helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

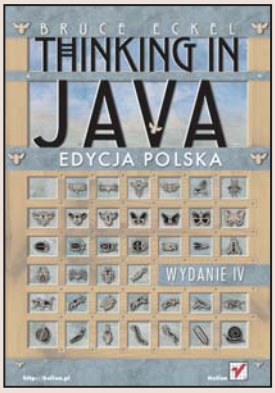
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Thinking in Java. Edycja polska. Wydanie IV



Autor: Bruce Eckel
Tłumaczenie: Przemysław Szeremiota
ISBN: 83-246-0111-2
Tytuł oryginału: Thinking in Java (4th Edition)
Format: B5, stron: 1256

Poznaj najnowszą wersję języka Java

- Opanuj zasady projektowania obiektowego
- Wykorzystaj Javę do tworzenia aplikacji
- Zastosuj najnowsze narzędzia i techniki

Popularność Javy stale rośnie, a każda następna wersja tego języka wnosi coś nowego. Poznanie wszystkich możliwości drzemiących w Javie wymaga sięgnięcia do sprawdzonego źródła wiedzy – książki „Thinking in Java". To światowy bestseller, który zyskał zasłużoną sławę najlepszego podręcznika do nauki Javy. Wzbudził entuzjazm i uznanie programistów przede wszystkim dzięki wyjątkowej przejrzystości, przemyślanej strukturze i trafnie dobranym przykładom.

Książka „Thinking in Java. Edycja polska. Wydanie IV" zawiera szczegółowe omówienie zasad programowania w Javie. Przeznaczona jest dla początkujących programistów i dla ekspertów. Przystępnie prezentuje zarówno zagadnienia podstawowe, jak i zaawansowane. Dziesiątki przykładów ułatwiają zrozumienie każdego tematu. Wszystko to sprawia, że poznajemy prawdziwą Javę – uniwersalną, czytelną, niezależną od platformy systemowej. Czytając tę książkę, dowiesz się, co jest niezbędne do tworzenia wydajnych i bezpiecznych aplikacji w Javie.

- Projektowanie obiektowe
- Zasady dokumentowania kodu źródłowego
- Operatory i sterowanie przebiegiem wykonywania programu
- Inicjalizacja i usuwanie obiektów
- Kolekcje obiektów
- Obsługa błędów
- Operacje wejścia i wyjścia
- Programowanie współbieżne
- Projektowanie interfejsów użytkownika

Przekonaj się, dlaczego książka „Thinking in Java"
jest uznawana za najlepszy podręcznik Javy dostępny na rynku

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

Przedmowa **19**

Wprowadzenie **29**

Rozdział 1. Wprowadzenie w świat obiektów **37**
Postępująca abstrakcja 38
Obiekt posiada interfejs 40
Obiekt dostarcza usługi 42
Ukrywanie implementacji 43
Wielokrotne wykorzystanie implementacji 44
Dziedziczenie 45
„Bycie czymś" a „bycie podobnym do czegoś" 48
Wymienialność obiektów z użyciem polimorfizmu 49
Hierarchia z pojedynczym korzeniem 52
Kontenery 53
Typy parametryzowane (typy ogólne) 54
Tworzenie obiektów i czas ich życia 55
Obsługa wyjątków — eliminowanie błędów 57
Współbieżność 57
Java i Internet 58
Czym jest sieć WWW? 58
Programowanie po stronie klienta 60
Programowanie po stronie serwera 65
Podsumowanie 65

Rozdział 2. Wszystko jest obiektem **67**
Dostęp do obiektów poprzez referencje 67
Wszystkie obiekty trzeba stworzyć 68
Gdzie przechowujemy dane 69
Przypadek specjalny: typy podstawowe 70
Tablice w Javie 71
Nigdy nie ma potrzeby niszczenia obiektu 72
Zasięg 72
Zasięg obiektów 73
Własne typy danych — słowo class 74
Pola i metody 74
Metody, argumenty i wartości zwracane 76
Lista argumentów 76

Tworzenie programu w Javie ... 78
Widoczność nazw ... 78
Wykorzystanie innych komponentów ... 78
Słowo kluczowe static ... 79
Twój pierwszy program w Javie ... 81
Kompilacja i uruchomienie ... 83
Komentarze oraz dokumentowanie kodu ... 84
Dokumentacja w komentarzach ... 84
Składnia ... 85
Osadzony HTML ... 86
Niektóre znaczniki dokumentacyjne ... 86
Przykład dokumentowania kodu ... 88
Styl programowania ... 89
Podsumowanie ... 90
Ćwiczenia ... 90

Rozdział 3. Operatory ... 93
Prosta instrukcja wyjścia ... 93
Używanie operatorów Javy ... 94
Kolejność operatorów ... 95
Przypisanie ... 95
Tworzenie nazw w wywołaniach metod ... 97
Operatory matematyczne ... 98
Jednoargumentowe operatory minus i plus ... 100
Operatory zwiększania i zmniejszania ... 100
Operatory relacji ... 101
Sprawdzanie równości obiektów ... 101
Operatory logiczne ... 103
Skracanie obliczenia wyrażenia logicznego ... 104
Literały ... 105
Zapis wykładniczy ... 106
Operatory bitowe ... 108
Operatory przesunięć ... 109
Operator trójargumentowy if-else ... 112
Operatory + i += dla klasy String ... 113
Najczęstsze pułapki przy używaniu operatorów ... 114
Operatory rzutowania ... 115
Obcinanie a zaokrąglanie ... 116
Promocja typu ... 117
W Javie nie ma „sizeof” ... 117
Kompendium operatorów ... 118
Podsumowanie ... 126

Rozdział 4. Sterowanie przebiegiem wykonania ... 127
Prawda i fałsz ... 127
if-else ... 128
Iteracja ... 129
do-while ... 129
for ... 130
Operator przecinka ... 131
Składnia foreach ... 132
return ... 134
break i continue ... 135
Niesławne „goto” ... 136
switch ... 140
Podsumowanie ... 142

Rozdział 5. Inicjalizacja i sprzątanie ... 143
Gwarantowana inicjalizacja przez konstruktor ... 143
Przeciążanie metod ... 145
Rozróżnianie przeciążonych metod ... 147
Przeciążanie a typy podstawowe ... 148
Przeciążanie przez wartości zwracane ... 151
Konstruktory domyślne ... 152
Słowo kluczowe this ... 153
Wywoływanie konstruktorów z konstruktorów ... 155
Znaczenie słowa static ... 157
Sprzątanie: finalizacja i odśmiecanie pamięci ... 157
Do czego służy finalize() ... 158
Musisz przeprowadzić sprzątanie ... 159
Warunek zakończenia ... 160
Jak działa odśmiecacz pamięci ... 161
Inicjalizacja składowych ... 164
Określanie sposobu inicjalizacji ... 166
Inicjalizacja w konstruktorze ... 167
Kolejność inicjalizacji ... 167
Inicjalizacja zmiennych statycznych ... 168
Jawna inicjalizacja statyczna ... 171
Inicjalizacja egzemplarza ... 172
Inicjalizacja tablic ... 173
Zmienne listy argumentów ... 177
Typy wyliczeniowe ... 182
Podsumowanie ... 185

Rozdział 6. Kontrola dostępu ... 187
Pakiet — jednostka biblioteczna ... 188
Organizacja kodu ... 189
Tworzenie unikatowych nazw pakietów ... 191
Własna biblioteka narzędziowa ... 194
Wykorzystanie instrukcji import do zmiany zachowania ... 196
Pułapka związana z pakietami ... 196
Modyfikatory dostępu w Javie ... 196
Dostęp pakietowy ... 197
public: dostęp do interfejsu ... 198
private: nie dotykać! ... 199
protected: dostęp „na potrzeby” dziedziczenia ... 200
Interfejs i implementacja ... 202
Dostęp do klas ... 203
Podsumowanie ... 207

Rozdział 7. Wielokrotne wykorzystanie klas ... 209
Składnia kompozycji ... 210
Składnia dziedziczenia ... 212
Inicjalizacja klasy bazowej ... 214
Delegacje ... 217
Łączenie kompozycji i dziedziczenia ... 218
Zapewnienie poprawnego sprzątania ... 220
Ukrywanie nazw ... 223
Wybór między kompozycją a dziedziczeniem ... 225
protected ... 226
Rzutowanie w górę ... 227
Dlaczego „w górę” ... 228
Jeszcze o kompozycji i dziedziczeniu ... 229

Słowo kluczowe final ... 229
Zmienne finalne ... 229
Metody finalne ... 233
Klasy finalne ... 235
Ostrożnie z deklaracją final ... 236
Inicjalizacja i ładowanie klas ... 237
Inicjalizacja w przypadku dziedziczenia ... 238
Podsumowanie ... 239

Rozdział 8. Polimorfizm ... 241
Rzutowanie w górę raz jeszcze ... 242
Zapominanie o typie obiektu ... 243
Mały trik ... 244
Wiązanie wywołania metody ... 245
Uzyskiwanie poprawnego działania ... 245
Rozszerzalność ... 248
Pułapka: „przesłanianie" metod prywatnych ... 251
Pułapka: statyczne pola i metody ... 252
Konstruktory a polimorfizm ... 253
Kolejność wywołań konstruktorów ... 253
Dziedziczenie a sprzątanie ... 255
Zachowanie metod polimorficznych wewnątrz konstruktorów ... 260
Kowariancja typów zwracanych ... 262
Projektowanie z użyciem dziedziczenia ... 263
Substytucja kontra rozszerzanie ... 264
Rzutowanie w dół a identyfikacja typu w czasie wykonania ... 265
Podsumowanie ... 267

Rozdział 9. Interfejsy ... 269
Klasy i metody abstrakcyjne ... 269
Interfejsy ... 273
Rozdzielenie zupełne ... 276
„Dziedziczenie wielobazowe" w Javie ... 280
Rozszerzanie interfejsu poprzez dziedziczenie ... 283
Kolizje nazw podczas łączenia interfejsów ... 284
Adaptowanie do interfejsu ... 285
Pola w interfejsach ... 287
Inicjalizacja pól interfejsów ... 288
Zagnieżdżanie interfejsów ... 289
Interfejsy a wytwórnie ... 291
Podsumowanie ... 294

Rozdział 10. Klasy wewnętrzne ... 295
Tworzenie klas wewnętrznych ... 295
Połączenie z klasą zewnętrzną ... 297
.this i .new ... 299
Klasy wewnętrzne a rzutowanie w górę ... 300
Klasy wewnętrzne w metodach i zasięgach ... 302
Anonimowe klasy wewnętrzne ... 304
Jeszcze o wzorcu Factory Method ... 308
Klasy zagnieżdżone ... 310
Klasy wewnątrz interfejsów ... 312
Sięganie na zewnątrz z klasy wielokrotnie zagnieżdżonej ... 313

Dlaczego klasy wewnętrzne ... 314
Domknięcia i wywołania zwrotne ... 316
Klasy wewnętrzne a szkielety sterowania ... 319
Dziedziczenie po klasach wewnętrznych ... 325
Czy klasy wewnętrzne mogą być przesłaniane? ... 326
Lokalne klasy wewnętrzne ... 327
Identyfikatory klas wewnętrznych ... 329
Podsumowanie ... 329

Rozdział 11. Kolekcje obiektów ... 331
Kontenery typowane i uogólnione ... 332
Pojęcia podstawowe ... 335
Dodawanie grup elementów ... 337
Wypisywanie zawartości kontenerów ... 339
Interfejs List ... 341
Interfejs Iterator ... 345
Interfejs ListIterator ... 348
Klasa LinkedList ... 349
Klasa Stack ... 350
Interfejs Set ... 352
Interfejs Map ... 355
Interfejs Queue ... 359
PriorityQueue ... 360
Collection kontra Iterator ... 362
Iteratory a pętle foreach ... 365
Idiom metody-adaptera ... 367
Podsumowanie ... 370

Rozdział 12. Obsługa błędów za pomocą wyjątków ... 375
Zarys koncepcji ... 376
Podstawy obsługi wyjątków ... 377
Argumenty wyjątków ... 378
Przechwytywanie wyjątku ... 379
Blok try ... 379
Obsługa wyjątków ... 379
Tworzenie własnych wyjątków ... 380
Rejestrowanie wyjątków ... 383
Specyfikacja wyjątków ... 386
Przechwytywanie dowolnego wyjątku ... 387
Stos wywołań ... 389
Ponowne wyrzucanie wyjątków ... 389
Sekwencje wyjątków ... 392
Standardowe wyjątki Javy ... 395
Przypadek specjalny: RuntimeException ... 396
Robienie porządków w finally ... 397
Do czego służy finally ... 399
Współdziałanie finally z return ... 401
Pułapka: zagubiony wyjątek ... 402
Ograniczenia wyjątków ... 404
Konstruktory ... 407
Dopasowywanie wyjątków ... 411
Rozwiązania alternatywne ... 413
Historia ... 414
Perspektywy ... 415

Przekazywanie wyjątków na konsolę ... 418
Zamiana wyjątków sprawdzanych na niesprawdzane ... 419
Wskazówki ... 421
Podsumowanie ... 421

Rozdział 13. Ciągi znaków ... 423
Niezmienność ciągów znakowych ... 423
StringBuilder kontra przeciążony operator '+' ... 424
Niezamierzona rekursja ... 428
Operacje na egzemplarzach klasy String ... 430
Formatowanie wyjścia ... 432
Funkcja printf() ... 432
System.out.format() ... 432
Klasa Formatter ... 433
Specyfikatory formatu ... 434
Konwersje ... 435
Metoda String.format() ... 438
Wyrażenia regularne ... 439
Podstawy ... 440
Tworzenie wyrażeń regularnych ... 442
Kwantyfikatory ... 444
Klasy Pattern oraz Matcher ... 446
metoda split() ... 453
Operacje zastępowania ... 454
Metoda reset() ... 456
Wyrażenia regularne i operacje wejścia-wyjścia Javy ... 457
Skanowanie wejścia ... 459
Separatory wartości wejściowych ... 461
Skanowanie wejścia przy użyciu wyrażeń regularnych ... 462
Klasa StringTokenizer ... 463
Podsumowanie ... 463

Rozdział 14. Informacje o typach ... 465
Potrzeba mechanizmu RTTI ... 465
Obiekt Class ... 467
Literały Class ... 472
Referencje klas uogólnionych ... 475
Nowa składnia rzutowania ... 477
Sprawdzanie przed rzutowaniem ... 478
Użycie literałów klas ... 484
Dynamiczne instanceof ... 485
Zliczanie rekurencyjne ... 487
Wytwórnie rejestrowane ... 488
instanceof a równoważność obiektów Class ... 491
Refleksja — informacja o klasie w czasie wykonania ... 493
Ekstraktor metod ... 494
Dynamiczne proxy ... 497
Obiekty puste ... 501
Imitacje i zalążki ... 507
Interfejsy a RTTI ... 507
Podsumowanie ... 512

Rozdział 15. Typy ogólne ... 515
Porównanie z językiem C++ ... 516
Proste uogólnienia ... 517
Biblioteka krotek ... 519
Klasa stosu ... 522
RandomList ... 523
Uogólnianie interfejsów ... 524
Uogólnianie metod ... 527
Wykorzystywanie dedukcji typu argumentu ... 528
Metody uogólnione ze zmiennymi listami argumentów ... 531
Metoda uogólniona w służbie klasy Generator ... 531
Uniwersalny Generator ... 532
Upraszczanie stosowania krotek ... 533
Uniwersalny kontener Set ... 535
Anonimowe klasy wewnętrzne ... 538
Budowanie modeli złożonych ... 540
Tajemnica zacierania ... 542
Jak to się robi w C++ ... 543
Słowo o zgodności migracji ... 546
Kłopotliwość zacierania ... 547
Na krawędzi ... 548
Kompensacja zacierania ... 552
Tworzenie egzemplarzy typów ... 553
Tablice typów ogólnych ... 556
Ramy ... 560
Symbole wieloznaczne ... 564
Jak bystry jest kompilator? ... 567
Kontrawariancja ... 568
Symbole wieloznaczne bez ram konkretyzacji ... 571
Konwersja z przechwyceniem typu ... 576
Problemy ... 578
Typy podstawowe jako parametry typowe ... 578
Implementowanie interfejsów parametryzowanych ... 580
Ostrzeżenia przy rzutowaniu ... 580
Przeciążanie ... 582
Zawłaszczenie interfejsu w klasie bazowej ... 583
Typy samoskierowane ... 584
Osobliwa rekurencja uogólnienia ... 584
Samoskierowanie ... 585
Kowariancja argumentów ... 588
Dynamiczna kontrola typów ... 591
Wyjątki ... 592
Domieszki ... 594
Domieszki w C++ ... 594
Domieszki z użyciem interfejsów ... 595
Zastosowanie wzorca projektowego Decorator ... 596
Domieszki w postaci dynamicznych proxy ... 598
Typowanie utajone ... 599
Kompensacja braku typowania utajonego ... 604
Refleksja ... 604
Aplikowanie metody do sekwencji obiektów ... 605
Kiedy nie ma pod ręką odpowiedniego interfejsu ... 608
Symulowanie typowania utajonego za pomocą adapterów ... 610

