawem klas, która okazuje się niemożliwa do zastosowania w danym języku. Można na przykład użyć w projekcie dziedziczenia wielobazowego, które nie jest dostępne w języku Smalltalk. Jak wtedy zaimplementować takie rozwiązanie w języku Smalltalk? Choć jest to możliwe, będzie wymagało dużego nakładu pracy. Dlatego warto wiedzieć, jaki język ma służyć do implementacji. Z drugiej strony, nie należy korzystać ze szczegółów składni na etapie projektowania. Projekt powinien być możliwie niezależny od specyficznych elementów języka. Powinna istnieć możliwość implementacji go w dowolnym obiektowym języku programowania.

Programowanie obiektowe

To już ostatni etap procesu konstruowania oprogramowania obiektowego. Koniec etapu projektowania obiektowego jest początkiem etapu programowania obiektowego. W tej fazie tworzony jest rzeczywisty kod w wybranym języku programowania. Jak zostało wcześniej powiedziane, programowanie obiektowe to metoda programowania, w której programy tworzone są ze współpracujących ze sobą obiektów, czyli egzemplarzy klas. Klasy z kolei najczęściej są ze sobą powiązane przez relacje dziedziczenia.

Kluczowe elementy modelu obiektowego

Przy rozwiązywaniu problemu w modelu obiektowym istotne są następujące elementy:

Abstrakcja danych

Hermetyzacja

Hierarchia

Wszystkie te elementy zostały szczegółowo opisane w kolejnych rozdziałach. Poniżej znajduje się tylko ich krótki opis.

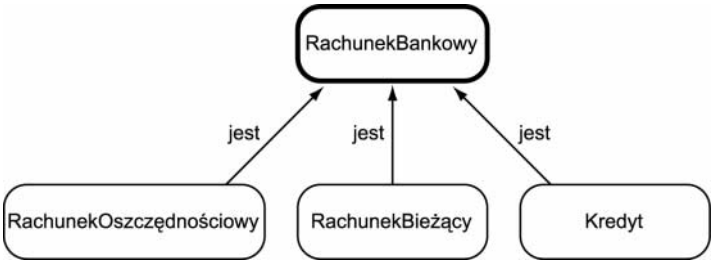
Abstrakcja danych to wynik definiowania klas z naciskiem na podobieństwa między obiektami i ignorowaniem różnic. Nieciekawe i rozpraszające elementy są ukrywane, podczas gdy ważne właściwości klasy są podkreślane. Klasa jest abstrakcją istotnych właściwości. Abstrakcja koncentruje się na zewnętrznym wyglądzie obiektu i oddziela ważne zachowania od wewnętrznych szczegółów implementacji. Dalszy jej opis znajdziesz w rozdziale 2.

[14]Można nawet stwierdzić, że systemy notacji same tworzą wizualny język.

Hermetyzacja, zwana też **ukrywaniem danych**, jest wynikiem ukrywania wewnętrznych szczegółów implementacji. Odgranicza interfejs zewnętrzny od zawiłych szczegółów implementacji. Hermetyzacja i abstrakcja uzupełniają się. Dobrze zaplanowana abstrakcja ukrywa dane, a ukryty byt pomaga jej zachować integralność. Należy pamiętać, że proces abstrahowania poprzedza hermetyzację. Hermetyzacja staje się istotna dopiero w momencie rozpoczęcia implementacji.

Hierarchia to podpora pracy nad abstrakcją. Abstrahowanie jest ważne i użyteczne, ale w większości złożonych problemów często tworzy się zbyt wiele abstrakcji, co utrudnia zrozumienie całego systemu. Hermetyzacja i moduły pomagają załagodzić skutki problemu, ale ciągle może istnieć za dużo abstrakcji. Umysł człowieka jest w stanie zrozumieć abstrakcję tylko do pewnego poziomu. Zbyt wysoki poziom może być dla odbiorcy prawdziwym testem na utrzymywanie informacji w pamięci. Aby uniknąć tego szkodliwego efektu, można zastosować hierarchię abstrakcji. W ten sposób pełna wiedza o systemie nie jest konieczna do zrozumienia jego głównych cech. Nie trzeba chyba wspominać, że tworzenie takiej hierarchii nie jest prostym zadaniem, a implementowanie jej jest jeszcze bardziej skomplikowane.

