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

Visual Basic .NET. Wzorce projektowe



Autorzy: Mark Grand, Brad Merrill
Tłumaczenie: Mikołaj Szczepaniak
ISBN: 83-246-0270-4
Tytuł oryginału: Visual Basic .NET Design Patterns
Format: B5, stron: 576

Poznaj zasady stosowania wzorców projektowych

- Wykorzystaj notację UML
- Użyj wzorców projektowych w swojej pracy
- Napisz własne wzorce

Wzorce projektowe to opisy rozwiązań problemów programistycznych. Ich stosowanie podczas tworzenia oprogramowania pozwala uzyskać oszczędności czasowe, zwiększyć efektywność pracy i zoptymalizować działanie programów. Wiele wzorców już udokumentowano, a proces odkrywania nowych ciągle trwa. Programista dysponujący wiedzą o wzorcach projektowych może łatwo rozpoznawać problemy, dla których te wzorce znajdują zastosowanie, i natychmiast przystępować do opracowywania rozwiązań, bez konieczności wstrzymywania projektu, analizowania problemu i rozważania możliwych strategii.

Książka „Visual Basic .NET. Wzorce projektowe" to kompendium wiedzy o stosowaniu wzorców projektowych podczas programowania w języku Visual Basic. Zawiera szczegółowe omówienie najpopularniejszych wzorców, reguły ich wykorzystywania oraz przykłady kodu na nich opartego. Przy opisie każdego wzorca znajdziesz również argumenty przemawiające za jego stosowaniem lub unikaniem w konkretnych przypadkach. Nauczysz się tworzyć własne wzorce i dokumentować je w postaci diagramów UML.

- Podstawowe elementy języka UML
- Proces projektowania oprogramowania
- Miejsce wzorców projektowych w procesie tworzenia oprogramowania
- Wzorce podstawowe
- Wzorce konstrukcyjne
- Wzorce partycjonujące
- Wzorce strukturalne
- Wzorce behawioralne
- Wzorce przetwarzania współbieżnego

Naucz się rozwiązywać rzeczywiste problemy, wykorzystując wzorce projektowe

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

Spis treści

O autorach **13**

Wprowadzenie **15**
Krótka historia wzorców projektowych 16
Organizacja niniejszej książki 17
Opis wzorców projektowych 18
Nazwa wzorca 18
Streszczenie 18
Kontekst 19
Zalety 19
Rozwiązanie 19
Implementacja 19
Skutki stosowania 20
Zastosowania w technologii .NET 20
Przykład kodu źródłowego 20
Wzorce pokrewne 20
Kto powinien przeczytać tę książkę? 20

Rozdział 1. Przegląd języka UML **23**
Diagram klas 24
Diagram współpracy 36
Diagram stanów 45

Rozdział 2. Cykl życia oprogramowania **47**
Studium przypadku 50
Przypadek biznesowy 51
Definiowanie specyfikacji wymagań 52
Opracowanie kluczowych przypadków użycia wysokiego poziomu 54
Analiza obiektowa 56
Projekt obiektowy 58

Rozdział 3. Podstawowe wzorce projektowe **69**
Delegation (kiedy nie należy stosować relacji dziedziczenia) 70
Streszczenie 70
Kontekst 70
Zalety 73

Rozwiązanie ... 75
Implementacja ... 75
Skutki stosowania ... 75
Zastosowania w technologii .NET ... 76
Przykład kodu źródłowego ... 76
Wzorce pokrewne ... 78
Interfejs ... 78
Streszczenie ... 78
Kontekst ... 78
Zalety ... 79
Rozwiązanie ... 80
Implementacja ... 80
Skutki stosowania ... 82
Zastosowania w technologii .NET ... 82
Przykład kodu źródłowego ... 82
Wzorce pokrewne ... 83
Abstract Base Class ... 83
Streszczenie ... 83
Kontekst ... 84
Zalety ... 85
Rozwiązanie ... 86
Implementacja ... 87
Skutki stosowania ... 87
Zastosowania w interfejsie API platformy .NET ... 87
Przykład kodu źródłowego ... 88
Wzorce pokrewne ... 90
Interface and Abstract Class ... 90
Streszczenie ... 90
Kontekst ... 90
Zalety ... 90
Rozwiązanie ... 91
Skutki stosowania ... 92
Zastosowania w interfejsie API platformy .NET ... 92
Przykład kodu źródłowego ... 93
Wzorce pokrewne ... 94
Immutable ... 94
Streszczenie ... 94
Kontekst ... 94
Zalety ... 96
Rozwiązanie ... 97
Implementacja ... 98
Skutki stosowania ... 98
Zastosowania w interfejsie API platformy .NET ... 99
Przykład kodu źródłowego ... 99
Wzorce pokrewne ... 100
Proxy ... 100
Streszczenie ... 100
Kontekst ... 100
Zalety ... 102
Rozwiązanie ... 102
Implementacja ... 103
Skutki stosowania ... 103
Przykład kodu źródłowego ... 103
Wzorce pokrewne ... 110

Rozdział 4. Konstrukcyjne wzorce projektowe 113
Factory Method 114
Streszczenie 114
Kontekst 115
Zalety 116
Rozwiązanie 116
Implementacja 119
Skutki stosowania 121
Zastosowania w technologii .NET 122
Przykład kodu źródłowego 122
Wzorce pokrewne 127
Abstract Factory 128
Streszczenie 128
Kontekst 128
Zalety 130
Rozwiązanie 131
Implementacja 133
Skutki stosowania 133
Przykład kodu źródłowego 135
Wzorce pokrewne 139
Builder 139
Streszczenie 139
Kontekst 139
Zalety 143
Rozwiązanie 143
Implementacja 145
Skutki stosowania 147
Zastosowania w technologii .NET 147
Przykład kodu źródłowego 147
Wzorce pokrewne 149
Prototype 149
Streszczenie 149
Kontekst 150
Zalety 151
Rozwiązanie 151
Implementacja 153
Skutki stosowania 154
Zastosowania w technologii .NET 155
Przykład kodu źródłowego 155
Wzorce pokrewne 159
Singleton 159
Streszczenie 159
Kontekst 160
Zalety 161
Rozwiązanie 161
Implementacja 162
Skutki stosowania 164
Zastosowania w technologii .NET 165
Przykład kodu źródłowego 165
Wzorce pokrewne 166
Object Pool 166
Streszczenie 166
Kontekst 166
Zalety 169

Rozwiązanie 169
Implementacja 171
Skutki stosowania 174
Przykład kodu źródłowego 174
Wzorce pokrewne 178

Rozdział 5. Partycjonujące wzorce projektowe 181
Filter 181
Streszczenie 181
Kontekst 182
Zalety 182
Rozwiązanie 183
Implementacja 186
Skutki stosowania 187
Zastosowania w technologii .NET 187
Przykład kodu źródłowego 188
Wzorce pokrewne 191
Composite 191
Streszczenie 191
Kontekst 192
Zalety 193
Rozwiązanie 194
Implementacja 195
Skutki stosowania 196
Zastosowania w technologii .NET 198
Przykład kodu źródłowego 198
Wzorce pokrewne 202
Read-Only Interface 203
Streszczenie 203
Kontekst 203
Zalety 205
Rozwiązanie 206
Implementacja 207
Skutki stosowania 207
Przykład kodu źródłowego 207
Wzorce pokrewne 209