Obiekty funkcyjne w roli strategii ... 613
Podsumowanie — czy rzutowanie jest aż tak złe? ... 618
Dalsza lektura ... 620

Rozdział 16. Tablice ... 621
Co w nich takiego specjalnego? ... 621
Tablice to pełnoprawne obiekty ... 623
Tablice w roli wartości zwracanych ... 625
Tablice wielowymiarowe ... 627
Tablice a typy ogólne ... 631
Wytwarzanie danych testowych ... 633
Metoda Arrays.fill() ... 633
Generatory danych ... 634
Tworzenie tablic za pomocą generatorów ... 639
Narzędzia klasy Arrays ... 643
Kopiowanie tablic ... 643
Porównywanie tablic ... 645
Porównywanie elementów tablic ... 646
Sortowanie tablic ... 649
Przeszukiwanie tablicy posortowanej ... 650
Podsumowanie ... 652

Rozdział 17. Kontenery z bliska ... 655
Pełna taksonomia kontenerów ... 655
Wypełnianie kontenerów ... 656
Rozwiązanie z generatorem ... 657
Generatory dla kontenerów asocjacyjnych ... 659
Stosowanie klas abstrakcyjnych ... 662
Interfejs Collection ... 669
Operacje opcjonalne ... 672
Operacje nieobsługiwane ... 673
Interfejs List ... 675
Kontenery Set a kolejność elementów ... 678
SortedSet ... 681
Kolejki ... 683
Kolejki priorytetowe ... 684
Kolejki dwukierunkowe ... 685
Kontenery asocjacyjne ... 686
Wydajność ... 688
SortedMap ... 691
LinkedHashMap ... 692
Haszowanie i kody haszujące ... 693
Zasada działania hashCode() ... 696
Haszowanie a szybkość ... 699
Przesłonięcie metody hashCode() ... 702
Wybór implementacji ... 707
Infrastruktura testowa ... 708
Wybieranie pomiędzy listami ... 711
Zagrożenia testowania w małej skali ... 717
Wybieranie pomiędzy zbiorami ... 719
Wybieranie pomiędzy odwzorowaniami ... 720
Narzędzia dodatkowe ... 724
Sortowanie i przeszukiwanie list ... 727
Niemodyfikowalne kontenery Collection i Map ... 729
Synchronizacja Collection i Map ... 730

Przechowywanie referencji ... 731
WeakHashMap ... 734
Kontenery Java 1.0 i 1.1 ... 735
Vector i Enumeration ... 735
Hashtable ... 736
Stack ... 736
BitSet ... 738
Podsumowanie ... 740

Rozdział 18. Wejście-wyjście ... 741

Klasa File ... 741
Wypisywanie zawartości katalogu ... 742
Narzędzie do przeglądania katalogów ... 745
Tworzenie katalogów i sprawdzanie ich obecności ... 750
Wejście i wyjście ... 752
Typy InputStream ... 752
Typy OutputStream ... 753
Dodawanie atrybutów i użytecznych interfejsów ... 754
Odczyt z InputStream za pomocą FilterInputStream ... 755
Zapis do OutputStream za pomocą FilterOutputStream ... 756
Klasy Reader i Writer ... 757
Źródła i ujścia danych ... 758
Modyfikacja zachowania strumienia ... 758
Klasy niezmienione ... 759
Osobna i samodzielna RandomAccessFile ... 760
Typowe zastosowania strumieni wejścia-wyjścia ... 760
Buforowany plik wejścia ... 761
Wejście z pamięci ... 762
Formatowane wejście z pamięci ... 762
Wyjście do pliku ... 763
Przechowywanie i odzyskiwanie danych ... 765
Odczyt i zapis do plików o dostępie swobodnym ... 766
Strumienie-potoki ... 768
Narzędzia do zapisu i odczytu danych z plików ... 768
Odczyt plików binarnych ... 771
Standardowe wejście-wyjście ... 772
Czytanie ze standardowego wejścia ... 772
Zamiana System.out na PrintWriter ... 773
Przekierowywanie standardowego wejścia-wyjścia ... 773
Sterowanie procesami zewnętrznymi ... 774
Nowe wejście-wyjście ... 776
Konwersja danych ... 779
Pobieranie podstawowych typów danych ... 782
Widoki buforów ... 783
Manipulowanie danymi przy użyciu buforów ... 787
Szczegółowe informacje o buforach ... 787
Pliki odwzorowywane w pamięci ... 791
Blokowanie plików ... 795
Kompresja ... 798
Prosta kompresja do formatu GZIP ... 798
Przechowywanie wielu plików w formacie Zip ... 799
Archiwa Javy (JAR) ... 801

Serializacja obiektów ... 803
Odnajdywanie klasy ... 806
Kontrola serializacji ... 808
Stosowanie trwałości ... 815
XML ... 821
Preferencje ... 824
Podsumowanie ... 826

Rozdział 19. Typy wyliczeniowe ... 827
Podstawowe cechy typów wyliczeniowych ... 827
Wyliczenia a importy statyczne ... 828
Dodawanie metod do typów wyliczeniowych ... 829
Przesłanianie metod typu wyliczeniowego ... 830
Wyliczenia w instrukcjach wyboru ... 831
Tajemnica metody values() ... 832
Implementuje, nie dziedziczy ... 835
Wybór losowy ... 836
Organizacja na bazie interfejsów ... 837
EnumSet zamiast znaczników ... 841
Stosowanie klasy EnumMap ... 843
Metody specjalizowane dla elementów wyliczenia ... 844
Typy wyliczeniowe w łańcuchu odpowiedzialności ... 848
Typy wyliczeniowe a automaty stanów ... 851
Rozprowadzanie wielokrotne ... 856
Rozprowadzanie z udziałem typów wyliczeniowych ... 859
Stosowanie metod specjalizowanych dla elementów wyliczenia ... 861
Rozprowadzanie za pomocą EnumMap ... 863
Z tablicą dwuwymiarową ... 864
Podsumowanie ... 865

Rozdział 20. Adnotacje ... 867
Podstawy składni adnotacji ... 868
Definiowanie adnotacji ... 869
Metaadnotacje ... 870
Procesory adnotacji ... 871
Elementy adnotacji ... 872
Ograniczenia wartości domyślnych ... 872
Generowanie plików zewnętrznych ... 873
Adnotacje nie dają się dziedziczyć ... 876
Implementowanie procesora ... 876
Przetwarzanie adnotacji za pomocą apt ... 879
Program apt a wizytacje ... 883
Adnotacje w testowaniu jednostkowym ... 886
@Unit a typy ogólne ... 895
Implementacja @Unit ... 896
Usuwanie kodu testującego ... 903
Podsumowanie ... 905

Rozdział 21. Współbieżność ... 907
Oblicza współbieżności ... 908
Szybsze wykonanie ... 909
Ulepszanie projektu ... 911
Podstawy wielowątkowości ... 912
Definiowanie zadań ... 913
Klasa Thread ... 914
Wykonawcy ... 916

Zwracanie wartości z zadań ... 919
Usypianie — wstrzymywanie wątku ... 920
Priorytet wątku ... 921
Przełączanie ... 923
Wątki-demony ... 924
Wariacje na temat wątków ... 928
Terminologia ... 933
Łączenie wątków ... 934
Tworzenie reaktywnego interfejsu użytkownika ... 935
Grupy wątków ... 936
Przechwytywanie wyjątków ... 937
Współdzielenie zasobów ... 940
Niewłaściwy dostęp do zasobów ... 940
Rozstrzyganie współzawodnictwa o zasoby współdzielone ... 943
Atomowość i widoczność ... 948
Klasy „atomowe” ... 955
Sekcje krytyczne ... 956
Synchronizacja dostępu na bazie innych obiektów ... 961
Lokalna pamięć wątku ... 962
Przerywanie wykonania zadań ... 964
Ogród botaniczny (symulacja) ... 964
Przerywanie zablokowanego wątku ... 967
Wymuszanie przerwania wykonania ... 968
Sprawdzanie przerwania ... 976
Współdziałanie wątków ... 978
Metody wait() i notifyAll() ... 979
notify() kontra notifyAll() ... 984
Producenci i konsumenci ... 987
Producenci, konsumenci i kolejki ... 992
Przekazywanie danych pomiędzy zadaniami za pomocą potoków ... 997
Zakleszczenie ... 999
Nowe komponenty biblioteczne ... 1004
CountDownLatch ... 1004
CyclicBarrier ... 1006
DelayQueue ... 1008
PriorityBlockingQueue ... 1011
Sterowanie szklarnią — planowanie uruchamiania zadań ... 1014
Semaphore ... 1017
Exchanger ... 1020
Symulacje ... 1022
Symulacja okienka kasowego ... 1022
Symulacja sali restauracyjnej ... 1027
Rozdzielanie zadań ... 1031
Wydajność ... 1036
Porównanie technologii muteksów ... 1036
Kontenery bez blokad ... 1044
Blokowanie optymistyczne ... 1051
Blokady ReadWriteLock ... 1053
Obiekty aktywne ... 1055
Podsumowanie ... 1059
Dalsza lektura ... 1061

Rozdział 22. Graficzne interfejsy użytkownika 1063
Aplety 1065
Podstawy biblioteki Swing 1066
Platforma prezentacyjna 1069
Tworzenie przycisku 1069
Przechwytywanie zdarzenia 1070
Obszary tekstowe 1073
Rozmieszczanie elementów interfejsu 1074
BorderLayout 1075
FlowLayout 1076
GridLayout 1076
GridBagLayout 1077
Pozycjonowanie bezpośrednie 1077
BoxLayout 1077
Najlepsze rozwiązanie? 1078
Model zdarzeń w Swingu 1078
Rodzaje zdarzeń i odbiorników 1079
Śledzenie wielu zdarzeń 1084
Wybrane komponenty Swing 1086
Przyciski 1087
Ikony 1089
Podpowiedzi 1091
Pola tekstowe 1091
Ramki 1093
Miniedytor 1094
Pola wyboru 1095
Przyciski wyboru 1096
Listy rozwijane 1097
Listy 1098
Zakładki 1100
Okna komunikatów 1100
Menu 1102
Menu kontekstowe 1107
Rysowanie 1109
Okna dialogowe 1112
Okna dialogowe plików 1115
HTML w komponentach Swing 1117
Suwaki i wskaźniki postępu 1117
Zmiana stylu interfejsu 1119
Drzewka, tabele i schowek 1121
JNLP oraz Java Web Start 1121
Swing a współbieżność 1126
Zadania długotrwałe 1126
Wizualizacja wielowątkowości interfejsu użytkownika 1133
Programowanie wizualne i komponenty JavaBean 1135
Czym jest komponent JavaBean? 1136
Wydobycie informacji o komponencie poprzez klasę Introspector 1138
Bardziej wyszukany komponent 1143
Komponenty JavaBean i synchronizacja 1146
Pakowanie komponentu JavaBean 1150
Bardziej złożona obsługa komponentów JavaBean 1151
Więcej o komponentach JavaBean 1152
Alternatywy wobec biblioteki Swing 1152

Flex — aplikacje klienckie w formacie Flash ... 1153
Ahoj, Flex ... 1154
Kompilowanie MXML ... 1155
MXML i skrypty ActionScript ... 1156
Kontenery i kontrolki ... 1156
Efekty i style ... 1158
Zdarzenia ... 1159
Połączenie z Javą ... 1159
Modele danych i wiązanie danych ... 1162
Kompilowanie i instalacja ... 1163
Aplikacje SWT ... 1164
Instalowanie SWT ... 1165
Ahoj, SWT ... 1165
Eliminowanie powtarzającego się kodu ... 1168
Menu ... 1170
Panele zakładek, przyciski i zdarzenia ... 1171
Grafika ... 1174
Współbieżność w SWT ... 1176
SWT czy Swing? ... 1178
Podsumowanie ... 1178
Zasoby ... 1179

Dodatek A Materiały uzupełniające ... 1181
Suplementy do pobrania ... 1181
Thinking in C ... 1181
Szkolenie Thinking in Java ... 1182
Szkolenie na CD-ROM-ie Hands-On Java ... 1182
Szkolenie Thinking in Objects ... 1182
Thinking in Enterprise Java ... 1183
Thinking in Patterns (with Java) ... 1184
Szkolenie Thinking in Patterns ... 1184
Konsultacja i analiza projektów ... 1185

Dodatek B Zasoby ... 1187
Oprogramowanie ... 1187
Edytory i środowiska programistyczne ... 1187
Książki ... 1188
Analiza i projektowanie ... 1189
Python ... 1191
Lista moich książek ... 1191

Skorowidz ... 1193

Rozdział 1.

Wprowadzenie w świat obiektów

„Dzielimy otaczający nas świat i kategoryzujemy go, przypisując po drodze tym kategoriom znaczenia — ponieważ jesteśmy częściami społeczeństwa bazującego na mowie, którego sens jest w mowie zakodowany… Nie moglibyśmy porozumiewać się, nie stając się częścią tej społeczności i nie korzystając z tego systemu klasyfikacji."

— Benjamin Lee Whorf (1897 – 1941)

Genezą rewolucji komputerowej była maszyna — dlatego genezą języków programowania było „udawanie" maszyny.

Komputery jednak bardziej niż maszynami są „wzmacniaczami umysłu" (lub „rowerami dla umysłu" — jak mawia Steve Jobs), a także nowym środkiem ekspresji. W rezultacie w coraz mniejszym stopniu przypominają maszyny, a w coraz większym fragmenty naszego umysłu, a zarazem klasyczne dziedziny sztuki, takie jak: literatura, malarstwo, rzeźba, animacja i film. Programowanie zorientowane obiektowo jest częścią ruchu chcącego uczynić z komputera kolejny środek ekspresji.

W tym rozdziale przedstawię podstawowe idee programowania zorientowanego obiektowo (ang. *Object Oriented Programming*, OOP). Dokonam także przeglądu obiektowych metod tworzenia oprogramowania. Podobnie jak reszta książki, rozdział ten zakłada pewien poziom doświadczenia Czytelnika w programowaniu proceduralnym, choć niekoniecznie w języku C. Kto nie czuje się mocny w tej dziedzinie, powinien przejść multimedialny kurs *Thinking in C* dostępny do pobrania na stronach witryny *www.MindView.net*.

Treść rozdziału stanowi podstawę, a jednocześnie materiał uzupełniający resztę książki. Wiele osób nie chce zagłębiać się w programowanie obiektowe bez uprzedniego zrozumienia jego ogólnych koncepcji — dlatego opiszę tu wiele z nich, tworząc w ten sposób solidny obraz całej techniki. Z drugiej jednak strony są tacy, którzy mają problemy ze zrozumieniem ogólnych idei przed poznaniem „mechaniki" — ludzie ci mogą poczuć się zagubieni i zakłopotani bez jakiegoś fragmentu kodu jako punktu zaczepienia. Jeżeli należysz do tej drugiej grupy i nie możesz się już doczekać omawiania specyfiki języka, oczywiście

możesz opuścić ten rozdział — na tym etapie nie uniemożliwi to pisania programów i nauki języka. Powinieneś jednakże powrócić do niego później, aby zrozumieć, dlaczego obiekty są ważne i jak przy ich użyciu projektować oprogramowanie.

Postępująca abstrakcja

Każdy język programowania dostarcza pewnej abstrakcji. Można próbować bronić tezy, że złożoność problemów, jakie jesteśmy w stanie rozwiązać, jest bezpośrednio zależna od rodzaju i jakości używanej abstrakcji. Przez „rodzaj" abstrakcji rozumiem odpowiedź na pytanie: „Czym jest to, czego abstrakcji dokonujemy?". Język asembler jest niewielką abstrakcją maszyny. Wiele tzw. imperatywnych języków, które stworzono później (takich jak Fortran, BASIC i C), stanowiło niewielką abstrakcję asemblera. Były w porównaniu z nim znacznym usprawnieniem, jednak pierwotna abstrakcja pozostała niezmieniona — wciąż wymagane było myślenie w kategoriach struktury komputera, a nie struktury rozwiązywanego problemu. Programista musiał dokonać powiązania pomiędzy modelem maszyny (w „przestrzeni rozwiązania", czyli miejscu, w którym modelujemy problem, takim jak komputer) a modelem samego rozwiązywanego problemu (w „przestrzeni problemu", czyli miejscu, w którym problem istnieje, np. w działalności firmy). Wysiłek wymagany przy konstrukcji takiego odwzorowania w połączeniu z faktem, iż jest ono czymś zewnętrznym w stosunku do języka programowania, powoduje, że programy są trudne do napisania i kosztowne w pielęgnacji. Ubocznym skutkiem tych kłopotów jest powstanie całej gałęzi wiedzy, zwanej „metodami programowania".