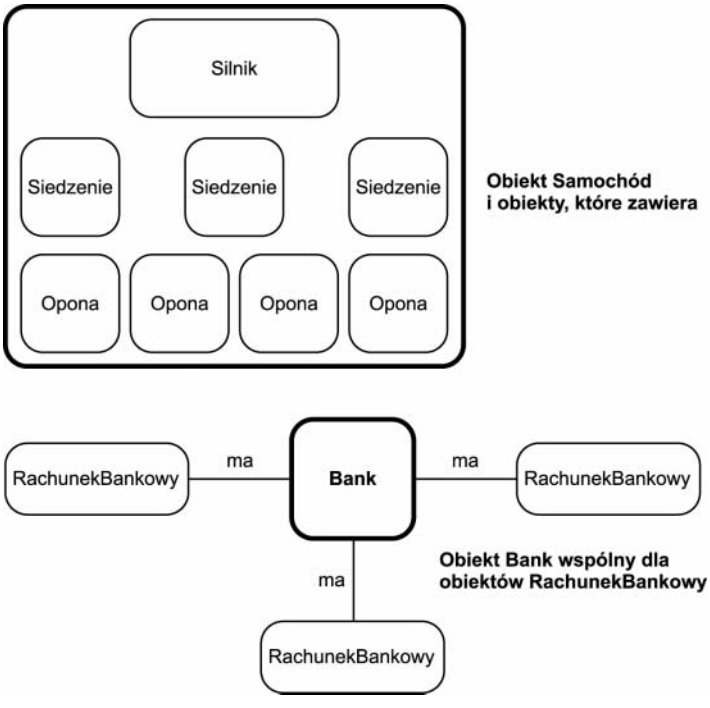
W programowaniu obiektowym istnieją dwa popularne typy hierarchii. Pojęcie dziedziczenia wprowadza hierarchię klas z relacją „jest-czymś" (rysunek 1.6). Relacja agregacji — „ma-coś" — wprowadza stosunek część-całość (rysunek 1.7). Dziedziczenie umożliwia stosowanie relacji ogólne-specyficzne, podczas gdy agregacja służy do przedstawiania relacji związanych z zawieraniem i współdzieleniem.



Rysunek 1.6. *Relacja dziedziczenia (jest-czymś)*

Dziedziczenie, zarówno jednokrotne, jak i wielokrotne, jest jednym z najistotniejszych paradygmatów programowania obiektowego. Duża część wartości programowania obiektowego i wielokrotnego wykorzystania kodu bierze się z dziedziczenia. W ramach przykładu można sobie wyobrazić różne rodzaje rachunków bankowych z klasą `RachunekBankowy` jako klasą bazową i klasami `RachunekOszczędnościowy`, `RachunekBieżący` oraz `RachunekKredytowy` jako klasami pochodnymi. Klient może zrozumieć schemat działania rachunku bankowego, poznając interfejs klasy bazowej `RachunekBankowy`, bez wgłębiania się w działanie różnych rodzajów rachunków.

Przykładem agregacji może być samochód, który składa się z silnika, opon, siedzeń. Można powiedzieć, że klasa `Samochód` posiada `Silnik`, `Koła`, `Siedzenia` i inne cechy. Dla normalnego użytkownika `Samochód` pojawia się jako pojedynczy obiekt, nawet jeśli jego wewnętrzna konstrukcja zawiera wiele innych obiektów. Klienta nie interesuje to, w jaki sposób obiekt `Samochód` kontroluje i zapewnia komunikację wewnętrznych obiektów. Zarządzanie zawieranymi obiektami jest zadaniem metod klasy `Samochód`.



Rysunek 1.7. *Relacja agregacji (ma-coś)*

Inny rodzaj relacji „ma-coś" przedstawia związek obiektu `Bank` z obiektami `RachunekBankowy`. Obiekt `RachunekBankowy` nie posiada obiektu `Bank`. Posiada jedynie odniesienie do niego. Obiekt `RachunekBankowy` współdzieli odniesienie do tego samego obiektu `Bank` z wieloma innymi rachunkami bankowymi. Relacja „ma-coś" nie zawsze oznacza, że jeden obiekt fizycznie zawiera egzemplarz drugiego. Oznacza tylko, że istnieje między klasami jakiś związek.