Rozdział 6. Strukturalne wzorce projektowe 211
Adapter 212
Streszczenie 212
Kontekst 212
Zalety 214
Rozwiązanie 214
Implementacja 215
Skutki stosowania 216
Przykład kodu źródłowego 217
Wzorce pokrewne 221
Iterator 221
Streszczenie 221
Kontekst 222
Zalety 223
Rozwiązanie 223
Implementacja 224
Skutki stosowania 226

Zastosowania w technologii .NET 226
Przykład kodu źródłowego 227
Wzorce pokrewne 228
Bridge 229
Streszczenie 229
Kontekst 229
Zalety 232
Rozwiązanie 232
Implementacja 234
Skutki stosowania 234
Zastosowania w technologii .NET 235
Przykład 235
Wzorce pokrewne 239
Façade 240
Streszczenie 240
Kontekst 240
Zalety 242
Rozwiązanie 242
Implementacja 243
Skutki stosowania 244
Zastosowania w technologii .NET 244
Przykład kodu źródłowego 244
Wzorce pokrewne 247
Flyweight 247
Streszczenie 247
Kontekst 247
Zalety 252
Rozwiązanie 252
Implementacja 253
Skutki stosowania 254
Zastosowania w technologii .NET 254
Przykład kodu źródłowego 254
Wzorce pokrewne 259
Dynamic Linkage 259
Streszczenie 259
Kontekst 259
Zalety 261
Rozwiązanie 262
Implementacja 263
Skutki stosowania 265
Zastosowania w technologii .NET 265
Przykład kodu źródłowego 265
Wzorce pokrewne 268
Virtual Proxy 269
Streszczenie 269
Kontekst 269
Zalety 271
Rozwiązanie 271
Implementacja 274
Skutki stosowania 275
Przykład kodu źródłowego 275
Wzorce pokrewne 277
Decorator 278
Streszczenie 278

Kontekst 278
Zalety 280
Rozwiązanie 281
Implementacja 282
Skutki stosowania 282
Przykład kodu źródłowego 283
Wzorce pokrewne 285
Cache Management 285
Streszczenie 285
Kontekst 285
Zalety 287
Rozwiązanie 288
Implementacja 289
Skutki stosowania 296
Przykład kodu źródłowego 297
Wzorce pokrewne 306

Rozdział 7. Behawioralne wzorce projektowe 307
Chain of Responsibility 308
Streszczenie 308
Kontekst 308
Zalety 310
Rozwiązanie 310
Implementacja 312
Skutki stosowania 313
Zastosowania w technologii .NET 313
Przykład kodu źródłowego 313
Wzorce pokrewne 318
Command 318
Streszczenie 318
Kontekst 318
Zalety 319
Rozwiązanie 320
Implementacja 321
Skutki stosowania 323
Zastosowania w technologii .NET 324
Przykład kodu źródłowego 324
Wzorce pokrewne 329
Little Language 330
Streszczenie 330
Kontekst 330
Zalety 342
Rozwiązanie 342
Implementacja 344
Skutki stosowania 345
Zastosowania w technologii .NET 346
Przykład kodu źródłowego 346
Wzorce pokrewne 355
Mediator 356
Streszczenie 356
Kontekst 356
Zalety 358
Rozwiązanie 359
Implementacja 360

Skutki stosowania 362
Przykład kodu źródłowego 363
Wzorce pokrewne 368
Snapshot 368
Streszczenie 368
Kontekst 369
Zalety 373
Rozwiązanie 374
Implementacja 376
Skutki stosowania 384
Przykład kodu źródłowego 385
Wzorce pokrewne 387
Observer 387
Streszczenie 387
Kontekst 387
Zalety 389
Rozwiązanie 389
Implementacja 391
Skutki stosowania 394
Zastosowania w technologii .NET 395
Przykład kodu źródłowego 395
Wzorce pokrewne 397
State 397
Streszczenie 398
Kontekst 398
Zalety 402
Rozwiązanie 402
Implementacja 404
Skutki stosowania 405
Przykład kodu źródłowego 405
Wzorce pokrewne 410
Strategy 410
Streszczenie 410
Kontekst 411
Zalety 412
Rozwiązanie 412
Implementacja 413
Skutki stosowania 413
Zastosowania w technologii .NET 414
Przykład kodu źródłowego 414
Wzorce pokrewne 416
Null Object 416
Streszczenie 417
Kontekst 417
Zalety 418
Rozwiązanie 418
Implementacja 419
Skutki stosowania 419
Przykład kodu źródłowego 420
Wzorce pokrewne 421
Template Method 422
Streszczenie 422
Kontekst 422
Zalety 424

Rozwiązanie 424
Implementacja 425
Skutki stosowania 426
Przykład kodu źródłowego 426
Wzorce pokrewne 428
Visitor 428
Streszczenie 429
Kontekst 429
Zalety 432
Rozwiązanie 433
Implementacja 436
Skutki stosowania 437
Przykład kodu źródłowego 437
Wzorce pokrewne 440
Hashed Adapter Objects 441
Streszczenie 441
Kontekst 441
Zalety 444
Rozwiązanie 445
Implementacja 446
Skutki stosowania 448
Przykład kodu źródłowego 448
Wzorce pokrewne 450

Rozdział 8. Wzorce projektowe przetwarzania współbieżnego 451

Single Threaded Execution 452
Streszczenie 453
Kontekst 453
Zalety 456
Rozwiązanie 456
Implementacja 457
Skutki stosowania 458
Przykład kodu źródłowego 459
Wzorce pokrewne 461
Static Locking Order 461
Streszczenie 461
Kontekst 461
Zalety 462
Rozwiązanie 463
Skutki stosowania 463
Implementacja 464
Znane zastosowania 464
Przykład kodu źródłowego 465
Wzorce pokrewne 466
Lock Object 467
Streszczenie 467
Kontekst 467
Zalety 468
Rozwiązanie 468
Implementacja 469
Skutki stosowania 471
Przykład kodu źródłowego 471
Wzorce pokrewne 473