Alternatywą dla modelowania maszyny jest modelowanie samego problemu. Wczesne języki programowania, takie jak LISP czy APL, przyjmowały określony sposób widzenia świata („Wszystkie problemy są w ostateczności listami" albo „Wszystkie problemy są algorytmiczne"). PROLOG sprowadza wszystkie problemy do ciągów decyzji. Powstały języki przeznaczone do programowania opartego na ograniczeniach (ang. *constraint-based programming*) oraz do programowania wyłącznie poprzez manipulację symbolami graficznymi (te ostatnie okazały się zbyt restrykcyjne). Każde z tych podejść jest odpowiednie dla określonej klasy problemów, dla której zostały opracowane, jednak w innych przypadkach staje się nieefektywne.

Programowanie obiektowe idzie krok dalej, dostarczając programiście narzędzie do reprezentowania elementów w przestrzeni problemu. Reprezentacja ta jest dostatecznie ogólna, by nie ograniczała się do jakiegoś konkretnego typu problemów. Odwołujemy się do elementów w przestrzeni problemu, jak i do ich odpowiedników w przestrzeni rozwiązania, używając tego samego słowa — „obiekt" (oczywiście potrzebne są również obiekty, które nie mają odpowiedników w przestrzeni problemu). Pomysł polega na umożliwieniu programowi dostosowania się do specyficznego języka danego problemu poprzez dodanie nowych typów obiektów, dzięki czemu przy czytaniu kodu opisującego rozwiązanie natrafiamy na słowa wyrażające sam problem. Abstrakcja taka jest potężniejsza i bardziej elastyczna od tych, którymi dysponowaliśmy wcześniej[1]. Programowanie obiektowe pozwala więc opisywać problem raczej w kategoriach mu właściwych, nie zaś

[1] Niektórzy projektanci języków uznali, że programowanie obiektowe samo w sobie nie wystarcza, aby w prosty sposób rozwiązać wszystkie problemy programistyczne. Promują oni kombinację kilku różnych rozwiązań, zwaną *programowaniem z wieloma paradygmatami* (ang. *multiparadigm programming languages*). Patrz Timothy Budd, *Multiparadigm programming in Leda*, Addison-Wesley 1995.

w kategoriach komputera, na którym zostanie uruchomione rozwiązanie. Połączenie z komputerem jednak wciąż istnieje. Każdy obiekt przypomina mały komputer — ma swój wewnętrzny stan oraz zestaw operacji, które może wykonać, jeśli zostanie o to poproszony. Nie wydaje się to jednak złą analogią do obiektów świata rzeczywistego — one również mają pewną charakterystykę oraz zachowują się w określony sposób.

Alan Kay podsumował pięć podstawowych cech Smalltalka, pierwszego udanego języka obiektowego, a zarazem jednego z poprzedników Javy. Cechy te reprezentują czyste podejście obiektowe:

1. **Wszystko jest obiektem**. Można wyobrazić sobie obiekt jako wymyślną zmienną — taką, która nie tylko przechowuje dane, ale może także realizować żądania, czyli wykonywać na sobie pewne operacje. Teoretycznie, dowolne pojęcia ze świata rozwiązywanego problemu (psy, budynki, usługi itd.) można reprezentować w programie za pomocą obiektu.
2. **Program jest zbiorem obiektów, które poprzez wysyłanie komunikatów mówią sobie nawzajem, co robić**. „Wysłanie komunikatu" do obiektu to inaczej zażądanie wykonania przez niego pewnej operacji. Można też myśleć o komunikacie jako o wywołaniu funkcji przynależnej do konkretnego obiektu.
3. **Każdy obiekt posiada własną pamięć, na którą składają się inne obiekty**. Innymi słowy, nowy obiekt tworzymy, łącząc w jeden pakiet grupę już istniejących obiektów. Budujemy w ten sposób złożone programy, ukrywając jednocześnie ich złożoność za prostotą obiektów.
4. **Każdy obiekt posiada swój typ**. Mówiąc potocznie, każdy obiekt jest *instancją* pewnej *klasy*, gdzie „klasa" jest synonimem „typu". Najistotniejszą cechą wyróżniającą klasę jest odpowiedź na pytanie: „Jakie komunikaty można do niej wysyłać?".
5. **Wszystkie obiekty danego typu mogą otrzymywać te same komunikaty**. To stwierdzenie, jak się później przekonamy, może być nieco mylące. Ponieważ obiekt typu „okrąg" jest jednocześnie obiektem typu „figura", zatem „okrąg" może otrzymywać komunikaty odpowiednie dla „figury". Oznacza to, iż możliwe jest pisanie kodu odwołującego się do „figury", który będzie automatycznie obsługiwał wszystkie obiekty pasujące do opisu typu „figura". Ta *zastępowalność* jest jedną z najpotężniejszych idei programowania obiektowego.

Bardziej zwięzły i przejrzysty opis obiektu podaje Booch:

> *Obiekt ma stan, zachowanie i tożsamość.*

Oznacza to, że obiekt może posiadać dane wewnętrzne (określające jego stan), metody (stanowiące zachowanie obiektu) i każdy obiekt można w jednoznaczny sposób odróżnić od wszelkich innych obiektów — a konkretnie rzecz biorąc, z każdym obiektem jest skojarzony unikalny adres pamięci[2].

[2] Założenie to jest nieco ograniczające, gdyż można sobie wyobrazić obiekty istniejące na różnych komputerach oraz w różnych przestrzeniach adresowych, istnieje możliwość przechowywania obiektów na dysku. W takich przypadkach tożsamość obiektu należy określać nie na podstawie adresu pamięci, lecz w inny sposób.

Obiekt posiada interfejs

Pierwszym człowiekiem, który prowadził dokładne rozważania nad pojęciem *typu*, był prawdopodobnie Arystoteles — mówił on o „klasie ryb" oraz „klasie ptaków". Idea mówiąca, że wszystkie obiekty, będąc unikatowymi, należą jednocześnie do pewnych klas obiektów o wspólnych cechach i sposobach zachowania, została bezpośrednio wykorzystana w języku Simula 67. Podstawowym słowem kluczowym tego pierwszego obiektowego języka było słowo `class`, wprowadzające do programu nowy typ.

Jak sama nazwa wskazuje, Simula została zaprojektowana w celu tworzenia symulacji, takich jak klasyczny „problem kasjera bankowego". Mamy w nim do czynienia ze zbiorem kasjerów, klientów, kont, transakcji oraz sum pieniężnych — mnóstwem „obiektów". Obiekty, które są identyczne z wyjątkiem ich stanu podczas wykonywania programu, są pogrupowane w „klasy obiektów" — stąd właśnie wzięło się słowo kluczowe `class` (*klasa*). Tworzenie abstrakcyjnych typów danych (*klas*) jest jedną z fundamentalnych koncepcji programowania zorientowanego obiektowo. Abstrakcyjne typy danych zachowują się prawie tak samo, jak typy podstawowe: można tworzyć zmienne takiego typu (w języku programowania obiektowego nazywane obiektami lub egzemplarzami), a następnie manipulować nimi (proces ten nazywamy wysyłaniem komunikatów lub *żądań* — wysyłamy do obiektu komunikat, a on „domyśla się", co z nim zrobić). Składowe (elementy) danej klasy posiadają pewne cechy wspólne (np. każde konto ma saldo, każdy kasjer może przyjąć wpłatę), równocześnie jednak poszczególne składniki moją swój indywidualny stan — każde konto ma inne saldo, każdy kasjer inaczej się nazywa itd. A zatem każdy klient, konto, transakcja mogą być reprezentowane w programie komputerowym przez unikatowy byt. Byt ten jest obiektem, a każdy obiekt należy do określonej klasy, która definiuje jego cechy i zachowanie.

Tak więc, mimo iż w programowaniu obiektowym zajmujemy się tak naprawdę tworzeniem nowych typów, niemal wszystkie obiektowe języki programowania stosują słowo kluczowe *class*, nie *type*. Zatem kiedy widzimy słowo „typ", powinniśmy pomyśleć „klasa" — i na odwrót[3].

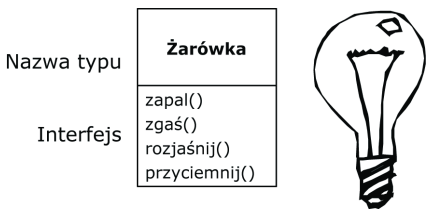
Ponieważ klasa opisuje zbiór obiektów o tej samej charakterystyce (danych składowych) oraz tych samych możliwych zachowaniach (możliwościach), jest typem danych tak samo prawdziwym, jak np. liczba zmiennoprzecinkowa, która także posiada pewne dane charakterystyczne (czyli swoją wartość) i zestaw dozwolonych na niej operacji (zachowań). Jedyna różnica polega na tym, że programista definiuje klasy odpowiadające raczej rozwiązywanemu problemowi — nie jest zmuszony do używania istniejących typów zaprojektowanych w celu reprezentowania właściwych maszynie jednostek przechowywania informacji. Dokonujemy więc rozszerzenia języka poprzez dodanie nowych typów danych, specyficznych dla naszych potrzeb. System programistyczny przyjmuje nasze klasy i zapewnia im taką samą „opiekę", jak typom wbudowanym, łącznie ze sprawdzaniem zgodności typów.

Programowanie obiektowe nie ogranicza się do budowania symulacji. Można nie zgadzać się z tezą, że każdy program jest w pewnym sensie symulacją rozwiązywanego przez siebie problemu, jednak mimo to prawdą pozostaje, że techniki obiektowe mogą z łatwością sprowadzić dużą klasę problemów do prostych rozwiązań.

[3] Niektórzy dokonują tu rozróżnienia, twierdząc, że typ określa tylko interfejs, natomiast klasa jest pewną implementacją tego interfejsu.

Gdy klasa jest już stworzona, możemy utworzyć dowolną liczbę jej obiektów, a następnie manipulować nimi tak, jak gdyby istniały w przestrzeni problemu. Jednym z podstawowych zadań w programowaniu obiektowym jest stworzenie jednoznacznego odwzorowania pomiędzy przestrzenią problemu a przestrzenią rozwiązania.

Jak jednak zmusić obiekt do zrobienia czegoś pożytecznego? Musi istnieć jakiś sposób zażądania od niego wykonania pewnej operacji, takiej jak zakończenie transakcji, narysowanie czegoś na ekranie czy też zmiana położenia przełącznika. Poza tym każdy obiekt może zrealizować tylko ściśle określone operacje. Żądania, jakie można skierować pod adresem danego obiektu, zdefiniowane są przez jego *interfejs*, a interfejs jest wyznaczony przez typ obiektu. Prostym przykładem może być reprezentacja żarówki:




```
Żarowka żr = new Żarowka();
żr.zapal();
```

Interfejs ustala, *jakie* żądania można kierować do danego obiektu, musi jednak istnieć kod realizujący te żądania. To on, wraz z ukrytymi danymi, składa się na *implementację*. Z punktu widzenia programowania proceduralnego nie jest to skomplikowane. Typ posiada powiązaną z każdym możliwym żądaniem funkcję, która jest wywoływana w chwili, gdy określone żądanie jest kierowane pod adresem jakiegoś obiektu. Proces ten jest określany jako „przesyłanie komunikatu" (żądanie) do obiektu i to obiekt wie, jak na dany komunikat zareagować (wykonuje kod).

W powyższym przykładzie typem (klasą) jest `Żarowka`, nazwą konkretnego obiektu typu `Żarowka` jest `żr`, a żądania, jakie `Żarowka` może obsłużyć, to: zapalenie, zgaszenie, rozjaśnienie i przyciemnienie. Stworzenie obiektu klasy `Żarowka` polega na zdefiniowaniu „referencji" (`żr`) dla tego obiektu, a następnie wywołaniu operatora `new` realizującego żądanie stworzenia nowego obiektu danego typu. W celu wysłania do obiektu komunikatu piszemy jego nazwę i łączymy ją za pomocą kropki z nazwą komunikatu, jaki chcemy wysłać. Z punktu widzenia użytkownika klas zdefiniowanych przez kogoś innego powiedzieliśmy już wszystko na temat programowania z wykorzystaniem obiektów.

Przedstawiony powyżej diagram jest zgodny z formatem zdefiniowanym przez UML (ang. *Unified Modeling Language* — ujednolicony język modelowania). Każda klasa jest reprezentowana przez prostokąt, którego górna część przeznaczona jest na jej nazwę, środkowa na pola danych (te, które chcemy opisywać), dolna zaś na *funkcje składowe* (funkcje należące do obiektu, to one otrzymują ostatecznie wszelkie komunikaty przesłane do obiektu). Bardzo często w diagramach projektowych języka UML przedstawiane są tylko nazwa i funkcje składowe klasy, dlatego środkowa część prostokąta nie jest rysowana. Jeśli interesuje nas tylko nazwa klasy — pomijamy także część dolną.

Obiekt dostarcza usługi

Pomimo prób tworzenia i zrozumienia projektu programu jednym z najlepszych sposobów pojmowania obiektów jest myślenie o nich jako o „dostawcach usług". Sam program świadczy pewne usługi dla użytkownika, realizując je przy użyciu usług oferowanych przez inne obiekty. Naszym celem jest stworzenie grupy obiektów (a jeszcze lepiej — odszukanie istniejącej biblioteki zawierającej takie obiekty) udostępniających usługi, które optymalnie nadają się do rozwiązania problemu.

Jednym ze sposobów, w jaki można rozpocząć realizację tego zadania, jest zadanie sobie pytania: „Gdybym, w jakiś magiczny sposób, mógł wyciągnąć je z kapelusza, jakie obiekty rozwiązałyby mój problem?". Na przykład, przypuścimy, że piszemy program księgowy. Można by sobie wyobrazić obiekty zawierające stosowne, predefiniowane formularze, grupę obiektów realizujących odpowiednie obliczenia księgowe oraz obiekt obsługujący drukowanie czeków i faktur na wszelkiego typu drukarkach. Może niektóre z tych obiektów już istnieją, a jeśli nie, to jak powinny wyglądać? Jakie usługi powinny udostępniać *te* obiekty oraz jakich innych obiektów będą *one* potrzebować w celu realizacji swych zobowiązań? Postępując w ten sposób, można w końcu dojść do etapu, w którym będzie można stwierdzić, że „ten obiekt jest na tyle prosty, że można go szybko napisać" albo „jestem pewny, że taki obiekt już istnieje". To bardzo rozsądny sposób dekompozycji problemu na zbiór obiektów.

Wyobrażanie sobie obiektu jako dostawcy usług ma jeszcze jedną zaletę: pomaga w poprawieniu spójności obiektu. *Wysoka spójność* obiektu jest podstawowym wyznacznikiem jakości projektu oprogramowania. Oznacza ona, że różne aspekty fragmentów oprogramowania (takich jak obiekty, lecz może to także dotyczyć metod lub bibliotek obiektów) dobrze „pasują do siebie". Jednym z problemów, które się pojawiają podczas projektowania obiektów, jest wypełnienie nich zbyt wieloma możliwościami funkcjonalnymi. Na przykład, tworząc moduł drukujący, można dojść do wniosku, że konieczny będzie obiekt, który „wie wszystko" na temat formatowania i drukowania. Zapewne okaże się, że to zbyt wiele jak na jeden obiekt i że w rzeczywistości konieczne są trzy lub nawet więcej obiektów. Jeden z nich mógłby być katalogiem zawierającym wszystkie możliwe formaty czeków; z niego byłyby pobierane informacje dotyczące sposobu drukowania czeków. Kolejny obiekt, lub grupa obiektów, mógłby stanowić ogólny interfejs drukujący, który dysponowałby wiedzą na temat wszystkich rodzajów drukarek (lecz nie wiedziałby niczego na temat księgowości — taki obiekt bardziej nadaje się do kupienia niż do samodzielnego napisania). Trzeci obiekt mógłby korzystać z usług dwóch poprzednich, aby realizować zamierzone zadanie. A zatem każdy z obiektów dysponuje spójnym zbiorem udostępnianych usług. W poprawnym, obiektowo zorientowanym projekcie każdy obiekt realizuje dobrze jedno zadanie, lecz nie stara się robić zbyt wiele. Jak można się było przekonać, rozwiązanie takie nie tylko pomaga w odnajdywaniu obiektów, które można by wykorzystać (obiekt stanowiący interfejs drukowania), lecz także stwarza możliwość pisania obiektów nadających się do wielokrotnego wykorzystania w różnych programach (na przykład katalog czeków).