Warto też zauważyć, że obiekt `RachunekBankowy` może komunikować się z obiektem `Bank` jedynie za pomocą publicznego interfejsu, jak każdy inny klient obiektu `Bank`. Rachunek bankowy nie posiada żadnych specjalnych przywilejów w stosunku do obiektu `Bank`. Istnieje wiele implikacji, które dotyczą obu rodzajów agregacji. Zostaną one opisane w kolejnych rozdziałach.

Agregacja pomaga uprościć interfejs i ułatwia dzielenie obiektów. Dziedziczenie i agregacja opisane są bardziej szczegółowo w kolejnym rozdziale.

Paradygmaty i języki programowania obiektowego

Abstrakcja danych, hermetyzacja i hierarchia to podstawowe pojęcia paradygmatu programowania obiektowego. Nie są to specyficzne cechy żadnego języka. Każdy język, który pozwala na programowanie obiektowe, musi umożliwiać stosowanie tych paradygmatów.

Ponadto osoby poznające te pojęcia nie muszą martwić się o naukę żadnego konkretnego języka. Możliwe jest zrozumienie tych pojęć ogólnie, bez wiedzy o składni języka, która umożliwia ich stosowanie. Kiedy projektant lub programista zrozumie pojęcie, dużo szybciej nauczy się składni w dowolnym języku. Przypomina to naukę jazdy samochodem. Kiedy nauczysz się zasad i sposobów ich stosowania, możesz pojechać, gdziekolwiek chcesz. Jedyną nowość stanowią reguły ruchu drogowego.

Jakie wymagania musi spełniać język obiektowy?

Teraz pora spojrzeć na języki programowania ze względu na udostępnianie cech programowania obiektowego. Co powoduje, że język jest językiem obiektowym?

Każdy język, który nazywamy obiektowym, musi posiadać właściwości ułatwiające projektowanie i implementowanie poniższych elementów.

- Abstrakcja danych
- Hermetyzacja
- Dziedziczenie

Określenie „właściwości ułatwiające" oznacza, że abstrakcja i hermetyzacja powinny powstawać naturalnie, bez wielkiego wysiłku ze strony programisty. Elementy języka powinny czynić elegancką implementację abstrakcji bardzo łatwym zadaniem. Musi też istnieć bardzo prosty sposób na zapewnienie hermetyzacji danych. Język powinien być zaprojektowany z uwzględnieniem zasad programowania obiektowego.

Dziedziczenie to kolejna kluczowa cecha programowania obiektowego. Bez możliwości dziedziczenia język nie może zostać nazwany obiektowym językiem programowania. Niektóre języki umożliwiają abstrakcję danych i hermetyzację, ale nie udostępniają żadnych sposobów dziedziczenia. Nie można nazwać ich „obiektowymi". Mają one swoją własną nazwę — mówi się o nich, że „opierają się na obiektach", ponieważ możliwe jest implementowanie obiektów, ale nie jest możliwe rozszerzanie ich za pomocą dziedziczenia. Do tej kategorii należą języki Ada i Modula-2. Warto zauważyć, że nawet w języku C da się zastosować abstrakcję danych, a nawet, w pewnym stopniu, hermetyzację. Jednak takie abstrahowanie i hermetyzacja wymagają wiele wysiłku od programisty. Nie są one naturalne dla języka. Możliwe jest programowanie obiektowe w języku C, a nawet w asemblerze. Powstaje tylko pytanie o praktyczność takiego działania. Jeśli chce się korzystać z paradygmatów programowania obiektowego, dużo lepiej będzie użyć języka, który bezpośrednio umożliwia ich stosowanie i zapobiega błędom. Należy pamiętać, że każdy język obiektowy jest jednocześnie językiem opierającym się na obiektach.

Języki takie, jak Smalltalk, C++, Eiffel lub Object Pascal są prawdziwymi językami obiektowymi, ponieważ udostępniają eleganckie możliwości używania abstrakcji danych, hermetyzacji i dziedziczenia. Dziedziczenie jest kluczową cechą, która odróżnia języki opierające się na obiektach od w pełni obiektowych języków (patrz tabela 1.2).

Tabela 1.2. *Rodzaje języków programowania*

Języki proceduralne	Języki opierające się na obiektach	Języki obiektowe	Języki deklaratywne
PL/1, Algol, COBOL, Fortran, Pascal, C i inne	Ada, Modula-2 i inne	Smalltalk, Eiffel, Objective-C, C++, Java i inne	LISP, Prolog i inne

Zalety modelu obiektowego

Łatwo wychwalać programowanie obiektowe, ale to, co rzeczywiście się liczy, to korzyści, jakie wynikają ze stosowania tego paradygmatu. Łatwo zrozumieć zalety modelu obiektowego i efektywnie ich używać.