Guarded Suspension 473
Streszczenie 473
Kontekst 473
Zalety 474
Rozwiązanie 475
Implementacja 476
Skutki stosowania 478
Zastosowania w technologii .NET 479
Przykład kodu źródłowego 479
Wzorce pokrewne 480
Balking 480
Streszczenie 480
Kontekst 480
Zalety 482
Rozwiązanie 482
Implementacja 483
Skutki stosowania 483
Przykład kodu źródłowego 483
Wzorce pokrewne 484
Scheduler 484
Streszczenie 485
Kontekst 485
Zalety 488
Rozwiązanie 488
Implementacja 490
Skutki stosowania 491
Przykład kodu źródłowego 491
Wzorce pokrewne 495
Read/Write Lock 495
Streszczenie 496
Kontekst 496
Zalety 497
Rozwiązanie 498
Implementacja 499
Skutki stosowania 500
Zastosowania w technologii .NET 501
Przykład kodu źródłowego 501
Wzorce pokrewne 506
Producer-Consumer 506
Streszczenie 506
Kontekst 506
Zalety 507
Rozwiązanie 507
Implementacja 508
Skutki stosowania 509
Zastosowania w technologii .NET 509
Przykład kodu źródłowego 509
Wzorce pokrewne 511
Double Buffering 511
Streszczenie 511
Kontekst 512
Zalety 513
Rozwiązanie 514
Implementacja 515

Skutki stosowania 517
Zastosowania w technologii .NET 518
Przykład kodu źródłowego 518
Wzorce pokrewne 530
Asynchronous Processing 531
Streszczenie 531
Kontekst 531
Zalety 534
Rozwiązanie 534
Implementacja 535
Skutki stosowania 537
Zastosowania w technologii .NET 537
Przykład kodu źródłowego 538
Wzorce pokrewne 540
Future 540
Streszczenie 540
Kontekst 541
Zalety 542
Rozwiązanie 543
Implementacja 545
Skutki stosowania 547
Zastosowania w technologii .NET 547
Przykład kodu źródłowego 549
Wzorce pokrewne 553

Bibliografia 555

Skorowidz 557

Rozdział 4.

Konstrukcyjne wzorce projektowe

Konstrukcyjne wzorce projektowe opisują techniki organizowania procesów tworzenia obiektów w sytuacjach, gdy realizacja tych procesów wymaga podejmowania istotnych decyzji. Tego rodzaju działania zwykle wymagają dynamicznego identyfikowania klas, których egzemplarze mają być tworzone, lub obiektów, do których należy delegować odpowiedzialność. Konstrukcyjne wzorce projektowe opisują struktury mechanizmów podejmowania tych decyzji.

Być może nie jest dla Ciebie jasne, dlaczego istnieją wyspecjalizowane wzorce projektowe dotyczące wyłącznie procesów tworzenia obiektów. Tworzenie obiektów klas wiąże się z istotnymi problemami, które nie występują w żadnych innych dziedzinach. Do najbardziej oczywistych problemów należą:

- Tworzenie obiektów zwykle polega na wywołaniu operacji `new`. Nazwa klasy, której egzemplarz tworzymy, występuje bezpośrednio po słowie `new`. Ponieważ musi to być konkretna, trwale zakodowana nazwa, a nie zmienna łańcuchowa, ten sposób tworzenia obiektów wyklucza możliwość stosowania jakiegokolwiek pośrednictwa. Jeśli decyzja odnośnie klasy tworzonego obiektu powinna być podejmowana w czasie wykonywania programu, należy zastosować inne rozwiązanie.
- Konstruowany obiekt może uzyskiwać dostęp wyłącznie do danych dzielonych (zadeklarowanych ze słowem `Shared`) oraz parametrów przekazanych na wejściu konstruktora. Dostęp do pozostałych informacji musi się odbywać za pośrednictwem odpowiednich mechanizmów.

W wielu sytuacjach można stosować więcej niż jeden konstrukcyjny wzorzec projektowy. O ile w niektórych przypadkach korzystnym wyjściem jest łączenie wielu wzorców, w pozostałych projektant musi wybrać pomiędzy wzorcami konkurencyjnymi. W związku z tym należy w dostatecznym stopniu opanować techniki korzystania ze wzorców projektowych, które przedstawiono w niniejszym rozdziale.

Jeśli dysponujesz czasem potrzebnym do opanowania tylko jednego spośród wzorców projektowych prezentowanych w niniejszym rozdziale, powinieneś zwrócić uwagę na wzorzec Factory Method (metody wytwórczej, metody fabrykującej), który jest najbardziej popularny. Wzorzec Factory Method umożliwia inicjowanie przez jeden obiekt procesu tworzenia innego obiektu w sytuacji, gdy nie jest znana klasa tworzonego obiektu.

Wzorzec projektowy Abstract Factory (wytwórni abstrakcji) umożliwia inicjowanie przez obiekty procesów tworzenia obiektów różnych typów bez znajomości klas tworzonych obiektów, ale z zapewnieniem zgodności kombinacji tych klas z określonymi kryteriami.

Wzorzec projektowy Builder (budowniczego) ma na celu identyfikowanie klasy tworzonego obiektu na podstawie jego zawartości lub bieżącego kontekstu.

Wzorzec projektowy Prototype (prototypu) umożliwia klasie inicjowanie procesu tworzenia obiektów w sytuacji, gdy nie jest znana klasa tych obiektów lub szczegóły związane z samym procesem ich tworzenia. W takim przypadku tworzenie żądanych obiektów polega na kopiowaniu obiektu już istniejącego.

Wzorzec projektowy Singleton (singletonu) oferuje możliwość współdzielenia przez wiele obiektów jednego, wspólnego obiektu niezależnie od tego, czy wiadomo o jego faktycznym istnieniu.

Wzorzec projektowy Object Pool (puli obiektów) umożliwia wielokrotnego wykorzystywania istniejących obiektów zamiast każdorazowo tworzyć nowe obiekty.

Factory Method

Wzorzec opisano po raz pierwszy w książce *Design Patterns: Elements of Reusable Object-Oriented Software* [Gamma, 1995].

Streszczenie

Musimy utworzyć obiekt reprezentujący zewnętrzne dane lub przetwarzający zewnętrzne zdarzenie. Typ tworzonego obiektu zwykle zależy od zawartości tych zewnętrznych danych lub rodzaju zdarzenia. Nie chcemy, aby źródło danych, źródło zdarzeń czy obiekty klienckie były ściśle związane z właściwym typem tworzonych obiektów ani jakimikolwiek specjalnymi procedurami inicjalizacji (wymaganymi przez tę czy inną klasę). Stosując wzorzec projektowy Factory Method, możemy ukryć decyzje o wyborze klasy tworzonych obiektów w innej klasie.

Kontekst

Przeanalizujmy problem opracowywania szkieletu (ang. *framework*) dla aplikacji zorganizowanych wokół dokumentów lub plików. Początkiem procesu tworzenia takiego szkieletu zwykle jest polecenie budowy nowego lub zmiany istniejącego dokumentu edytora tekstu, arkusza kalkulacyjnego, wykresu w czasie lub innego rodzaju dokumentu obsługiwanego przez daną aplikację.

Szkielet opracowany z myślą o tym typie aplikacji powinien obejmować wysokopoziomową obsługę typowych operacji, w tym operacji tworzenia, otwierania oraz zapisywania dokumentów. W tym przypadku obsługa wspomnianych operacji powinna polegać na zapewnieniu spójnego zbioru metod wywoływanych w odpowiedzi na polecenia wydawane przez użytkownika. Na potrzeby dalszych rozważań nadamy naszej klasie udostępniającej odpowiednie operacje nazwę `DocumentManager`.