Traktowanie obiektu jako dostawcy usług jest podejściem, które wiele ułatwia i jest przydatne nie tylko podczas procesu projektowania, lecz także w sytuacjach, gdy inne osoby starają się przeanalizować kod lub powtórnie wykorzystać obiekty — jeśli można ocenić wartość obiektu na podstawie świadczonych przez niego usług, znacznie łatwiej można go dopasować do tworzonego projektu.

Ukrywanie implementacji

Warto dokonać rozróżnienia pomiędzy *twórcami klas* (ang. *class creators*, którzy definiują nowe typy danych) a *programistami-klientami* (ang. *client programmers*[4]; „konsumentami" klas, którzy wykorzystują te typy danych w swoich aplikacjach). Celem programisty-klienta jest zebranie zestawu klas-narzędzi gotowych do wykorzystania w szybkim tworzeniu aplikacji. Celem twórcy klas jest natomiast przygotowywanie takich klas, które udostępniają programiście-klientowi jedynie to, co dla niego niezbędne, a całą resztę trzymają w ukryciu. Dlaczego? Ponieważ to, co ukryte, nie może zostać wykorzystane przez programistę-klienta, a zatem twórca może zmienić niewidoczną część, nie przejmując się ewentualnym wpływem tych zmian na inne części. Ukryta część zwykle reprezentuje delikatne wnętrze obiektu, które mogłoby z łatwością zostać „popsute" przez nieuważnego lub niedoinformowanego programistę-klienta. Z tego powodu ukrywanie implementacji zmniejsza liczbę błędów w programach.

Nie sposób przecenić idei ukrywania implementacji. W każdej relacji istotne jest istnienie ograniczeń respektowanych przez wszystkie uczestniczące w niej strony. Gdy tworzymy bibliotekę, nawiązujemy równocześnie pewną relację z jej klientem, będącym również programistą, ale takim, który ma na celu „złożenie" aplikacji za pomocą naszej biblioteki, ewentualnie zbudowanie biblioteki większej. Gdy wszystkie składniki klasy są dostępne dla wszystkich, wtedy programista-klient może z klasą zrobić wszystko — nie istnieje żaden sposób na wymuszenie przestrzegania reguł. Nawet jeśli naprawdę nie chcemy, aby klient manipulował bezpośrednio niektórymi ze składników naszej klasy, bez mechanizmów kontroli dostępu nie istnieje sposób uniemożliwienia tego. Wszystko jest publiczne i widoczne dla całego świata.

Pierwszym uzasadnieniem kontroli dostępu jest chęć wymuszenia na kliencie trzymania rąk z daleka od rzeczy, których nie powinien dotykać — części niezbędnych do wykonywania wewnętrznych operacji naszego typu danych, nie będących jednak częścią interfejsu, którego potrzebują użytkownicy do rozwiązywania konkretnych problemów. Kontrola taka jest korzystna dla użytkowników, pozwala im bowiem łatwo odróżnić to, co dla nich istotne, od tego, co mogą zignorować.

Drugim powodem wprowadzenia kontroli dostępu jest umożliwienie projektantowi biblioteki wymiany wewnętrznych mechanizmów klasy bez zastanawiania się nad wpływem tej czynności na programistów-klientów. Można na przykład zaimplementować jakąś klasę w prymitywny sposób w celu uproszczenia pracy, a w późniejszym terminie (jeśli okaże się to konieczne) przepisać ją, aby działała szybciej. Jeżeli interfejs i implementacja są od siebie wyraźnie oddzielone i chronione, wtedy osiągnięcie tego nie jest trudne.

Java posiada trzy słowa kluczowe służące do ustanowienia rozgraniczeń w klasach. Są to: `public` (publiczny), `private` (prywatny) i `protected` (chroniony). Ich użycie i znaczenie jest dosyć oczywiste. Są to tzw. *specyfikatory dostępu* (ang. *access specifiers*) — określają, kto jest upoważniony do używania następujących po nich definicji. Specyfikator `public` oznacza, że definicje są dostępne dla każdego, natomiast `private`, że dostęp do nich posiada jedynie twórca danej klasy wewnątrz jej funkcji składowych. Jeżeli ktoś

[4] Termin ten autor zawdzięcza przyjacielowi, Scottowi Meyersowi.

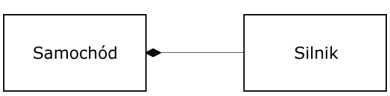
próbuje używać prywatnych (`private`) elementów naszej klasy, powoduje tym samym wystąpienie błędu już w czasie kompilacji programu. Słowo kluczowe `protected` działa prawie tak samo jak `private` — różnica polega na tym, że klasy dziedziczące z naszej mają dostęp do elementów typu `protected`, nie mają natomiast do tych określonych jako `private`. Pojęcie dziedziczenia wprowadzę już niedługo.

Java posiada także „domyślny" tryb dostępu, który jest wykorzystywany, jeśli nie podamy żadnego ze wspomnianych specyfikatorów. Określany jest on czasem jako „pakietowy" (ang. *package access*), ponieważ do składowych pakietowych mogą się odwoływać inne klasy z tego samego pakietu, natomiast poza pakietem widziane są one jako prywatne.

Wielokrotne wykorzystanie implementacji

Stworzona i przetestowana klasa powinna (w sytuacji idealnej) stanowić użyteczny fragment kodu. Okazuje się jednak, iż osiągnięcie takiego stanu rzeczy nie jest tak proste, jak mogłoby się niektórym wydawać — stworzenie dobrego rozwiązania wymaga sporego doświadczenia i intuicji. Gdy jednak rozwiązanie takie już powstanie, wtedy aż prosi się o wielokrotne wykorzystanie. Umożliwienie takiego wielokrotnego wykorzystania kodu jest jedną z głównych zalet obiektowo zorientowanych języków programowania.

Najprostszym sposobem wykorzystania kodu klasy jest bezpośrednie użycie obiektu tej klasy, można jednak również umieścić obiekt wewnątrz nowej klasy. Nazywamy to „tworzeniem obiektu składowego". Nowa klasa może być zbudowana z dowolnej liczby obiektów (mogą one także należeć do różnych typów) połączonych w dowolne kombinacje potrzebne do osiągnięcia pożądanych możliwości. Ponieważ tworzymy nowe klasy, używając klas już istniejących, dlatego koncepcja ta nazywa się *kompozycją* (ang. *composition*); jeśli do kompozycji dochodzi dynamicznie (w czasie wykonania programu), mówimy o niej jako o *agregacji* (ang. *aggregation*). Kompozycja jest często określana jako relacja typu „posiada", tak jak w zdaniu „samochód posiada silnik".



(Na powyższym diagramie języka UML kompozycja zaznaczona jest za pomocą wypełnionego rombu. Zwykle będziemy używać prostszej formy — kreski niezakończonej rombem — dla oznaczenia asocjacji[5]).

Kompozycja umożliwia wysoki stopień elastyczności. Obiekty składowe naszej nowej klasy są zwykle prywatne, a więc niedostępne dla używających jej programistów-klientów. Można zatem składowe te podmieniać, nie zakłócając istniejącego kodu klienta. Obiekty składowe można także wymieniać w czasie wykonania, co pozwala na dynamiczną zmianę zachowania programu. Opisane dalej dziedziczenie nie zapewnia takiej elastyczności, ponieważ ograniczenia na klasy utworzone za jego pomocą nakładane są już w czasie kompilacji.

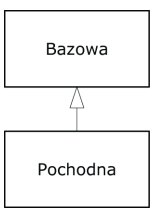
[5] Taki poziom szczegółowości jest wystarczający dla większości diagramów, zazwyczaj nie jest także konieczne rozróżnienie agregacji od kompozycji.

Ponieważ dziedziczenie jest tak istotne w programowaniu zorientowanym obiektowo, często jego znaczenie jest nadmiernie podkreślane, co może u początkującego programisty wytworzyć fałszywy pogląd, że należy je stosować wszędzie. W rezultacie tworzy on nieeleganckie i zbytnio skomplikowane programy. Przy tworzeniu nowych klas powinniśmy raczej w pierwszej kolejności rozważać użycie kompozycji, która jest prostsza i bardziej elastyczna. Dzięki takiemu rozwiązaniu projekty programów będą znacznie czytelniejsze. Zidentyfikowanie sytuacji wymagających użycia dziedziczenia jest, gdy już zdobędzie się pewne doświadczenie, stosunkowo proste.

Dziedziczenie

Idea obiektu jest sama w sobie wygodnym narzędziem. Pozwala połączyć ze sobą dane i zestawy funkcji w celu reprezentowania określonego *pojęcia* z przestrzeni problemu bez używania specyficznego języka maszyny. Pojęcia takie są wyrażane jako podstawowe jednostki w języku programowania dzięki użyciu słowa kluczowego `class`.

Niedobrze by było, gdyby po rozwiązaniu wszystkich problemów związanych z tworzeniem klasy trzeba było tworzyć od podstaw nową klasę, posiadającą podobną funkcję. Łatwiejsze byłoby „sklonowanie" istniejącej klasy i wykonanie na powstałej kopii wymaganych przeróbek i rozszerzeń. To właśnie efektywnie osiągamy poprzez wykorzystanie *dziedziczenia* (ang. *inheritance*) — z tym, że gdy oryginalna klasa (nazywana klasą *bazową* lub *nadklasą*) zostanie zmodyfikowana, wtedy nasz „klon" (nazywany klasą *potomną*, klasą *pochodną* lub *podklasą*) odzwierciedli również te zmiany.



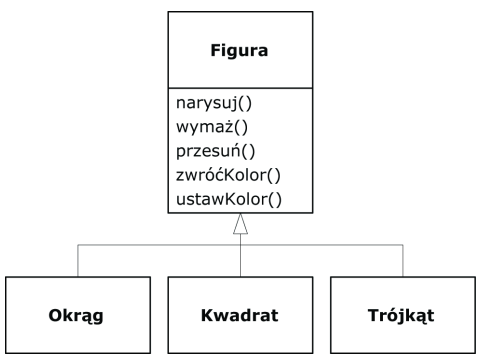
(Strzałka na powyższym diagramie UML jest skierowana od klasy pochodnej do bazowej. Jak później zobaczymy, klas pochodnych może być więcej).

Typ nie opisuje jedynie więzów nakładanych na pewien zbiór obiektów — wchodzi także w relacje z innymi typami. Dwa typy mogą posiadać wspólne cechy i zachowania, ale jeden może zawierać więcej danych niż drugi, lub obsługiwać więcej komunikatów (albo obsługiwać je inaczej). Dziedziczenie wyraża tego rodzaju podobieństwo między typami, używając pojęcia typów bazowych i typów pochodnych. Typ bazowy posiada te cechy i sposoby zachowania, które są wspólne dla wszystkich jego typów pochodnych. Tworzymy go do reprezentowania trzonu podstawowych idei, wspólnych dla pewnej grupy obiektów. Wywiedzione z niego typy pochodne reprezentują natomiast różne możliwości realizacji tych idei.

Rozważmy na przykład maszynę sortującą odpadki w celu odzysku surowców. Typem bazowym będzie „odpadek". Każdy odpadek posiada pewną wagę, wartość itd. Może być pocięty, przetopiony lub rozłożony na części. Dalej można wprowadzić bardziej

specyficzne typy śmieci — takie, które mają dodatkowe cechy (np. butelka posiada kolor) i sposoby zachowania (puszka aluminiowa może zostać zgnieciona, puszka stalowa reaguje na pole magnetyczne). Niektóre zachowania mogą dodatkowo ulec zmianie (np. wartość papieru zależy od jego rodzaju oraz stanu). Wykorzystanie dziedziczenia pozwala zbudować hierarchię klas wyrażającą rozwiązywany problem właśnie w kategoriach występujących w nim typów.

Jako drugi rozważymy klasyczny przykład z figurami geometrycznymi — może to być część systemu komputerowego wspomagania projektowania albo symulacji gry. Bazowym typem będzie tu „figura" posiadająca zawsze rozmiar, kolor, pozycję itp. Każda figura może zostać narysowana, wymazana, przesunięta, pokolorowana itd. Wywodzimy z niej (poprzez dziedziczenie) różne specyficzne typy figur: okręgi, kwadraty, trójkąty i inne, z których każdy może posiadać dodatkowe cechy i sposoby zachowania. Niektóre figury mogą być np. obrócone „do góry nogami". Niektóre zachowania mogą być różne, np. pole figury liczymy w różny sposób. Hierarchia typów wyraża zarówno podobieństwa, jak i różnice pomiędzy poszczególnymi rodzajami figur.



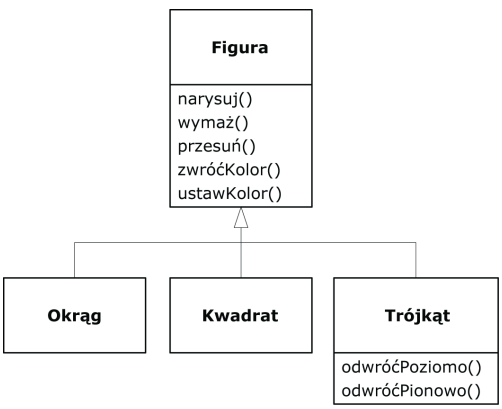
Opisanie rozwiązania w terminach problemu daje olbrzymie korzyści, ponieważ przejście od opisu problemu do opisu rozwiązania nie wymaga już zastosowania licznych modeli pośrednich. Gdy używamy obiektów, hierarchia klas stanowi model pierwotny, zatem przechodzimy wprost od opisu systemu w świecie rzeczywistym do opisu systemu w kodzie. Jednym z głównych problemów projektowania obiektowego jest zbyt prosta droga od początku do końca. Prostota często zbija początkowo z tropu umysły wytrenowane w poszukiwaniu skomplikowanych rozwiązań.

Poprzez dziedziczenie tworzymy nowy typ, który nie tylko zawiera wszystkie składowe (choć te zadeklarowane jako prywatne są ukryte i niedostępne), ale także, co ważniejsze, kopiuje interfejs klasy bazowej. A zatem wszystkie komunikaty, jakie można wysyłać do obiektów klasy bazowej, mogą być wysyłane również do obiektów klasy pochodnej. Ponieważ to zespół odbieranych komunikatów wyznacza typ, można powiedzieć, że *klasa pochodna jest tego samego typu co bazowa*. Odwołajmy się do naszego ostatniego przykładu: „Okrąg jest figurą". Ta osiągana przez dziedziczenie równoważność jest jednym z podstawowych etapów zrozumienia sensu programowania zorientowanego obiektowo.

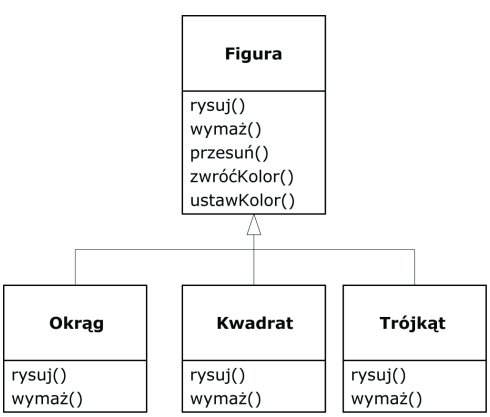
Ponieważ zarówno klasa bazowa, jak i pochodna mają ten sam interfejs, musi zatem istnieć jakaś związana z nim implementacja, tzn. musi istnieć kod wykonywany, gdy obiekt otrzyma określony komunikat. Jeżeli wszystko, co zrobimy, to stworzenie nowej klasy za

pomocą dziedziczenia, metody klasy bazowej stanowiące jej interfejs również przechodzą do klasy pochodnej — a zatem obiekty mają nie tylko ten sam typ co klasa bazowa, ale także zachowują się dokładnie tak samo, co nie jest szczególnie interesujące.

Istnieją dwa sposoby odróżnienia nowej klasy pochodnej od oryginalnej klasy bazowej. Pierwszy jest dosyć prosty: dodajemy do niej po prostu nowe metody. Nie są one częścią interfejsu klasy bazowej. Oznacza to, że klasa bazowa nie robiła wszystkiego, czego żądaliśmy, a zatem uzupełniliśmy jej zestaw metod. To proste i prymitywne użycie dziedziczenia jest, jak na razie, idealnym rozwiązaniem problemu. Warto jednakże zastanowić się, czy klasa bazowa nie potrzebuje tych dodatkowych metod. Taki proces iteracyjnego odkrywania właściwego projektu występuje często w programowaniu obiektowym.



Choć dziedziczenie wydaje się czasami (szczególnie w Javie, gdzie wprowadza się je za pomocą słowa kluczowego `extends`, czyli rozszerzać) sugerować konieczność dodania nowych metod do interfejsu, nie jest to do końca prawdą. Drugim — i do tego ważniejszym — sposobem, dzięki któremu możemy uczynić klasę pochodną różną od bazowej, jest *zmiana zachowania* istniejącej już metody bazowej, określana jako jej *przesłonięcie* (ang. *overriding*).