1. Model obiektowy zachęca do tworzenia systemów, które mogą podlegać zmianom. Dzięki temu systemy są stabilne i elastyczne. W celu dostarczenia nowych funkcji nie trzeba wyrzucać całego systemu lub projektować go od nowa. Dziedziczenie powoduje, że jest to łatwe zadanie[15]. Można używać istniejącego oprogramowania, podczas gdy dodawane są do niego nowe możliwości.
2. Myślenie w kategoriach obiektów i klas jest dużo łatwiejsze dla człowieka dzięki regularnej styczności z wieloma obiektami. Nawet ludzie, którzy nie mają doświadczenia w pracy z komputerem, łatwiej rozumieją model obiektowy od tradycyjnego.
3. Model obiektowy wymusza zdyscyplinowane programowanie przez oddzielenie klienta i programisty, przez co zapobiega przypadkowym szkodom. Hermetyzacja pozwala uniknąć takich problemów.
4. Wielokrotne wykorzystanie prostych klas, takich jak data, czas lub punkt, pozwala lepiej wykorzystać kod i uniknąć jego replikacji. Proste klasy mogą także służyć do tworzenia bardziej złożonych, przez co powtórne wykorzystanie kodu jest jeszcze bardziej efektywne.
5. Model obiektowy zachęca do wielokrotnego używania istniejącego oprogramowania i projektów. Klasy mogą być rozszerzane przez dziedziczenie, co pozwala na ponowne użycie kodu na poziomie klasy. Dużo większe korzyści daje wykorzystanie całego istniejącego systemu klas do rozwiązania problemu. Klienci powinni mieć możliwość łatwego poprawienia, wielokrotnego użycia, przystosowania do własnych potrzeb i rozszerzenia systemu klas. Prosty przykład stanowi jednolity schemat urządzeń klasy Mass Storage Device. Pozwala on na łatwe tworzenie nowych sterowników. Podobnie wygląda sytuacja w przypadku schematów zarządzania różnymi protokołami przesyłu danych, takimi jak TCP/IP, X.25 lub XNS. Częste jest również wielokrotne wykorzystanie hierarchii klas. Wiele struktur danych, których używa się regularnie, na przykład list, kolejek i tablic haszujących, dostarczanych jest w formie hierarchii klas. Taki poziom wielokrotnego wykorzystania kodu jest bardzo trudny do osiągnięcia w programowaniu proceduralnym.

[15] Jest to prawdą tylko wtedy, kiedy ogólny projekt systemu jest poprawny.



Z całego tego opisu nie powinieneś wyciągać wniosku, że wraz z modelem obiektowym programowanie proceduralne stało się niepotrzebne. Wprost przeciwnie. Model obiektowy wprowadza nowe elementy, dzięki którym można wykorzystać wszelkie doświadczenia nabyte w pracy z innymi modelami programowania. Duża wiedza nabyta przez dziesięciolecia pracy z innymi modelami powinna ułatwić tworzenie lepszych programów. Model obiektowy daje możliwości, które nie istnieją w innych modelach. Z drugiej strony, im ktoś jest lepszy w kompozycji funkcjonalnej i programowaniu w języku C, tym trudniej będzie mu w pełni zrozumieć i stosować technologię obiektową oraz język C++. Taka zmiana paradygmatu to poważne wyzwanie.

Podsumowanie

Model obiektowy koncentruje się na klasach i obiektach.

W programowaniu obiektowym ważniejsze jest, **co** robić, niż **jak** to robić.

Obiekty posiadają stany i zachowania.

W celu rozwiązania problemów wykorzystywana jest komunikacja między obiektami.

Abstrakcja danych, hermetyzacja i dziedziczenie to istotne paradygmaty modelu obiektowego. Nie są one cechą żadnego konkretnego języka.

Język, który umożliwia efektywną abstrakcję, hermetyzację i dziedziczenie, jest językiem obiektowym.

Pełne korzyści z zastosowania modelu obiektowego można odnieść tylko dzięki uważnej analizie problemu, poprawnemu projektowaniu i efektywnej implementacji.