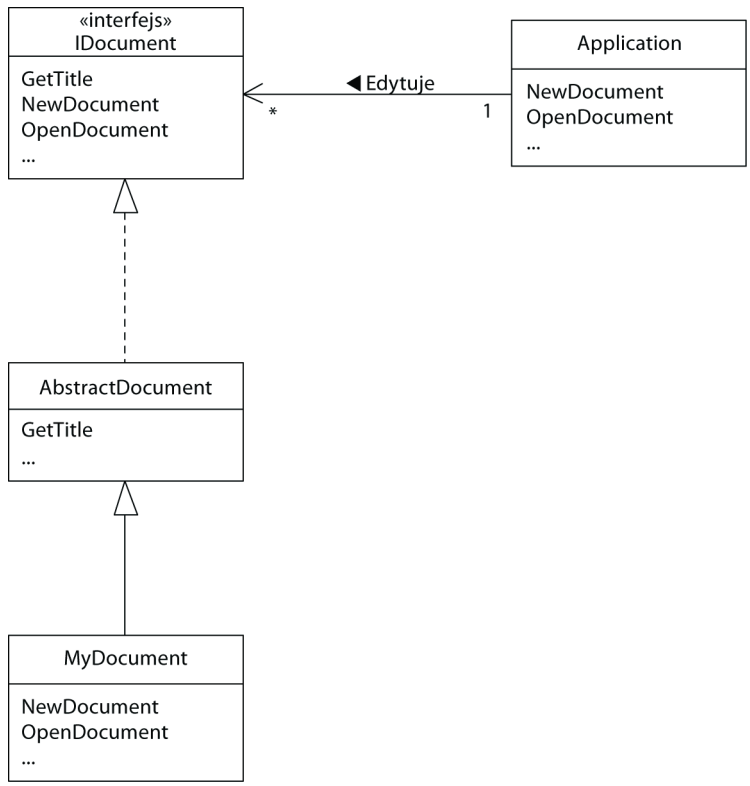
Ponieważ logika implementacji większości operacji jest ściśle uzależniona od rodzaju dokumentu, klasa `DocumentManager` musi delegować zadania obsługi większości poleceń użytkownika do odpowiednich obiektów dokumentów. Logika tych obiektów w zakresie implementowania wspomnianych poleceń odpowiada reprezentowanym typom dokumentów. Warto jednak pamiętać, że część operacji, np. wyświetlanie tytułu dokumentu, jest wspólna dla wszystkich obiektów dokumentów. Na tej podstawie można zaprojektować następującą strukturę naszego szkieletu:

- Interfejs dokumentu niezależny od aplikacji.
- Klasa abstrakcyjna udostępniająca niezależną od aplikacji logikę dla konkretnych klas dokumentów.
- Konkretne, właściwe dla danej aplikacji klasy, które implementują interfejs dla poszczególnych rodzajów dokumentów.

Schemat tego szkieletu aplikacji przedstawiono na rysunku 4.1.

Ani dotychczasowa analiza, ani rysunek 4.1 nie rozstrzygają, jak obiekty klasy `DocumentManager` mogą tworzyć egzemplarze klas dokumentów właściwych dla poszczególnych aplikacji, skoro sama klasa `DocumentManager` ma być niezależna od aplikacji.

Jednym ze sposobów realizacji tego celu jest opracowanie przez programistów korzystających z tego szkieletu klasy, która będzie zawierała niezbędną logikę w zakresie wyboru klas właściwych dla budowanych aplikacji i tworzenia ich egzemplarzy. Aby klasa `DocumentManager` mogła wywoływać klasy dostarczone przez programistów aplikacji bez wprowadzania dodatkowych zależności, szkielet powinien udostępniać interfejs, który będzie implementowany przez te klasy. Taki interfejs powinien deklarować funkcję implementowaną przez klasy dostarczane przez programistów aplikacji i obsługującą operacje wyboru oraz tworzenia egzemplarzy właściwej klasy. Klasa `DocumentManager` miałaby wówczas kontakt z interfejsem szkieletu, zatem nie byłaby bezpośrednio uzależniona od szczegółów implementacyjnych klas opracowanych przez programistów aplikacji. Schemat tego rozwiązania przedstawiono na rysunku 4.2.



Rysunek 4.1. *Szkielet aplikacji*

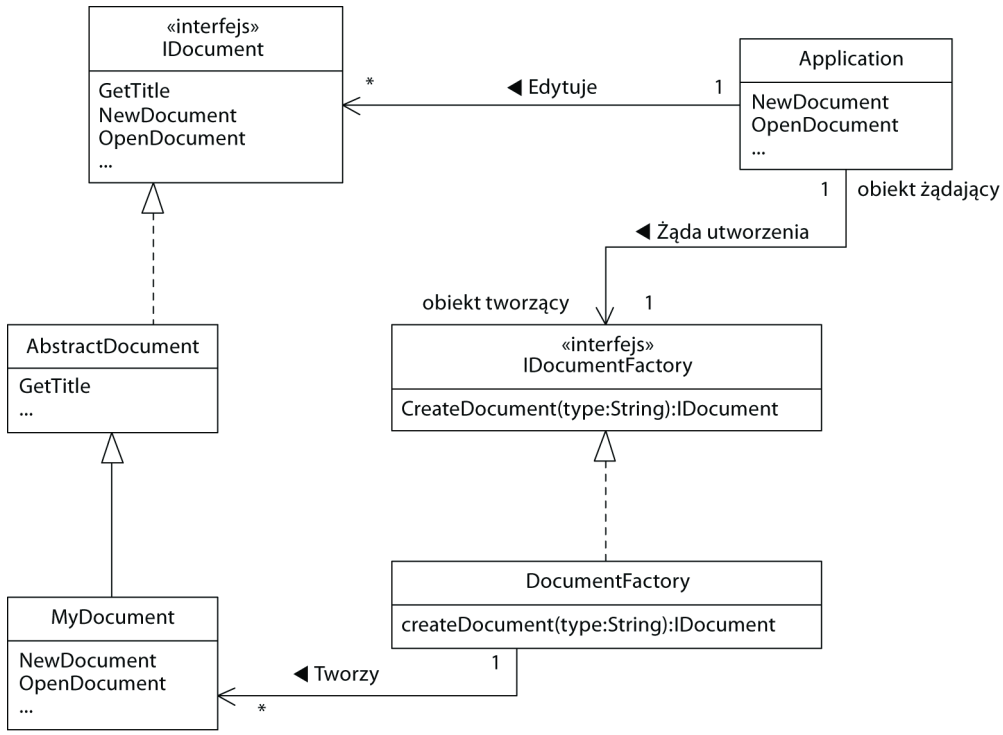
Z diagramu przedstawionego na rysunku 4.2 wynika, że obiekt klasy `DocumentManager` wywołuje funkcję `createDocument` obiektu klasy implementującej interfejs `IDocument-Factory`. Na wejściu tej funkcji obiekt klasy `DocumentManager` przekazuje łańcuch, na podstawie którego funkcja określa klasę implementującą interfejs `IDocumentFactory`, której egzemplarz należy utworzyć. Klasa `DocumentManager` nie musi więc dysponować informacjami ani na temat rzeczywistych klas obiektów, których funkcje wywołuje, ani o klasach tworzonych obiekty (w tym przypadku podklasach klasy `Document`).