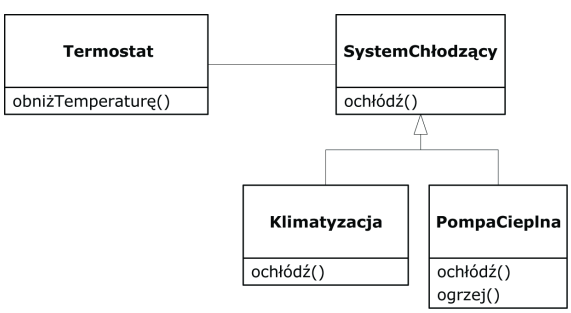


W celu przedefiniowania metody po prostu tworzymy w klasie pochodnej jej nową definicję. Mówimy: „Chcę tu wykorzystać tę samą metodę z interfejsu, ale dla nowego typu ma ona robić coś innego”.

„Bycie czymś” a „bycie podobnym do czegoś”

W kwestii dziedziczenia może powstać pewna wątpliwość: czy nie należałoby ograniczyć się *jedynie* do przedefiniowywania metody klasy bazowej (powstrzymując się od dodawania nowych)? Oznaczałoby to, że klasa pochodna jest *dokładnie* tego samego typu co bazowa, ponieważ ma taki sam interfejs. W rezultacie obiekt klasy potomnej jest dokładnym substytutem obiektu klasy bazowej — możemy mówić o *czystej zastępowalności*, a całe to stanowisko jest określane mianem *zasady zastępowalności* (ang. *substitution principle*). Jest to w pewnym sensie idealny sposób traktowania dziedziczenia. Związek pomiędzy klasą bazową a pochodną określamy w tym przypadku często jako relację *bycia czymś* (ang. *is-a relationship*), ponieważ możemy powiedzieć „okrąg *jest* figurą”. Sprawdzian poprawności użycia dziedziczenia polega na zbudowaniu podobnego zdania dla naszych klas i stwierdzeniu, czy ma ono sens.

W niektórych sytuacjach dodanie w typie pochodnym nowych elementów interfejsu jest jednak konieczne — takie jego rozszerzenie tworzy nowy typ. Może on być nadal używany zamiast typu bazowego, jednakże zastępowalność nie jest już idealna, ponieważ nowe funkcje nie są dostępne z poziomu bazowego. Ten rodzaj relacji możemy określić jako *bycie podobnym do czegoś*; nowy typ posiada cały interfejs starego typu, jednakże zawiera także inne metody, przez co nie można powiedzieć, że jest taki sam. Rozważmy na przykład klimatyzację. Przypuśćmy, że mamy zainstalowany w domu system chłodzący, a zatem interfejs pozwalający na kontrolę chłodzenia. Wyobraźmy sobie, że klimatyzacja psuje się i zastępujemy stary system nową pompą cieplną, która pozwala zarówno chłodzić, jak i ogrzewać. Zasada działania pompy cieplnej *jest podobna* do działania klimatyzacji, ale pompa może zrobić więcej od niej. Ponieważ system chłodzący jest zaprojektowany do kontroli chłodzenia, jest ograniczony do komunikowania się z chłodzącą częścią nowego obiektu. Interfejs tego obiektu został poszerzony, ale istniejący system nie zna niczego poza interfejsem oryginalnym.



Oczywiście po przestudiowaniu tego projektu staje się jasne, że klasa bazowa `SystemChłodzący` nie jest dostatecznie ogólna i powinna zostać zmieniona na „system kontroli temperatury”, aby mogła obsługiwać również ogrzewanie — po czym zasada zastępowalności zaczęłaby znowu działać. Diagram ten przedstawia jednak sytuację, jaka może się zdarzyć w projektowaniu i w świecie rzeczywistym.

Po zapoznaniu się z zasadą zastępowalności można pomyśleć, że to rozwiązanie (czysta zastępowalność) jest jedynym słusznym, i faktycznie dobrze jest, gdy projekt działa w ten sposób. Z czasem jednak mogą wystąpić sytuacje, w których konieczność dodania nowych funkcji w interfejsie klasy pochodnej jest równie jasna. Po dokładnym zbadaniu rozróżnienie między tymi przypadkami powinno być oczywiste.

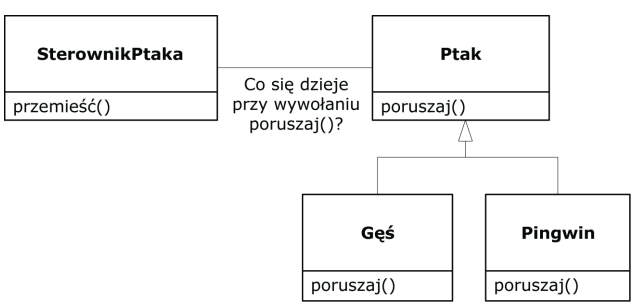
Wymienialność obiektów z użyciem polimorfizmu

Przy pracy z hierarchiami typów chcemy często traktować dany obiekt nie jako reprezentanta typu specjalizowanego, lecz raczej bazowego. Pozwala to na pisanie kodu niezależnego od konkretnego typu. W przykładzie z figurami funkcje manipulują nimi, nie zwracając uwagi na to, czy mają do czynienia z okręgami, kwadratami, trójkątami czy też takimi figurami, które nie zostały jeszcze nawet zdefiniowane. Wszystkie one mogą być narysowane, wymazane lub przesunięte, zatem funkcje te przesyłają po prostu komunikaty do obiektu typu figura, nie przejmują się sposobem, w jaki obiekt reaguje na te komunikaty.

Na kod taki nie ma wpływu dodawanie nowych typów, a dodawanie takie jest najpowszechniejszym sposobem rozszerzania programu obiektowego w celu obsłużenia nowych sytuacji. Możemy na przykład stworzyć nowy typ figury, zwany pięciokątem, nie modyfikując funkcji odnoszących się jedynie do ogólnych figur. Taka możliwość rozszerzania programu poprzez tworzenie nowych typów pochodnych jest ważnym sposobem hermetyzacji zmian, ponieważ znacząco ulepsza projekty, obniżając równocześnie koszty pielęgnacji oprogramowania.

Przy próbie traktowania typów pochodnych jako ich ogólnych typów podstawowych (np. okręgów jako figur, rowerów jako pojazdów, kormoranów jako ptaków itd.) powstaje jednak pewien problem. Jeżeli funkcja zamierza powiedzieć ogólnej figurze, aby się narysowała, ogólnemu pojazdowi, aby jechał lub ogólnemu ptakowi, aby się poruszył, kompilator nie może w czasie kompilacji określić, który kawałek kodu powinien zostać wykonany — funkcja rysowania może równie dobrze dotyczyć okręgu, kwadratu czy trójkąta, a obiekt wykona odpowiedni kod, wykorzystując swój właściwy, specyficzny typ.

Jeżeli nie musimy wiedzieć, który fragment kodu będzie wykonany, wtedy przy dodawaniu nowego podtypu kod przez niego wykonywany może być inny bez konieczności dokonywania zmian w wywołaniach metod. Kompilator nie może więc wiedzieć, który kawałek kodu zostanie wykonany. Co zatem robi? Na przykład na poniższym rysunku klasa `SterownikPtaka` pracuje wyłącznie z ogólnymi obiektami typu `Ptak`, nie znając ich konkretnych typów. Jest to wygodne z punktu widzenia klasy `SterownikPtaka`, ponieważ nie musi ona zawierać specjalnego kodu wyznaczającego dokładny typ lub zachowanie `Ptaka`, z jakim ma do czynienia. Jak więc się to dzieje, że choć `poruszaj()` wywoływane jest bez znajomości konkretnego typu `Ptaka`, to jednak ma miejsce właściwe zachowanie (`Gęś` biegnie, leci lub płynie, `Pingwin` biegnie lub płynie)?



Odpowiedzią jest podstawowa sztuczka programowania obiektowego: kompilator nie może wywołać funkcji w tradycyjny sposób. Wywołania funkcji generowane przez kompilatory języków niezorientowanych obiektowo używają tzw. *wczesnego wiązania* (ang. *early binding*) — terminu tego można nie znać, ponieważ wcześniej funkcji nie można było wywoływać inaczej. Znaczy to, że kompilator generuje wywołanie funkcji o określonej nazwie, program łączący (ang. *linker*) zamienia zaś to wywołanie na bezwzględny adres kodu, który ma zostać wykonany. W programowaniu obiektowym adres odpowiedniego fragmentu kodu nie może zostać wyznaczony aż do czasu wykonania, zatem przy wysyłaniu komunikatu do ogólnego obiektu konieczne jest inne rozwiązanie.

W celu rozwiązania tego problemu programowanie obiektowe wprowadza koncepcję *późnego wiązania* (ang. *late binding*). Przy wysyłaniu komunikatu do obiektu kod, który będzie wywołany, nie jest zdeterminowany aż do czasu wykonania. Kompilator upewnia się, że metoda istnieje, sprawdza typy argumentów i typ zwracanej wartości, nie wie jednak, jaki kod należy wykonać.

W celu przeprowadzenia późnego wiązania Java umieszcza zamiast bezwzględnego wywołania specjalny fragment kodu obliczający adres ciała metody na podstawie informacji przechowywanej w obiekcie (proces ten jest szczegółowo opisany w rozdziale 8., „Polimorfizm"). Każdy obiekt może zatem zachowywać się inaczej, w zależności od wyniku działania tego małego fragmentu kodu. Gdy wysyłamy komunikat do obiektu, obiekt decyduje, co z nim zrobić.

W niektórych językach musimy jawnie zadeklarować, że oczekujemy od funkcji elastyczności zapewnianej przez późne wiązanie (w języku C++ w tym celu wykorzystywane jest słowo kluczowe `virtual`). W językach tych wywołania funkcji składowych domyślnie *nie* są dynamicznie wiązane. Powodowało to problemy, dlatego w Javie dynamiczne wiązanie jest domyślne i nie musimy używać żadnych dodatkowych słów kluczowych, aby skorzystać z polimorfizmu.

Rozważmy przykład z figurami. Rodzina klas (bazujących na tym samym interfejsie) została przedstawiona nieco wcześniej na diagramie. Aby zademonstrować działanie polimorfizmu, chcielibyśmy napisać fragment programu ignorujący charakterystyczne dla typu szczegóły i wykorzystujący jedynie interfejs klasy bazowej. Kod ten będzie *niezależny* od informacji specyficznej dla typu, a przez to prostszy do napisania i łatwiejszy do zrozumienia. Poza tym, jeśli nowy typ, np. `Sześciokąt`, zostanie dodany poprzez dziedziczenie, kod, który napiszemy, będzie działał z tym nowym typem figury tak samo dobrze jak z istniejącymi wcześniej typami. Program jest zatem *rozszerzalny*.

Jeżeli napisze się w Javie (a niedługo nauczymy się to robić) następującą metodę:

```
void zróbCoś(Figura f) {
  f.wymaż();
  // ...
  f.narysuj();
}
```

komunikuje się ona z dowolną figurą, jest więc niezależna od konkretnego typu obiektu, który ma być rysowany i wymazywany. Jeżeli funkcję `zróbCoś()` wykorzystamy w jakimś innym fragmencie programu:

```
Okrąg o = new Okrąg();
Trójkąt t = new Trójkąt();
Linia l = new Linia();
zróbCoś(o);
zróbCoś(t);
zróbCoś(l);
```

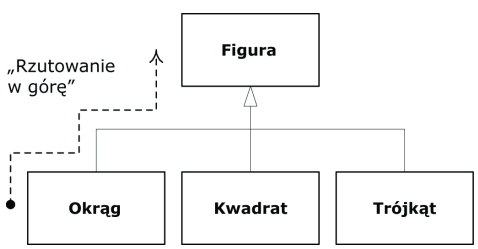
wywołania `zróbCoś()` będą automatycznie działać poprawnie, bez względu na dokładny typ obiektu.

W istocie jest to dosyć zaskakująca sztuczka. Rozważmy wiersz:

```
zróbCoś(o);
```

`Okrąg` jest przekazywany funkcji oczekującej argumentu typu `Figura`. Ponieważ `Okrąg` jest rodzajem figury, może więc być za taką uznawany przez metodę `zróbCoś()`. Znaczy to, że każdy komunikat, jaki `zróbCoś()` może wysłać do obiektu typu `Figura`, może być zaakceptowany przez `Okrąg`. To, co tutaj robimy, jest więc całkowicie logiczne i bezpieczne.

Proces taki, polegający na traktowaniu typu pochodnego tak, jakby był swoim typem bazowym, nazywamy *rzutowaniem w górę* (ang. *upcasting*). Rzutowanie używane jest tu w sensie dostosowywania do pewnej formy, *w górę* zaś odnosi się do sposobu, w jaki zwykle rysuje się diagramy dziedziczenia — z klasą bazową na szczycie i klasami pochodnymi rozwijającymi się w dół. Rzutowanie do typu bazowego oznacza zatem przesuwanie się w górę diagramu: jest więc rzeczywiście „rzutowaniem w górę".



Program zorientowany obiektowo zawiera w pewnych miejscach rzutowanie w górę, ponieważ w ten sposób uwalniamy się od konieczności znajomości dokładnego typu, z którym pracujemy. Spójrzmy na kod metody `zróbCoś()`:

```
f.wymaż();
  // ...
f.narysuj();
```

Jak widać, nie napisaliśmy tu: „Jeśli jesteś okręgiem, zrób to, jeśli jesteś kwadratem, zrób tamto, itd". Jeśli piszemy kod sprawdzający wszystkie typy, jakie może mieć `Figura`, staje się to nieeleganckie, a poza tym musi być zmieniane przy każdym dodaniu nowego rodzaju figury. W powyższym fragmencie mówimy po prostu: „Jeżeli jesteś figurą, to znaczy, że potrafisz wymazać się i narysować. Zrób więc to i zajmij się odpowiednio szczegółami".

W kodzie metody `zróbCoś()` imponuje fakt, że w jakiś sposób dzieje się to, co powinno. Wywołanie `narysuj()` dla obiektu `Okrąg` powoduje wykonanie innego kodu niż ten wykonywany przy wywołaniu tej samej metody dla `Kwadrat` albo `Linia`, jednak jeśli komunikat

`narysuj()` zostanie wysłany do anonimowej `Figury`, wtedy ma miejsce zachowanie poprawne dla rzeczywistego typu tej `Figury`. Jest to zadziwiające, ponieważ — jak już wspomniałem — kompilator Javy, kompilując `zróbCoś()`, nie wie, z jakimi dokładnie typami ma do czynienia. Normalnie oczekiwalibyśmy zatem, że użyje on wersji `narysuj()` i `wymaż()` zdefiniowanych dla bazowej klasy `Figura`, nie zaś specyficznych dla `Okrąg`, `Kwadrat` czy `Linia`. Mimo to dzięki polimorfizmowi dzieje się to, co powinno. Kompilator i system czasu wykonania zajmują się szczegółami — musimy jedynie wiedzieć, że to działa i, co istotniejsze, jak projektować z wykorzystaniem tego mechanizmu. Gdy wyślemy komunikat do obiektu, zrobi on to, co powinien, nawet jeśli użyte było rzutowanie w górę.

Hierarchia z pojedynczym korzeniem

Jedną z kwestii spornych, dotyczących programowania obiektowego, często poruszaną od czasu pojawienia się C++ jest to, czy wszystkie klasy powinny ostatecznie dziedziczyć po wspólnej klasie bazowej. W Javie (podobnie jak w niemal wszystkich językach obiektowych) odpowiedź brzmi „tak", nazwą tej wspólnej klasy bazowej jest zaś `Object`. Okazuje się, że hierarchia z pojedynczym korzeniem daje liczne korzyści.

W hierarchii z pojedynczym korzeniem wszystkie obiekty posiadają wspólny interfejs, więc w ostateczności wszystkie są tego samego typu. Alternatywa (dostarczana przez C++) polega na tym, że nie wiemy, czy wszystko ma ten sam fundamentalny typ. Z punktu widzenia wstecznej zgodności odpowiada to bardziej modelowi C, ale jest mniej ograniczające. Jednak jeśli chcemy programować w pełni obiektowo, musimy budować własną hierarchię, aby uzyskać taką samą wygodę, jaka jest wbudowana w inne języki obiektowe. Dodatkowo w każdej nowej bibliotece klas, jaką pozyskamy, użyty będzie jakiś inny, niezgodny interfejs. Dopasowanie tego nowego interfejsu do naszego projektu wymagać będzie wysiłku (a być może także wielokrotnego dziedziczenia). Czy dodatkowa „elastyczność" C++ jest tego warta? Jeżeli się jej potrzebuje — jeżeli zainwestowało się wiele w C — wtedy jest dosyć cenna. Jeżeli zaczynamy od zera, inne opcje, takie jak Java, mogą okazać się bardziej produktywne.