Zalety

- ☺ Klasa oferuje możliwość inicjowania procesu tworzenia obiektów bez jakichkolwiek zależności łączących tę klasę z tworzonymi obiektami.
- ☺ Zbiór klas, których obiekty są tworzone przez daną klasę, może mieć charakter dynamiczny i ulegać zmianom wraz z pojawianiem się nowych klas.

Rozwiązanie

Należy stworzyć obiekty niezależne od aplikacji, które będą delegowały do obiektów właściwych dla danej aplikacji zadania związane z tworzeniem innych obiektów właściwych dla tej samej aplikacji. Obiekty niezależne od aplikacji, które inicjują proces



Rysunek 4.2. *Szkielet aplikacji wzbogacony o fabrykę dokumentów*

tworzenia obiektów właściwych dla aplikacji, zakładają, że klasy wszystkich tworzonych obiektów implementują wspólny interfejs.

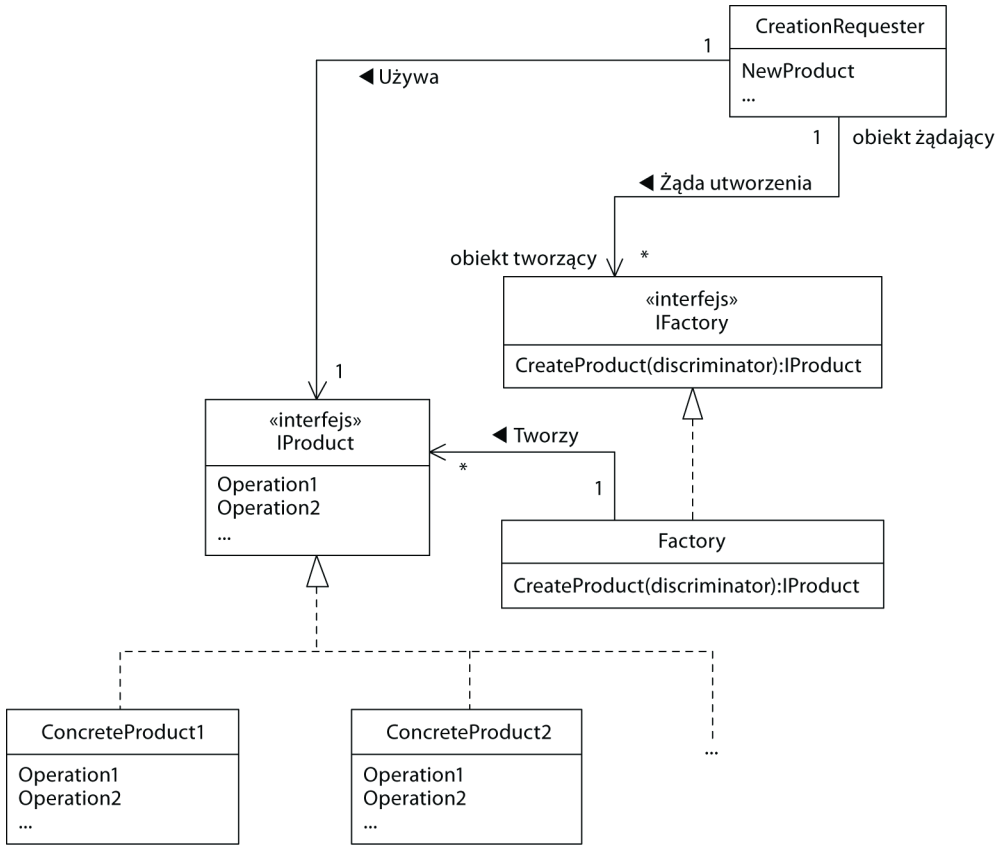
Na rysunku 4.3 przedstawiono interfejsy i klasy, które zwykle składają się na implementacje wzorca projektowego Factory Method.

Diagram klas zaprezentowany na rysunku 4.3 definiuje następujące role odgrywane przez klasy i interfejsy wchodzące w skład wzorca projektowego Factory Method:

`IProduct`. Klasy obiektów tworzonych zgodnie z zaleceniami wzorca projektowego Factory Method musza implementować interfejs występujący w tej roli.

`ConcreteProduct1`, `ConcreteProduct2`, ... Egzemplarze klas występujących w tej roli są tworzone przez obiekt klasy `Factory`. Klasy `ConcreteProduct1`, `ConcreteProduct2` itd. muszą implementować interfejs `IProduct`.

`CreationRequester`. Klasa występująca w tej roli jest klasą niezależną od aplikacji, której zadaniem jest tworzenie egzemplarzy klas zależnych od aplikacji (właściwych dla konkretnych implementacji tego szkieletu). Operacje tworzenia tych egzemplarzy są realizowane za pośrednictwem obiektu klasy implementującej interfejs `IFactory`.



Rysunek 4.3. *Przykład praktycznego zastosowania wzorca projektowego Factory Method*

`IFactory`. W roli `IFactory` występują interfejsy niezależne od aplikacji. Obiekty tworzące egzemplarze interfejsów `IFactory` (w imieniu obiektów klasy `CreationRequester`) muszą te interfejsy implementować. Wszelkie interfejsy występujące w tej roli deklarują funkcję, która jest każdorazowo wywoływana przez obiekty klasy `CreationRequester` celem utworzenia konkretnego obiektu produktu (klasy `ConcreteProduct1`, `ConcreteProduct2` itd.). Argumenty odpowiedniej funkcji omówiono w punkcie „Implementacja". Nazwy interfejsów występujących w tej roli najczęściej zawierają słowo `Factory`, np. `IDocumentFactory` lub `IImageFactory`.

`Factory`. W roli `Factory` występuje klasa właściwa dla danej aplikacji, która implementuje odpowiedni interfejs `IFactory` i definiuje funkcję odpowiedzialną za tworzenie obiektów klas `ConcreteProduct1`, `ConcreteProduct2` itd. Nazwy klas występujących w tej roli zwykle zawierają słowo `Factory`, np. `DocumentFactory` lub `ImageFactory`.

Implementacja

W wielu implementacjach wzorca projektowego Factory Method klasy `ConcreteProduct` nie implementują interfejsu `IProduct` bezpośrednio, tylko rozszerzają odpowiednią klasę abstrakcyjną implementującą ten interfejs. Omówienie tego rodzaju rozwiązań oraz analizę korzyści wynikających z dodatkowego stosowania abstrakcyjnej klasy bazowej znajdziesz w podrozdziale rozdziału 3. poświęconym wzorcowi projektowemu Interface and Abstract Class.