W hierarchii z pojedynczym korzeniem mamy gwarancję posiadania pewnych funkcji przez wszystkie obiekty. Wiemy, że pewne podstawowe operacje można wykonać na każdym obiekcie w systemie. Wszystkie obiekty można łatwo tworzyć w pamięci sterty, co znacząco uproszcza również przekazywanie argumentów.

Hierarchia o pojedynczym korzeniu w dużym stopniu ułatwia implementację *mechanizmu przywracania pamięci* (co jest jedną z ważniejszych zalet Javy wobec C++). Ponieważ informacja czasu wykonania o typie jest zagwarantowana w każdym obiekcie, nigdy nie pozostaniemy z obiektem, którego typu nie możemy określić. Jest to szczególnie istotne w przypadku operacji na poziomie systemu, takich jak obsługa wyjątków, oraz dla umożliwienia większej elastyczności w programowaniu.

Kontenery

Skoro zasadniczo nie wiemy, ile obiektów będzie nam potrzebnych do rozwiązania określonego problemu, ani jak długo obiekty te będą istnieć, nie wiemy również, w jaki sposób obiekty te przechowywać. Skąd mamy wiedzieć, ile miejsca dla nich przeznaczyć? Ta informacja jest niedostępna aż do czasu wykonania programu.

Rozwiązania większości problemów w projektowaniu obiektowym wydają się dziwnie proste: tworzymy nowy typ obiektów. W przypadku omawianego problemu tym nowym typem jest obiekt przechowujący referencje do innych obiektów. Moglibyśmy oczywiście uzyskać analogiczny efekt, używając *tablic* dostępnych w niemal każdym języku programowania. Dostajemy jednak więcej: nowy obiekt, zwany *kontenerem* (lub *kolekcją*, lecz słowo to jest używane w Javie w innym znaczeniu, w związku z tym w tej książce będzie używany termin „kontener"), będzie rozszerzał się, gdy będzie to konieczne, by przyjąć wszystko, co do niego wstawimy. Nie musimy więc wiedzieć, ile obiektów będzie przechowywanych w kontenerze. Po prostu tworzymy go i pozwalamy mu zająć się szczegółami.

Na szczęście każdy dobry język obiektowy dostarczany jest z zestawem kontenerów. W C++ zestaw taki jest częścią biblioteki standardowej i określany jest czasami akronimem STL (*Standard Template Library*). W języku Object Pascal kontenery stanowią część biblioteki VCL (*Visual Component Library*). Smalltalk posiada bardzo rozbudowaną bibliotekę kontenerów. Także Java posiada kontenery w swej bibliotece standardowej. W niektórych bibliotekach ogólny kontener uważany jest za wystarczająco dobry do wszystkich zastosowań, w innych zaś (np. w Javie) istnieją różne typy kontenerów dla różnych potrzeb: kilka różnych rodzajów klas `List` (służących do przechowywania sekwencji), klasy `Map` (zwane także *tablicami asocjacyjnymi*, które kojarzą jedne obiekty z innymi), klasy `Set` (do przechowywania niepowtarzających się elementów). Biblioteki kontenerów mogą również zawierać: kolejki, drzewa, stosy itd.

Z projektowego punktu widzenia wszystko, czego potrzebujemy, to kontener, którym można się posługiwać w celu rozwiązania problemu. Gdyby pojedynczy typ kontenera wystarczał, nie byłoby potrzeby posiadania różnych ich rodzajów. Istnieją dwa powody, dla których potrzebny jest wybór między kontenerami. Po pierwsze, kontenery posiadają różne interfejsy i zewnętrzne zachowania. Zachowanie się i interfejs stosu są różne od tych zapewnianych przez kolejkę, która z kolei różni się od zbioru czy listy. Określony interfejs może zapewniać bardziej elastyczne rozwiązanie naszego problemu niż inny. Po drugie, koszt wykonania tych samych operacji na różnych kontenerach jest inny. Najlepszy przykład stanowią `ArrayList` i `LinkedList`. Obie te klasy reprezentują proste listy elementów, mają identyczne interfejsy i cechują się podobnym zachowaniem zewnętrznym. Jednak określone operacje mogą mieć dla nich zasadniczo rózny koszt. Dostęp swobodny do elementów w klasie `ArrayList` jest operacją wykonywaną w czasie stałym — trwa tak samo długo bez względu na to, do którego elementu się odwołujemy. Jednakże w przypadku `LinkedList` wybranie określonego elementu wymaga przejścia przez elementy poprzedzające go na liście, co jest operacją tym kosztowniejszą, im dalej na liście znajduje się element. Z drugiej strony, dodanie elementu w środku listy jest

znacznie tańsze w wypadku listy powiązanej (`LinkedList`). Te i inne operacje mają zróżnicowaną efektywność w zależności od wewnętrznej implementacji listy. W fazie projektowania moglibyśmy zacząć od wykorzystania listy powiązanej, po czym przy dostrajaniu efektywności przestawić się na wektor (`ArrayList`). Dzięki zapewnianej przez iteratory abstrakcji wpływ takiej zmiany na kod byłby minimalny.

Typy parametryzowane (typy ogólne)

W wersjach języka poprzedzających wydanie Java SE5 kontenery przechowywały elementy jednego, uniwersalnego typu: `Object`. Hierarchia z pojedynczym korzeniem implikuje, że wszystkie obiekty są reprezentantami klasy `Object`, a zatem kontenery mogące przechowywać obiekty tej klasy mogą przechowywać cokolwiek[6]. To właśnie dzięki temu kontenery nadają się do tak powszechnego stosowania.

Używając takiego kontenera, wstawiamy do niego po prostu referencje do obiektów, a później prosimy o ich zwrot. Jednak, skoro kontener przechowuje obiekty klasy `Object`, przy wstawianiu do kontenera dokonuje się rzutowania na tę klasę, przez co tracimy informację o konkretnym typie wstawianego obiektu. Wyjmując obiekt, otrzymujemy również referencję do typu `Object`, a nie do oryginalnego typu włożonego obiektu. Jak zatem zamienić ją na powrót na coś o interfejsie obiektu, który wstawiliśmy do kontenera?

I w tym przypadku potrzebujemy rzutowania, jednak nie będzie to już rzutowanie w górę hierarchii dziedziczenia dla uzyskania bardziej ogólnego typu — rzutujemy teraz w dół w celu uzyskania typu specjalizowanego. Ten rodzaj rzutowania nazywamy właśnie *rzutowaniem w dół* (ang. *downcasting*). Gdy rzutujemy w górę, wiemy na przykład, że „okrąg" jest rodzajem „figury", tak więc rzutowanie jest tu zawsze bezpieczne. Nie możemy jednak z góry przewidzieć, że dany obiekt typu `Object` jest z pewnością „okręgiem" czy „figurą" — trudno więc uznać ten rodzaj rzutowania za bezpieczny, chyba że w jakiś sposób poznamy właściwy typ obiektu, z którym mamy do czynienia.

Sytuacja nie jest jednak bardzo niebezpieczna — dokonując rzutowania na niewłaściwy typ, spowodujemy wystąpienie błędu czasu wykonania, zwanego *wyjątkiem* (ang. *exception*) — o wyjątkach dowiemy się więcej już niebawem. Mimo to przy wyjmowaniu z kontenera referencji do obiektów musimy, w celu dokonania prawidłowego rzutowania, pamiętać, jakiego właściwie te obiekty są typu.

Rzutowanie w dół, w połączeniu z koniecznością wykonywania testów czasu wykonania, powoduje pewien narzut czasowy w działającym programie oraz wymaga dodatkowego wysiłku od programisty. Można zapytać, czy nie dałoby się w jakiś sposób stworzyć kontenera „znającego" typ przechowywanych przez siebie obiektów i wyeliminować w ten sposób konieczność rzutowania oraz możliwość popełnienia błędów? Rozwiązaniem są *typy parametryzowane* (ang. *parametrized types*) — klasy, które kompilator automatycznie dostosowuje do pracy z podanymi typami-parametrami. Na przykład parametryzowany kontener może zostać przystosowany tak, aby akceptował tylko obiekty klasy `Figura` i zwracał także tylko `Figury`.

[6] Nie dotyczyło to typów elementarnych; mechanizm pakowania wartości elementarnych (skalarnych) w obiektach (ang. *autoboxing*) wprowadzony w Javie SE5 wyeliminował to ograniczenie niemal całkowicie. Wrócimy do tego w dalszej części książki.

Jedną z ważniejszych nowości w Javie SE5 jest właśnie obecność typów parametryzowanych, zwanych też *typami ogólnymi* (ang. *generics*). Parametryzację rozpoznajemy po obecności nawiasów kątowych przy nazwie klasy, z listą typów parametryzujących daną klasę. Na przykład obiekt klasa `ArrayList` (klasy kontenera) w wersji przystosowanej do przechowywania obiektów klasy `Figura` tworzy się tak:

```
ArrayList<Figura> figury = new ArrayList<Figura>();
```

Parametryzacja typów spowodowała liczne zmiany wielu komponentów biblioteki standardowej języka Java. Jak się niebawem okaże, dostępność typów ogólnych wpłynie na większość kodów przykładowych prezentowanych w tej książce.

Tworzenie obiektów i czas ich życia

Każdy obiekt, aby mógł istnieć, wymaga zasobów — przede wszystkim pamięci. Kiedy obiekt przestaje być potrzebny, należy po nim „posprzątać", czyli zwolnić przydzielone mu zasoby, aby można je było powtórnie wykorzystać. Zazwyczaj problem „posprzątania" po obiekcie nie wydaje się być szczególnym wyzwaniem — obiekt jest tworzony, stosowany tak długo, jak jest potrzebny, a następnie należy go usunąć. Jednak nietrudno spotkać sytuacje znacznie bardziej skomplikowane.

Załóżmy, że tworzymy system obsługi ruchu powietrznego dla lotniska. (Tego samego modelu można by także używać do zarządzania towarami w magazynie, w systemie wypożyczania kaset wideo lub firmie zajmującej się tresurą zwierząt). Na pierwszy rzut oka zadanie wydaje się proste: trzeba stworzyć kontener służący do przechowywania samolotów, a następnie umieścić w nim każdy samolot znajdujący się w kontrolowanym obszarze przestrzeni powietrznej. W ramach sprzątnięcia należy usunąć obiekt samolotu, który opuścił kontrolowany obszar.

Być może jednak dysponujemy innym systemem do rejestracji danych o samolotach; być może nie są to dane wymagające tak natychmiastowej uwagi, jak główne funkcje sterowania lotami. Być może jest to zapis planu lotów wszystkich małych samolotów startujących z lotniska. A zatem mamy drugi kontener dla małych samolotów i za każdym razem, gdy tworzony jest obiekt samolotu, jeśli jest to samolot mały, zapisujemy go także w tym drugim kontenerze. Następnie, podczas okresów bezczynności systemu, jakiś proces działający w tle wykonuje na tym kontenerze pewne operacje.

Teraz problem się skomplikował: w jaki sposób określić, kiedy można usunąć obiekt? Gdy obiekt nie jest już potrzebny w jednej części systemu, inne wciąż mogą z niego korzystać. Ten sam problem może się pojawiać w wielu różnych sytuacjach, a w systemach programistycznych wymagających jawnego usuwania obiektów (takich jak C++) może on stać się bardzo złożony.

Gdzie znajdują się dane obiektu i jak jest kontrolowany czas jego życia? Język C++ przyjmuje założenie, że najważniejszą sprawą jest kontrola efektywności, dlatego daje programiście wybór. W celu osiągnięcia maksymalnej szybkości czasu wykonania może on określić sposów przechowywania i czas życia obiektu na etapie pisania programu poprzez umieszczenie go na stosie (mówi się wtedy czasami o zmiennych *automatycznych*

albo *lokalnych*) albo w obszarze pamięci statycznej. Kładzie się w ten sposób nacisk na szybkie rezerwowanie i zwalnianie miejsca, poświęcając jednakże elastyczność, ponieważ musimy znać dokładną liczbę, czas życia oraz typ obiektów na etapie pisania programu. Jeżeli próbujemy rozwiązać bardziej ogólny problem, taki jak komputerowe wspomaganie projektowania, zarządzanie magazynem lub kontrola ruchu powietrznego, wtedy staje się to zbyt poważnym ograniczeniem.

Drugie rozwiązanie polega na dynamicznym tworzeniu obiektów w obszarze pamięci zwanym *stertą*. Stosując to rozwiązanie, nie wiemy aż do czasu wykonania, ilu obiektów potrzebujemy, jaki ma być czas ich życia oraz dokładny typ. Kwestie te są rozstrzygane w odpowiednim momencie podczas działania programu. Jeżeli potrzebujemy nowego obiektu, tworzymy go po prostu na stercie. Ponieważ pamięć jest zarządzana dynamicznie w czasie wykonania, zatem okres niezbędny do jej zarezerwowania na stercie jest o wiele dłuższy niż ten potrzebny do zrobienia tego samego na stosie (zarezerwowanie miejsca na stosie wiąże się często jedynie z pojedynczą instrukcją asemblerową do przesunięcia wskaźnika stosu w dół oraz drugą do przesunięcia go z powrotem w górę; czas konieczny do utworzenia miejsca na stercie zależy natomiast od konstrukcji mechanizmu obsługi pamięci). Czas dynamicznego przydziału w pamięci sterty jest zaś zależny od szczegółów mechanizmu obsługi pamięci w danym systemie.

Rozwiązanie dynamiczne bazuje na słusznym założeniu, że obiekty mogą być skomplikowane, a zatem narzut związany z samym znajdowaniem i zwalnianiem miejsca na te obiekty nie będzie miał istotnego wpływu na tworzenie obiektu. Ponadto zwiększona elastyczność tego rozwiązania ma podstawowe znaczenie dla rozwiązywania ogólnych problemów programistycznych.

Java stosuje wyłącznie drugie rozwiązanie[7]. Za każdym razem, gdy tworzymy obiekt, używamy słowa kluczowego `new`, aby utworzyć jego dynamiczny egzemplarz.

Inną sprawę stanowi czas życia obiektu. W językach pozwalających na tworzenie obiektów na stosie kompilator wyznacza czas trwania obiektu i może go automatycznie zniszczyć. Jeżeli jednak obiekt zostanie stworzony na stercie, wtedy kompilator nie zna czasu jego życia. W języku takim jak C++ musimy programowo określić, kiedy należy zniszczyć obiekt, co prowadzi do *wycieków pamięci*, jeśli nie robi się tego poprawnie (a jest to częsty przypadek w programach C++). Java posiada udogodnienie zwane *odśmiecaczem pamięci*, automatycznie wykrywającym, który obiekt nie jest już używany, a następnie niszczącym go. Odśmiecacz (ang. *garbage collector*) jest rozwiązaniem znacznie wygodniejszym, ponieważ redukuje liczbę zdarzeń, które musimy śledzić, oraz ilość kodu, jaki musimy napisać. Co ważniejsze, stanowi on znacznie wyższy stopień zabezpieczenia przed problemem wycieków pamięci (który zniweczył wiele projektów w C++).

W Javie problem zwalniania pamięci został oddelegowany do specjalnie zaprojektowanego odśmiecacza pamięci (choć nie obejmuje on pozostałych aspektów „sprzątania" po obiektach). Odśmiecacz „wie", kiedy obiekt nie jest już potrzebny, i zwalnia przydzieloną mu pamięć. Dzięki temu (oraz ze względu na fakt, że wszystkie obiekty dziedziczą po jednej klasie bazowej — `Object`, a wszystkie obiekty można tworzyć tylko w jeden sposób — na stercie) proces programowania w Javie jest znacznie prostszy niż w C++. Mniej tu decyzji do podjęcia i problemów do rozwiązania.

[7] Specjalny przypadek stanowią typy podstawowe, które poznamy później.

Obsługa wyjątków — eliminowanie błędów

Od początków istnienia języków programowania obsługa błędów była jednym z najtrudniejszych zadań. Z powodu trudności, jakich nastręcza zaprojektowanie dobrego schematu takiej obsługi, wiele języków po prostu ignoruje to zagadnienie, zrzucając odpowiedzialność na projektantów bibliotek. Proponują oni zwykle półśrodki działające poprawnie w wielu sytuacjach, będące jednak łatwe do obejścia (najczęściej po prostu przez ich zignorowanie). Podstawowym problemem większości schematów obsługi błędów jest to, że ich podstawą jest gotowość programisty do przestrzegania pewnej ustalonej konwencji, nieobsługiwanej w żaden sposób przez język. Gdy programista nie ma na to ochoty, a zdarza się to często, np. gdy się spieszy, może z łatwością o konwencji zapomnieć.

Mechanizm *obsługi wyjątków* (ang. *exception handling*) wiąże bezpośrednio obsługę błędów z językiem programowania, a czasem wręcz z systemem operacyjnym. Wyjątek jest obiektem, który jest „wyrzucany" z miejsca wystąpienia błędu, a następnie może zostać „przechwycony" przez odpowiednią *procedurę obsługi wyjątku* (ang. *exception handler*) — zaprojektowaną specjalnie do radzenia sobie z danym typem błędów. Obsługa wyjątków jest alternatywną ścieżką przepływu sterowania, wybieraną w przypadku, gdy coś pójdzie nie tak — dzięki temu nie musi się mieszać z kodem wykonywanym w normalnej sytuacji. Czyni to ten ostatni znacznie prostszym do napisania, ponieważ nie trzeba bez przerwy sprawdzać, czy nie wystąpiły jakieś błędy. Zgłaszanie („wyrzucanie") wyjątku różni się od ustawiania znacznika błędu czy zwracania ustalonej (oznaczającej błąd) wartości także tym, że w przeciwieństwie do nich nie może zostać zignorowane — mamy zatem gwarancję, że wyjątek zostanie w pewnym momencie obsłużony. Wyjątki dostarczają także godny zaufania sposób wyjścia ze złej sytuacji. Zamiast po prostu przerwać działanie programu, można często przywrócić warunki do jego dalszego wykonywania — napisane w ten sposób programy są znacznie solidniejsze.

Pod względem obsługi wyjątków Java znacząco różni się od innych języków programowania, gdyż mechanizmy te zostały wbudowane w język a programiści są zmuszeni do korzystania z nich. Jeśli tworzony kod nie będzie w poprawny sposób obsługiwać wyjątków, podczas jego kompilacji pojawią się błędy. Ta gwarantowana konsekwencja czasami może ułatwić obsługę błędów.

Warto zaznaczyć, że choć w językach zorientowanych obiektowo wyjątek jest zwykle reprezentowany przez obiekt, to jednak sam mechanizm obsługi wyjątków nie jest cechą przynależną jedynie tym językom — powstał od nich wcześniej.

Współbieżność

Jedną z podstawowych koncepcji programowania jest pomysł wykonywania kilku zadań w tym samym czasie. Liczne problemy programistyczne wymagają, aby program był w stanie przerwać swą bieżącą aktywność, załatwić jakąś inną sprawę i powrócić do głównego zadania. Próbowano wielu rozwiązań. Początkowo programiści posiadający

niskopoziomową wiedzę na temat maszyny pisali procedury obsługi przerwań, a zawieszanie głównego programu było inicjowane przez przerwania sprzętowe. Choć działało to zadowalająco, było jednak skomplikowane i nieprzenośne, czyniąc proces przenoszenia programu na nowy typ maszyny powolnym i kosztownym.

Przerwania są czasami niezbędne dla obsługi zadań o krytycznych ograniczeniach czasowych, jednakże w przypadku dużej klasy problemów chodzi nam o podzielenie zadania na wykonujące się niezależnie fragmenty, dzięki czemu program jako całość staje się bardziej efektywny. Wewnątrz programu te niezależnie wykonujące się fragmenty nazywamy wątkami, a cała idea nosi nazwę *współbieżności* (ang. *concurrency*) lub *wielowątkowości* (ang. *multithreading*). Typowym przykładem wykorzystania wielowątkowości jest interfejs użytkownika. Dzięki wątkom użytkownik może otrzymać szybką reakcję na naciśnięcie przycisku, nie będąc zmuszonym do oczekiwania na zakończenie wykonywania przez program jego aktualnego zadania.

Wątki są zwykle jedynie sposobem podziału czasu pojedynczego procesora. Jeżeli jednak system operacyjny obsługuje wiele procesorów, wtedy każdy wątek może zostać przydzielony innemu z nich, dzięki czemu będą się one wykonywać rzeczywiście równolegle. Jedną z zalet wielowątkowości na poziomie języka jest to, że programista nie musi interesować się, czy ma do dyspozycji wiele procesorów czy też tylko jeden. Program jest logicznie podzielony na wątki i jeżeli maszyna posiada wiele procesorów, wtedy działa szybciej bez żadnych poprawek.

Powyższe stwierdzenia mogą sugerować, że wielowątkowość jest prostym zagadnieniem. Jest jednak pewna pułapka — zasoby wspólne. Jeżeli więcej niż jeden wątek ma zamiar odwoływać się do tego samego zasobu, wtedy powstaje problem. Na przykład dwa procesy nie mogą równocześnie wysyłać informacji na drukarkę. W celu rozwiązania tego problemu zasoby, które mogą (tak jak drukarka) być dzielone, muszą być blokowane na czas użycia. A zatem wątek blokuje zasób, wykonuje swoje zadanie, po czym zwalnia blokadę, aby umożliwić innym wątkom skorzystanie z zasobu.

Wielowątkowość jest w Javie częścią języka; w Java SE5 doczekała się też istotnego wsparcia bibliotecznego.

Java i Internet

Jeśli tak naprawdę Java jest jeszcze jednym językiem programowania, można zapytać, czemu wobec tego jest tak ważna i dlaczego jest określana jako rewolucyjny krok w programowaniu? Z punktu widzenia tradycyjnego programowania odpowiedź nie jest od razu oczywista. Java jest bardzo użyteczna przy rozwiązywaniu tradycyjnych problemów programistycznych i rozwiązuje również problemy programowania dla sieci World Wide Web.

Czym jest sieć WWW?

Początkowo sieć WWW może się wydawać tajemnicza, razem z tymi wszystkimi surfowaniami, prezentacjami i stronami domowymi. Może warto przyjrzeć się z boku temu, czym tak naprawdę jest WWW? Jednak aby to zrobić, należy zrozumieć model systemów typu klient-serwer — kolejny pełen nieporozumień aspekt informatyki.

Przetwarzanie typu klient-serwer

Podstawą systemu klient-serwer jest założenie, że mamy centralne repozytorium informacji (jakiegoś rodzaju danych, często zapisanych w bazie danych), które chcemy rozsyłać na żądanie do pewnej grupy ludzi lub maszyn. Kluczem modelu klient-serwer jest centralne położenie składnicy informacji. Jeśli informacje zostaną zmienione, wszystkie zmiany zostaną przekazane ich odbiorcom. Całość: składnica informacji, oprogramowanie rozsyłające informacje oraz maszyna, na której znajduje się informacja, i jej oprogramowanie nazywane jest serwerem. *Klientem* nazywa się oprogramowanie znajdujące się na maszynie zdalnej, które komunikuje się z serwerem, pobiera informacje, przetwarza je i wyświetla na tej maszynie.

Podstawowa idea przetwarzania typu klient-serwer nie jest specjalnie skomplikowana. Problemy wynikają z tego, że mamy jeden serwer próbujący jednocześnie obsłużyć wiele klientów. Ogólnie, projektant wykorzystując systemy zarządzania bazami danych, stara się równomiernie rozmieścić dane w tabelach, aby osiągnąć jak największą użyteczność systemu. Dodatkowo systemy często pozwalają klientom na wprowadzanie nowych informacji do serwera. Oznacza to, że trzeba pilnować, aby nowe dane jednego klienta nie zamazały nowych danych innego lub żeby dane nie zaginęły w procesie dodawania ich do bazy (nazywa się to przetwarzaniem transakcyjnym). Kiedy oprogramowanie klienta się zmienia, musi być kompilowane, testowane i instalowane na maszynach klientów, co okazuje się bardziej skomplikowane i kosztowne, niż mogłoby się wydawać. Wielu problemów przysparza obsługa różnych typów komputerów i systemów operacyjnych. Na dodatek aktualny pozostaje problem wydajności: w każdej chwili mogą potencjalnie istnieć setki żądań skierowanych przez klientów do serwera, a zatem każde opóźnienie jest krytyczne. Aby ograniczyć te opóźnienia, programiści ciężko pracują nad tym, aby przenieść przetwarzane zadania na maszynę klienta, a czasem na inne maszyny po stronie serwera, składające się na tak zwaną *warstwę pośrednią* (ang. *middleware*). Warstwa pośrednia poprawia również możliwość konserwacji systemu.

Prosta idea rozsyłania ludziom informacji ma tak wiele poziomów złożoności, że cały problem może się wydawać bardzo zagmatwany. Jednocześnie jest bardzo ważny: przetwarzanie typu klient-serwer stanowi połowę wszystkich przedsięwzięć programistycznych. Odpowiada za wszystko, począwszy od przyjmowania zamówień i transakcji przy użyciu kart kredytowych, skończywszy na rozpowszechnianiu dowolnego typu danych — giełdowych, naukowych, rządowych. W przeszłości pojawiały się tylko indywidualne rozwiązania dla indywidualnych problemów — za każdym razem wynajdywano nowe. Były one trudne do tworzenia i używania, i za każdym razem użytkownik musiał się nauczyć obsługi nowego interfejsu. Problem klient-serwer wymaga rozwiązań na dużą skalę.

Sieć WWW jako gigantyczny serwer

Sieć WWW jest właściwie gigantycznym systemem typu klient-serwer. Nawet więcej — ponieważ wszystkie serwery i klienty współistnieją jednocześnie w tej samej sieci. Nie trzeba o tym wiedzieć, gdyż wszystko, o co trzeba się zatroszczyć w danym momencie, to połączenie i interakcja z jednym tylko serwerem (ale aby go znaleźć, można przeszukać pół świata).

Początkowo był to prosty proces jednokierunkowy. Na żądanie przesłane do serwera przekazywał on z powrotem plik, który był interpretowany i formatowany przez przeglądarkę na maszynie lokalnej klienta. Wkrótce ludzie zaczęli żądać czegoś więcej niż tylko dostarczania stron z serwera. Chcieli pełnych możliwości klient-serwer, tak aby klient mógł odsyłać informacje do serwera, na przykład zażądać konkretnych danych z serwera, dodać nowe informacje lub złożyć zamówienie (co wymagało większych zabezpieczeń niż te oferowane przez oryginalny system). Są to zmiany, jakie zaszły w rozwoju WWW.

Przeglądarka WWW — a właściwie sam pomysł, że ta sama informacja może być wyświetlona na komputerze dowolnego typu bez zmieniania jej — była wielkim krokiem naprzód. Jednakże przeglądarki były nadal prymitywne i nie spełniały stawianych im żądań. W szczególności nie były interaktywne i miały skłonność do blokowania zarówno serwerów, jak i Internetu, bowiem za każdym razem, kiedy konieczne było wykonanie czegoś, co wymagało przetwarzania, trzeba było wysłać informację z powrotem do serwera, aby dopiero tam została przetworzona. Mogło upłynąć wiele sekund lub minut, nim okazało się, że żądanie było błędnie sformułowane. Przeglądarka służyła jedynie do oglądania, więc nie mogła wykonywać nawet najprostszych zadań obliczeniowych (z drugiej strony była bezpieczna, ponieważ nie mogła na lokalnej maszynie wykonywać żadnych programów, które mogłyby zawierać błędy lub wirusy).

Aby rozwiązać ten problem, przyjęto kilka rozwiązań. Na początek standardy graficzne poszerzono tak, by umożliwić lepszą animację obrazu w przeglądarkach. Rozwiązaniem pozostałych problemów jest umożliwienie przeglądarce uruchamiania programów po stronie klienta. Nazywamy to *programowaniem po stronie klienta*.

Programowanie po stronie klienta

Pierwotny projekt architektury sieciowej typu serwer-przeglądarka zapewniał interaktywną zawartość, ale była ona w całości dostarczana przez serwer. Serwer produkował statyczne strony dla przeglądarki klienta, która je po prostu interpretowała i wyświetlała. Podstawowy HTML zawiera proste mechanizmy pobierania danych: pola tekstowe, pola wyboru, pola wielokrotnego wyboru, listy i listy rozwijane oraz przyciski, które można zaprogramować tylko do wyczyszczenia formularza lub wysłania (ang. *submit*) danych formularza z powrotem do serwera. To żądanie przechodzi przez *Common Gateway Interface* (uniwersalny interfejs komunikacyjny, w skrócie CGI) dostarczany przez wszystkie serwery WWW. Tekst wewnątrz żądania mówi CGI, co należy z nim zrobić. Najczęstszym działaniem jest wykonanie programu znajdującego się na serwerze w katalogu nazywanym zwykle *cgi-bin* (obserwując okienko adresu na górze przeglądarki, kiedy naciskasz przycisk na stronie sieciowej, możesz czasem zobaczyć *cgi-bin* pomiędzy innymi znajdującymi się tam śmieciami). Programy te mogą być pisane w większości języków. Najczęściej wybierany jest Perl, ponieważ jest przeznaczony do manipulowania tekstem, a programy w nim napisane są wykonywane przez interpreter, więc może być instalowany na dowolnych serwerach, niezależnie od procesora i systemu operacyjnego. Coraz większą popularność zdobywa jednak Python (mój ulubiony — patrz *www.Python.org*), głównie ze względu na swoje ogromne możliwości i prostotę.

Obecnie wiele dużych witryn internetowych zbudowano wyłącznie z wykorzystaniem CGI, za pomocą którego można zrobić praktycznie wszystko. Jednakże utrzymanie witryn zbudowanych na programach CGI może szybko stać się bardzo skomplikowane.

Problemem jest również długi czas reakcji. Odpowiedź programu CGI zależy od liczby danych, które muszą zostać przesłane, oraz od obciążenia serwera i Internetu (poza tym uruchamianie programu CGI jest na ogół wolne). Pierwsi projektanci Internetu nie przewidzieli, że jego przepustowość zostanie tak gwałtownie wyczerpana przez różnorodne aplikacje. Przykładowo, nie da się zrealizować procesu dynamicznego tworzenia grafiki, ponieważ plik GIF (*Graphics Interchange Format*) musi zostać stworzony i przesłany od serwera do klienta dla każdej wersji obrazu. Każdy bez wątpienia miał bezpośredni kontakt z czymś tak prostym, jak sprawdzenie danych w formularzu. Po naciśnięciu przycisku „Zatwierdź" dane są przesyłane z powrotem do serwera. Serwer uruchamia program CGI, który wykrywa błąd, formułuje stronę HTML informującą o błędzie, a następnie odsyła ją z powrotem. Trzeba wtedy cofnąć się do poprzedniej strony i spróbować ponownie. Jest to nie tylko wolne, ale i nieeleganckie.

Rozwiązaniem jest programowanie po stronie klienta. Większość komputerów, na których pracują przeglądarki, to potężne maszyny, będące w stanie wykonać ogromną ilość obliczeń, a przy pierwotnym sposobie pracy statycznego HTML-a czekające bezczynnie, aż serwer poda im następną stronę. Programowanie po stronie klienta sprawia, że przeglądarka WWW zostaje zaprzęgnięta do pracy, a dla użytkownika oznacza to o wiele bogatsze, interaktywne przeżycia podczas korzystania z takiej witryny.

Przy omawianiu programowania po stronie klienta problemem jest to, że nie różni się ono jakoś szczególnie od programowania w ogóle. Parametry są niemalże takie same, ale inna jest platforma: przeglądarka internetowa jest jak ograniczony system operacyjny. W końcu nadal trzeba programować, a to wiąże się z zawrotną liczbą problemów i rozwiązań stwarzanych przez programowanie po stronie klienta. Reszta tego podrozdziału przedstawia przegląd kwestii związanych z programowaniem po stronie klienta.

Moduły rozszerzające

Jednym z bardziej znaczących kroków w kierunku programowania po stronie klienta jest stworzenie modułów rozszerzających (ang. *plug-in*). Jest to sposób, w jaki program może dodać nowe funkcje do przeglądarki przez ściągnięcie kawałka kodu, który jest dołączany w odpowiednie miejsce przeglądarki. Mówi on przeglądarce: „od tej pory możesz wykonywać takie a takie nowe czynności" (moduł rozszerzający ładuje się jednorazowo). Dzięki rozszerzeniom przeglądarka zostaje wzbogacona o potężne możliwości, jednak pisanie ich nie jest zadaniem trywialnym i zapewne nikt nie chciałby tego robić w procesie budowania konkretnej strony. Moduły rozszerzające mają dużą wartość dla programowania klient-serwer, ponieważ pozwalają doświadczonemu programiście na stworzenie nowego języka programowania i dodanie go do przeglądarki bez konieczności uzyskania zgody jej twórcy. Zatem rozszerzenia dostarczają dodatkowe metody tworzenia nowych języków programowania po stronie klienta (jednak nie wszystkie języki są implementowane jako moduły rozszerzające).