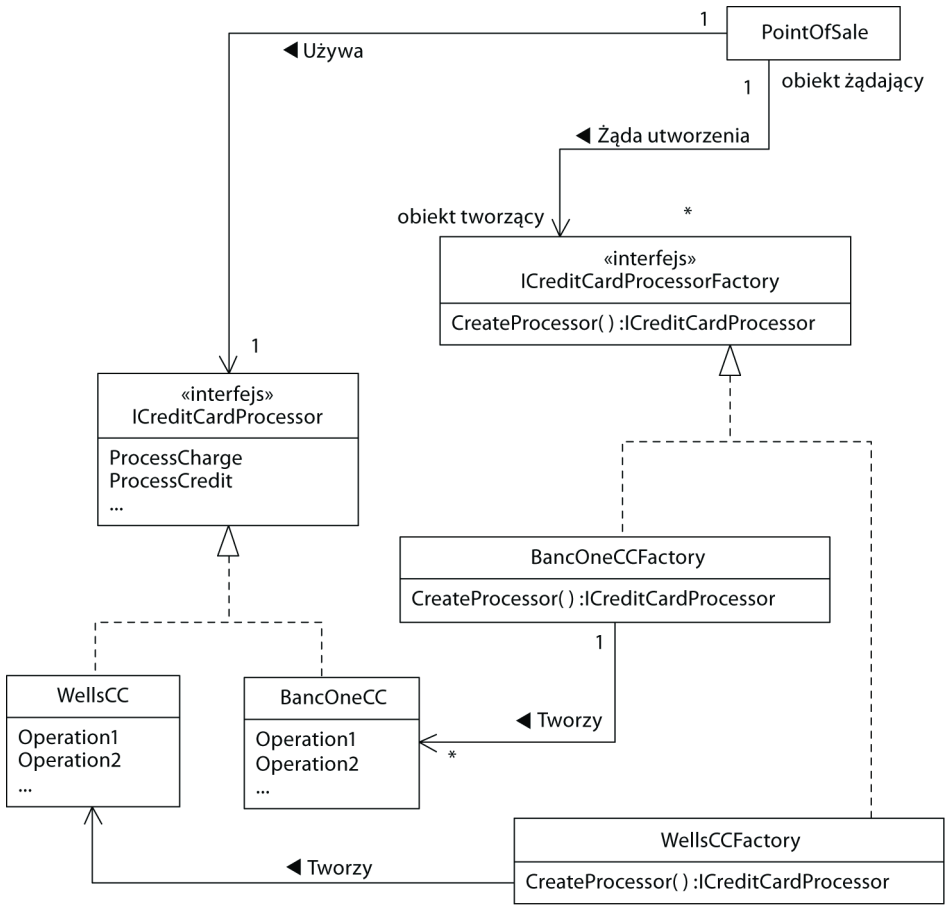
Określanie klasy według konfiguracji

Istnieją dwie główne odmiany wzorca projektowego Factory Method. Najczęściej klasa tworzonych obiektów jest określana w chwili inicjowania procesu tworzenia. Nie można jednak wykluczyć, że klasa tworzonych obiektów nie będzie się zmieniała w czasie wykonywania programu, ale operację jej określenia trzeba będzie wykonać jeszcze przed zainicjowaniem procesu tworzenia.

Program może korzystać z obiektu klasy `Factory`, który stale tworzy egzemplarze jednej klasy określonej na podstawie informacji zawartych w konfiguracji programu. Przypuśćmy na przykład, że jakaś firma oferuje swoim klientom oprogramowanie obsługujące punkt sprzedaży, którego ważną częścią jest system odpowiedzialny za komunikację ze zdalnym komputerem celem właściwego przetwarzania operacji bezgotówkowych. Klasy tego systemu powinny przesyłać do zdalnego komputera komunikaty i otrzymywać odpowiedzi generowane przez ten zdalny komputer za pomocą obiektów klas implementujących określony interfejs. Dokładny format komunikatów przesyłanych pomiędzy punktem sprzedaży a komputerem zdalnym zależy od firmy, która obsługuje transakcje dokonywane kartami płatniczymi. Dla każdej z tych firm istnieje właściwa klasa implementująca wymagany interfejs i zawierająca logikę niezbędną do przesyłania komunikatów w formacie zgodnym z oczekiwaniami tej firmy. Istnieje też odpowiednia klasa fabryki. Schemat opisanego rozwiązania przedstawiono na rysunku 4.4.

Poniżej zawarto wyjaśnienie projektu przedstawionego na rysunku 4.4. W momencie uruchamiania systemu punktu sprzedaży następuje odczytanie niezbędnych informacji zawartych w ustawieniach konfiguracyjnych. Ustawienia konfiguracyjne nakazują systemowi informatycznemu punktu sprzedaży przekazywanie danych o transakcjach dokonywanych z użyciem kart płatniczych do przetworzenia albo przez firmę BancOne, albo przez firmę Wells Fargo. Na bazie tych informacji system tworzy albo obiekt klasy `BancOneCCFactory`, albo obiekt klasy `WellsCCFactory`. System uzyskuje dostęp do tego obiektu za pośrednictwem interfejsu `ICreditCardProcessorFactory`. Kiedy zaistnieje konieczność przetworzenia transakcji z użyciem karty płatniczej, system wywoła funkcję `CreateProcessor` swojego obiektu klasy `ICreditCardProcessorFactory`, który z kolei utworzy obiekt odpowiedniej klasy zgodnie z dostępnymi ustawieniami konfiguracyjnymi.

Warto pamiętać, że funkcje `Create` klas fabryki nie muszą otrzymywać na wejściu żadnych argumentów, ponieważ zawsze zwracają obiekty tych samych typów.



Rysunek 4.4. *Przetwarzanie danych karty kredytowej*

Określanie klasy według danych

Klasa obiektów tworzonych przez obiekt fabryki bardzo często zależy od rodzaju danych, które mają być reprezentowane przez te obiekty. Za określanie właściwej klasy odpowiada funkcja `CreateProduct` obiektu fabryki. Proces wyboru klasy zwykle bazuje na informacjach przekazywanych do tej funkcji za pośrednictwem jej parametru. Przykładową implementację tego rodzaju funkcji przedstawiono poniżej:

```
Public Function createDrawing(ByVal theFile As FileInfo) As Drawing
    Select Case theFile.Extension
        Case "gfx1"
            Return New Gfx1Drawing(theFile)
        Case "mxfi"
            Return New MfxiDrawing(theFile)
            ...
    End Select
End Function
```

Wyrażenie `select` zdaje egzamin w tych funkcjach `CreateProduct`, które tworzą egzemplarze klas składających się na zbiór stałych rozmiarów. Podczas pisania funkcji `CreateProduct` obsługującej zmienną lub bardzo dużą liczbę klas produktów można użyć wzorca projektowego Hashed Adapter Object (zaproponowanego i omówionego w książce [Grand, 1998]). Alternatywnym rozwiązaniem jest użycie obiektów określających klasy, których obiekty należy utworzyć, w formie kluczy tablicy mieszającej oraz obiektów klasy `System.Reflection.Emit.ConstructorBuilder` występujących w roli wartości tej tablicy. Stosowanie tej techniki umożliwia odnajdywanie w tablicy mieszającej wartości przekazywanych za pośrednictwem argumentu i używanie właściwych obiektów klasy `ConstructorBuilder` do tworzenia egzemplarzy odpowiednich klas.