Języki skryptowe

Wprowadzenie modułów rozszerzających zaowocowało eksplozją języków skryptowych. Kod źródłowy programu napisanego w języku skryptowym i działającego po stronie klienta jest osadzony wewnątrz strony HTML. Moduł interpretujący ten kod jest automatycznie aktywowany w momencie wyświetlania takiej strony. Języki skryptowe są na ogół łatwe

do zrozumienia i jako tekst są częścią kodu HTML. Program w takim języku jest tekstem stanowiącym fragment strony HTML, dlatego ładuje się z serwera jako część tego samego żądania, które wyświetla stronę. Wadą tego rozwiązania jest to, że kod źródłowy programu może być przeglądany (oraz skradziony) przez każdego. Jednak nie wydaje się to być zbyt wysoką ceną, ponieważ w językach skryptowych nie realizujemy wyszukanych zadań.

Jest taki język skryptowy, którego obsługi (i to bez dodatkowych modułów rozszerzających) można oczekiwać od większości nowoczesnych przeglądarek WWW: to JavaScript (który nie ma nic wspólnego z Javą — został tak nazwany, żeby się „załapać" na trochę marketingowego pędu związanego z Javą). Niestety, wersje języka zaimplementowane w różnych przeglądarkach mogą się znacznie od siebie różnić. Sytuację poprawiła nieco standaryzacja języka JavaScript w postaci *ECMAScript*, ale z kolei powszechne wdrożenie tego standardu wlecze się niemiłosiernie (ma w tym swój udział firma Microsoft, forsująca swój język skryptowy VBScript, wielce zresztą podobny do JavaScript). Zasadniczo należałoby przy programowaniu po stronie klienta ograniczyć się do najmniejszego wspólnego mianownika różnych implementacji JavaScript — tylko wtedy można mieć nadzieję bezproblemowego działania kodu w różnych przeglądarkach. Diagnostykę i wychwytywanie błędów w kodzie JavaScript można opisać jedynie jako udrękę. Dowodem stopnia złożoności problemu jest choćby to, że dopiero niedawno powstał pierwszy naprawdę rozbudowany projekt z użyciem JavaScript — mowa o Google GMail.

Widać z tego, że języki skryptowe, używane wewnątrz przeglądarek WWW, są przeznaczone do rozwiązywania specyficznego typu problemów, głównie do tworzenia bogatszych i bardziej atrakcyjnych graficznych interfejsów użytkownika (ang. *graphical user interface*, w skrócie GUI). Języki skryptowe mogą rozwiązać 80 procent problemów napotykanych przy programowaniu po stronie klienta. Również Twój problem może w całości mieścić się w tych 80 procentach, a ponieważ języki skryptowe pozwalają na łatwiejszą i szybszą produkcję stron WWW, powinieneś rozważyć ich użycie, zanim zwrócisz się ku bardziej złożonym rozwiązaniom, jak programowanie w języku Java.

Java

Jeśli języki skryptowe mogą rozwiązać 80 procent problemów programowania klient-serwer, co z pozostałymi 20 procentami — tymi naprawdę trudnymi? Obecnie najpopularniejszym rozwiązaniem jest Java. Jest to nie tylko potężny język programowania stworzony jako bezpieczny, przenośny i międzynarodowy. Java jest ciągle rozszerzana, by dostarczać nowe rozwiązania, które obsługują problemy uznawane za trudne w tradycyjnych językach, takie jak: wielowątkowość, dostęp do baz danych, programowanie sieciowe czy przetwarzanie rozproszone. Java umożliwia programowanie po stronie klienta poprzez *aplety* oraz technologię *Java Web Start*.

Aplet jest miniprogramem, który działa wyłącznie pod kontrolą przeglądarki. Jest on ładowany automatycznie jako element strony WWW (tak samo jak obrazek). Kiedy aplet zostanie aktywowany, wykona swój program. Jest to jego duży plus — zapewnia metodę automatycznego rozpowszechniania oprogramowania klienta z serwera w momencie, kiedy użytkownik go potrzebuje, a nie wcześniej. Użytkownik otrzymuje najnowszą wersję oprogramowania klienta bez borykania się z trudnościami ponownej instalacji. Java została zaprojektowana w taki sposób, że programista tworzy tylko jeden program, który automatycznie będzie działał na wszystkich komputerach wyposażonych w przeglądarki

z wbudowanym interpreterem Javy (można do nich zaliczyć większość maszyn). Ponieważ Java jest w pełni rozwiniętym językiem programowania, można więc obarczyć klienta dużą ilością pracy zarówno przed, jak i po wysłaniu żądania do serwera. Na przykład nie trzeba wysyłać żądania, aby odkryć, że źle podałeś datę lub inny parametr, a sam klient może szybko wykonać całą pracę związaną ze sporządzeniem wykresu, zamiast czekać, aż serwer przygotuje wykres i wyśle gotowy obrazek. Otrzymujesz natychmiastowy wzrost prędkości działania, a ogólny ruch sieci i obciążenie serwerów mogą zostać zredukowane, przyspieszając tym samym działanie całego Internetu.

Alternatywy

Gwoli szczerości, aplety Javy nie spełniły do końca pokładanych w nich pierwotnie oczekiwań. Kiedy Java ujrzała światło dzienne, wszyscy fascynowali się właśnie apletami jako wyczekiwanymi narzędziami programowania po stronie klienta, a więc zwiększaniem reaktywności i zmniejszeniem wymagań co do szybkości kanału transmisyjnego w aplikacjach internetowych. Apletom prorokowano wielką karierę.

W rzeczy samej w sieci WWW widuje się wielce zmyślne aplety, ale nigdy nie doszło do prawdziwej migracji do tej technologii. Największym problemem była najprawdopodobniej niechęć przeciętnych użytkowników do ściągania i instalowania środowiska wykonawczego Javy (JRE, od Java Runtime Environment) w postaci 10-megabajtowego pakietu. Los apletów mógł zostać przypieczętowany decyzją firmy Microsoft o niewłączaniu JRE do przeglądarki Internet Explorer. Tak czy inaczej aplety Javy nie zaistniały na większą skalę.

Mimo to aplety jako takie i technologia Java Web Start wciąż okazują się w pewnych sytuacjach przydatne. Znajdują zastosowanie wszędzie tam, gdzie zachodzi potrzeba kontrolowania maszyn użytkowników, na przykład w obrębie systemu informatycznego korporacji; służą tam do rozprowadzania i aktualizacji aplikacji klienckich przy znacznej oszczędności czasu oraz zasobów ludzkich i finansowych — widocznej zwłaszcza tam, gdzie aktualizacje są częste.

W rozdziale „Graficzne interfejsy użytkownika” przyjrzymy się nowej, obiecującej technologii *Flex* firmy Macromedia, która pozwala na tworzenie odpowiedników apletów, tyle że opartych na technologii Flash. Ponieważ jakieś 98 procent przeglądarek WWW dysponuje odtwarzaczami formatu Flash (dotyczy to przeglądarek dla wszystkich popularnych systemów operacyjnych), można go uznać za powszechnie przyjęty standard. Instalacja i aktualizacje odtwarzacza Flash odbywają się szybko i prosto. Stosowany tu język ActionScript bazuje na języku ECMAScript, przez co łatwo się do niego przyzwyczaić, ale Flex pozwala na programowanie bez martwienia się o charakterystykę przeglądarki, co czyni go znacznie atrakcyjniejszym od JavaScriptu. W dziedzinie programowania po stronie klienta jest to z pewnością alternatywa godna rozważenia.

.NET i C#

Od dłuższego czasu głównym konkurentem apletów Javy były komponenty ActiveX, choć wymagały one, aby klient działał w systemie operacyjnym Windows. Następnie Microsoft stworzył godnego rywala dla Javy — platformę *.NET* oraz język *C#*. Platforma .NET jest mniej więcej tym samym co *wirtualna maszyna Javy* oraz wszystkie biblioteki tego języka, a podobieństwa języków C# i Java są oczywiste. Bez wątpienia jest to najlepsza

robota, jaką Microsoft wykonał w dziedzinie języków programowania i środowisk programistycznych. Oczywiście Microsoft miał znaczącą przewagę, gdyż mógł przeanalizować, co w Javie było dobre a co złe, i bazować na tej wiedzy. Jednak została ona wykorzystana z doskonałym rezultatem. Po raz pierwszy od momentu pojawienia się Java zyskała godnego siebie konkurenta; dzięki temu z kolei projektanci języka Java przysiedli fałdów i, przyglądając się C# oraz przyczynom, dla których programiści mogą wybierać go, a nie Javę, odpowiedzieli rozszerzeniem Javy w postaci wydania Java SE5 z jego fundamentalnymi usprawnieniami.

Aktualnie głównym słabym punktem oraz pytaniem dotyczącym platformy .NET jest to, czy Microsoft zezwoli na przeniesienie go w *całości* na inne platformy systemowe. Microsoft twierdzi, że nie powinno to być żadnym problemem, a projekt Mono (*www.go-mono.com*) stanowi częściową implementację platformy .NET działającą w systemach Linux. Jednak do momentu zakończenia prac nad pełną implementacją i podjęcia przez Microsoft decyzji dotyczącej wszystkich elementów platformy .NET wykorzystanie jej jako rozwiązania międzysystemowego jest ryzykowne.

Internet kontra intranet

Sieć WWW jest najogólniejszym rozwiązaniem problemu klient-serwer, więc wydaje się sensowne użycie tej samej technologii do podzbioru tego problemu — w szczególności klasycznego problemu klient-serwer *wewnątrz* firmy. Przy tradycyjnym podejściu klient-serwer występują trudności z powodu wielu typów komputerów po stronie klientów oraz kłopoty z instalacją nowego oprogramowania. Obydwa problemy są dobrze rozwiązywane przez przeglądarkę WWW i programowanie po stronie klienta. Kiedy technologia WWW jest używana w sieci informacyjnej ograniczonej do konkretnej firmy, określa się ją mianem intranetu. Intranet dostarcza o wiele wyższy poziom bezpieczeństwa niż Internet, ponieważ można fizycznie kontrolować dostęp do firmowych serwerów. Jeśli wziąć zaś pod uwagę szkolenie pracowników, wydaje się, że kiedy już zrozumieją ogólne zasady pracy z przeglądarką, jest im dużo łatwiej radzić sobie z różnicami między działaniem stron i apletów, a czas potrzebny do nauczenia się nowych systemów jest krótszy.

Problem bezpieczeństwa doprowadził do jednego z podziałów powstałych automatycznie w świecie programowania typu klient-serwer. Jeśli program działa w Internecie, nie wiemy, na jakiej platformie będzie pracował, a jednocześnie zwracamy szczególną uwagę, aby nie rozpowszechniać kodu zawierającego błędy. Potrzebujemy czegoś tak przenośnego i bezpiecznego, jak język skryptowy lub Java.

Pracując w intranecie, napotykamy zestaw innych ograniczeń. Nie jest rzeczą niezwykłą, że wszystkie maszyny będą pracować na platformie Intel-Windows. W intranecie odpowiadamy za jakość własnego kodu i możemy naprawiać błędy zaraz po ich wykryciu. W dodatku często trzeba wykorzystać kod pozostały po wcześniejszych, tradycyjnych implementacjach systemu. Trzeba wtedy fizycznie instalować programy klientów przy każdorazowym uaktualnieniu. Czas tracony na instalowanie uaktualnień jest najczęstszym powodem przejścia na korzystanie z przeglądarki, ponieważ tutaj uaktualnienia są niewidoczne i automatyczne. Jeśli jesteś zaangażowany w tego typu projekt intranetowy, najrozsądniejszym rozwiązaniem jest obranie najprostszej drogi, umożliwiającej wykorzystanie istniejącej bazy kodu zamiast przepisywania programów ponownie w nowym języku.

W przypadku tak konsternującego bogactwa rozwiązań problemu klient-serwer najlepszym wyjściem jest analiza kosztów i korzyści. Rozważ ograniczenia postawionego problemu i odszukaj najkrótszą ścieżkę do rozwiązania. Ponieważ programowanie po stronie klienta nadal pozostaje programowaniem, zawsze dobrym pomysłem jest przyjęcie podejścia najszybciej prowadzącego do rozwiązania. Jest to agresywna postawa przygotowująca na spotkanie z problemami nieuniknionymi przy tworzeniu oprogramowania.

Programowanie po stronie serwera

W dotychczasowej dyskusji pomijany był temat programowania po stronie serwera. Co się dzieje w momencie wysłania żądania do serwera? Najczęstszym żądaniem jest proste: „Wyślij mi ten plik". Następnie przeglądarka interpretuje otrzymany plik w odpowiedni sposób: jako stronę HTML, obrazek, aplet Javy, skrypt itd.

Bardziej skomplikowane żądania kierowane do serwera wymagają na ogół komunikacji z bazą danych. W wielu przypadkach wymaga to wykonania złożonego zapytania na bazie danych, które serwer formatuje jako stronę HTML i odsyła z powrotem do klienta (oczywiście jeśli klient ma większe możliwości — dzięki Javie lub językom skryptowym — to surowe dane mogą być przesłane i sformatowane po stronie klienta, co jest szybsze i mniej obciąża serwer). Podobnie rejestracja użytkownika w chwili dołączenia do grupy dyskusyjnej lub złożenia zamówienia wymaga wprowadzenia zmian w bazie danych. Te wszystkie żądania muszą zostać przetworzone przez jakiś program działający po stronie serwera, co ogólnie określane jest jako programowanie po stronie serwera. Tradycyjnie programy CGI działające po stronie serwera były tworzone przy użyciu Perla, Pythona lub C++, ale pojawiały się też bardziej wyszukane systemy. Weźmy pod uwagę np. serwery sieciowe wykorzystujące Javę. Umożliwiają one wykonanie całości programowania po stronie serwera przez pisanie *serwletów*. Serwlety i ich pochodna JSP eliminują problemy związane z różnorodnym poziomem zaawansowania różnego typu przeglądarek. Z tego powodu wiele firm tworzących strony WWW przechodzi na Javę. (Te zagadnienia zostały opisane w książce *Thinking in Enterprise Java — zobacz www.MindView.net*).

Większość szumu wokół Javy związana była z apletami. W rzeczywistości Java jest językiem programowania ogólnego przeznaczenia, który może rozwiązywać dowolny rodzaj problemów. Siłą Javy jest nie tylko jej przenośność, ale także możliwości programistyczne, niezawodność, powszechność, dostępność bibliotek standardowych i licznych łatwo dostępnych i żywiołowo rozwijających się bibliotek dodatkowych.

Podsumowanie

Wszyscy wiemy, jak wygląda program proceduralny: definicje danych i wywołania funkcji. Aby odgadnąć znaczenie takiego programu, trzeba się troszkę napracować, prześledzić wywołania funkcji i zbadać niskopoziomowe pojęcia, aby stworzyć własny myślowy model. Z tego powodu potrzebujemy reprezentacji pośrednich przy projektowaniu programu proceduralnego — programy te same w sobie mogą być niezrozumiałe, ponieważ środki wyrazu są skierowane bardziej na komputery niż na rozwiązywany problem.

Ponieważ programowanie obiektowe dodaje wiele pojęć do tych, które można znaleźć w językach proceduralnych, może się wydawać naturalnym założeniem, że wynikowy program w języku Java może być o wiele bardziej skomplikowany, niż jego równoważnik w języku proceduralnym. Tutaj spotyka nas miła niespodzianka: dobrze napisany program jest na ogół o wiele łatwiejszy do zrozumienia niż odpowiadający mu kod proceduralny. To, co widzimy, to definicje obiektów reprezentujących pojęcia z przestrzeni problemu (a nie kwestie związane z reprezentacją komputerową) oraz komunikaty wysyłane do tych obiektów, reprezentujące działania w przestrzeni problemu. Jedną z przyjemnych stron programowania zorientowanego obiektowo jest to, że dobrze zaprojektowany program można zrozumieć, czytając jego kod. Najczęściej kod jest też o wiele krótszy, ponieważ wiele problemów zostało rozwiązanych przez zastosowanie kodu z istniejących bibliotek.

Programowanie obiektowe i Java nie muszą być dobre dla każdego. Ważne jest, by ocenić swoje potrzeby i zdecydować, czy Java w sposób optymalny je zaspokaja, czy też lepiej będzie użyć innego języka programowania (wliczając obecnie używany). Jeśli wiadomo, że w przewidywalnej przyszłości stawiane wymagania będą bardzo specjalne i jeśli dodatkowo pojawiają się specyficzne ograniczenia, których Java nie wyeliminuje, lepiej będzie zbadać rozwiązania alternatywne (w szczególności polecam zwrócenie uwagi na język Python; patrz *www.Python.org*). Jeśli ostatecznie wybór padnie na Javę jako preferowany język programowania, będzie to świadoma decyzja podjęta po rozważeniu innych opcji.