Inny fragment powyższego przykładu dobrze pokazuje, że metody wytwórcze (fabrykujące) są właściwym miejscem stosowania wspominanego już wyrażenia `select` lub łańcuchów wyrażeń `if`. W wielu przypadkach obecność w ciele funkcji wyrażeń `select` lub łańcuchów wyrażeń `if` wskazuje na konieczność zaimplementowania tego kodu w formie metody polimorficznej. Metody wytwórcze nie mogą być implementowanie z wykorzystaniem polimorfizmu, ponieważ idea polimorfizmu dotyczy wyłącznie stanu zaistniałego po utworzeniu obiektu.

W wielu implementacjach wzorca projektowego Factory Method za prawidłowe wartości argumentów funkcji `CreateProduct` obiektu fabryki uważa się elementy ustalonego z góry zbioru. Często wygodnym rozwiązaniem jest zdefiniowanie na potrzeby klasy fabryki symbolicznych nazw dla wszystkich obsługiwanych wartości należących do tego zbioru. Klasy, które żądają od klasy fabryki tworzenia obiektów, mogą wykorzystywać takie stałe definiujące nazwy symboliczne dla poszczególnych typów (klas) tworzonych obiektów. Jeśli wszystkie wartości są liczbami całkowitymi, najlepszym rozwiązaniem niż definiowanie poszczególnych stałych jest zdefiniowanie nazw symbolicznych za pomocą pojedynczego wyrażenia `Enum`

W niektórych sytuacjach proces określania klasy według danych ma charakter wielowarstwowy. Przykładowo, możemy mieć do czynienia z obiektem fabryki wysokiego poziomu odpowiedzialnym za tworzenie innych obiektów fabryki, które z kolei odpowiadają za tworzenie właściwych obiektów produktów. Z tego względu wersja wzorca Factory Method bazująca na technice określania klas według danych bywa nazywana *inicjalizacją wielowarstwową* (ang. *layered initialization*).

Skutki stosowania

Do najważniejszych skutków stosowania wzorca projektowego Factory Method należą:

- ☺ Klasa żądająca tworzenia obiektów jest całkowicie niezależna od klasy obiektów, które stanowią konkretne produkty i które powstają wskutek działań tej klasy żądającej.
- ☺ Zbiór klas produktów, których obiekty są tworzone przez klasę fabryki, może być zmieniany w sposób dynamiczny.
- ☹ Brak bezpośrednich relacji łączących operacje inicjowania tworzenia obiektu i określania klasy, której obiekt ma zostać utworzony, może utrudniać konserwację oprogramowania i zmniejszać czytelność kodu źródłowego.

Zastosowania w technologii .NET

Architektura zabezpieczeń platformy .NET Framework obejmuje mechanizm uprawnień. Analizując najwyższy poziom tej architektury, można przyjąć, że jego funkcjonowanie polega na wiązaniu egzemplarzy interfejsu `System.Security.IPermission` z kontekstami metod. Wywoływana metoda może sprawdzić, czy obiekt wywołujący dysponuje uprawnieniami w zakresie żądanych operacji — weryfikacja dotyczy zawierania odpowiednich uprawnień w kontekście obiektu wywołującego.

Architektura zabezpieczeń platformy .NET Framework wykorzystuje obiekty klas implementujących wspomniany interfejs `System.Security.IPermission` do określania, czy obiekty wywołujące metody faktycznie mają prawo żądać działań obsługiwanych przez te metody. Niektóre egzemplarze interfejsu `IPermission` bezpośrednio reprezentują prawa obiektu wywołującego do generowania żądań określonych usług. Przykładowo, obiekty klasy `System.Drawing.Printing.PrintingPermission` reprezentują prawa drukowania. Inne implementacje interfejsu `IPermission` nie odnoszą się równie bezpośrednio do precyzyjnie definiowanych usług.

Niektóre egzemplarze interfejsu `IPermission` umożliwiają wyłącznie identyfikację obiektu wywołującego. W takich przypadkach wyciąganie odpowiednich wniosków zależy od mechanizmów weryfikujących zaimplementowanych w wywoływanych metodach. Przykładowo, obiekty klasy `System.Security.Permissions.PublisherIdentityPermission` reprezentują wydawcę oprogramowania. Istnieją też klasy implementujące interfejs `IPermission`, które reprezentują takie dane jak adres URL, z którego pochodzi dany podzespół .NET, lub nazwę samego podzespołu.

Klasy reprezentujące informacje, które pozwalają zidentyfikować obiekt wywołujący, zwykle nie implementują interfejsu `IPermission`. Oznacza to, że klasy, które potrzebują egzemplarzy interfejsu `IPermission` potwierdzających tego rodzaju informacje, muszą te egzemplarze tworzyć w oparciu o otrzymywane obiekty. Twórcy platformy .NET Framework zastosowali wzorzec projektowy Factory Method, aby zwolnić klasy korzystające z tworzonych w ten sposób egzemplarzy interfejsu `IPermission` z obowiązku obsługi różnych rodzajów obiektów identyfikujących.

Wszystkie klasy identyfikujące oferowane w ramach platformy .NET Framework implementują interfejs `System.Security.Policy.IIdentityPermissionFactory`. Interfejs `IIdentityPermissionFactory` deklaruje funkcję `CreateIdentityPermission`, która otrzymuje na wejściu (za pośrednictwem argumentu) kolekcję identyfikatorów wymagających potwierdzenia przez egzemplarz interfejsu `IPermission` i która zwraca egzemplarz interfejsu `IPermission` potwierdzający odpowiednie uprawnienia.

Przykład kodu źródłowego

Przypuśćmy, że opracowujemy aplikację przetwarzającą rekordy, które zapisano w pliku dziennika utworzonym przez system obsługujący punkt sprzedaży[1]. Dla każdego wiersza drukowanego na paragonach kas fiskalnych istnieje odpowiedni rekord w pliku

[1] Sercem punktu sprzedaży jest nowoczesna kasa fiskalna.

dziennika. Nasza aplikacja będzie odczytywała wszystkie rekordy zarejestrowane w pliku dziennika i na ich podstawie generowała zestawienia na poziomie transakcji.

Nasza aplikacja będzie musiała prawidłowo współpracować z punktami sprzedaży zbudowanymi na bazie rozmaitych kas fiskalnych dostarczonych przez różnych producentów. Ponieważ pliki dziennika generowane przez kasy fiskalne różnych producentów mają odmienny format, w przypadku tej aplikacji szczególnie istotne jest zapewnienie niezależności klasy odpowiedzialnej za generowanie raportów od formatu przetwarzanych plików dzienników.

Formaty wszystkich plików dzienników (niezależnie od producenta kasy fiskalnej) składają się z sekwencji rekordów. Typy tych rekordów są do siebie dość podobne. Zdecydowaliśmy, że klasy generujące zestawienia i raporty powinny być niezależne od formatów plików dzienników, zatem muszą wewnętrznie reprezentować odczytaną zawartość tych plików w formie sekwencji obiektów, z których każdy odpowiada jednemu rekordowi dziennika. Mając to na uwadze, projektujemy zbiór klas właściwych dla poszczególnych rodzajów rekordów występujących w plikach dzienników.

W przykładzie prezentowanym w niniejszym podrozdziale skoncentrujemy się na sposobie, w jaki nasza aplikacja będzie tworzyła obiekty reprezentujące rekordy odczytane z plików dzienników. W przykładzie użyjemy obu najważniejszych form wzorca projektowego Factory Method.

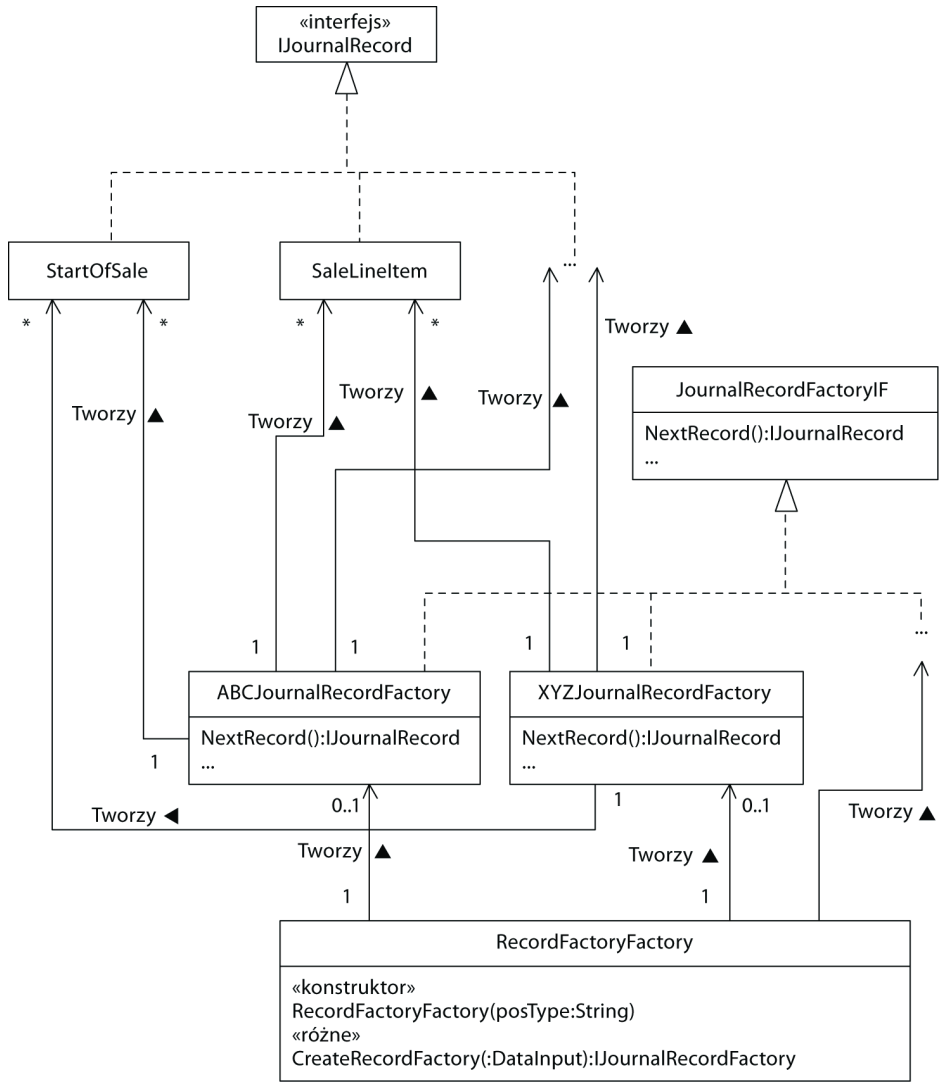
- Aplikacja wczytująca rekordy z pliku dziennika wykorzystuje specjalny obiekt fabryki, który tworzy obiekty reprezentujące rekordy pliku dziennika (po jednym dla każdego rekordu). Za każdym razem, gdy obiekt fabryki otrzymuje żądanie utworzenia nowego obiektu, wybiera dla tego obiektu właściwą klasę w zależności od informacji zawartych w samym rekordzie. Wspomniana klasa fabryki dobrze ilustruje mechanizm określania klasy tworzonych egzemplarzy w czasie wykonywania programu.
- Także klasa samego obiektu fabryki, który odpowiada za tworzenie obiektów reprezentujących rekordy, zależy od rodzaju przetwarzanego przez aplikację pliku dziennika. Ponieważ wybór klasy obiektów fabryki zależy od informacji zawartych w ustawieniach konfiguracyjnych, do ich tworzenia wykorzystuje się jeszcze jeden obiekt fabryki, który realizuje to zadanie bezpośrednio po wczytaniu przez aplikację ustawień konfiguracyjnych.

Na rysunku 4.5 przedstawiono diagram klas składających się na opisany przykład. Omówienie klas i interfejsów widocznych na tym rysunku znajdziesz poniżej.

`IJournalRecord`. Obiekt reprezentujący zawartość pojedynczego rekordu dziennika jest egzemplarzem klasy implementującej interfejsu `IJournalRecord`. Na diagramie z rysunku 4.5 przedstawiono co prawda tylko dwie takie klasy, jednak kompletna aplikacja mogłaby ich zawierać dużo więcej.

`StartOfSale`. Egzemplarze tej klasy reprezentują te rekordy dzienników, które wskazują na początki kolejnych transakcji sprzedaży.

`SaleLineItem`. Egzemplarze tej klasy reprezentują rekordy dzienników zawierające wszystkie szczegółowe informacje na temat sprzedawanych towarów, które są umieszczane na wydrukach kasy fiskalnej.



Rysunek 4.5. *Klasy fabryk związane z plikami dzienników*

`IJournalRecordFactory`. Interfejs `IJournalRecordFactory` jest implementowany przez wszystkie klasy odpowiedzialne za odczytywanie informacji zawarte w rekordach pliku dziennika i konwertowanie ich na egzemplarze interfejsu `IJournalRecord`. Obiekty klas implementujących interfejs `IJournalRecordFactory` tworzą obiekty takich klas implementujących interfejs `IJournalRecord` jak `StartOfSale` czy `SaleLineItems`.

`ABCJournalRecordFactory`. Egzemplarze tej klasy odpowiadają za właściwą interpretację formatu plików dzienników generowanych przez systemy sprzedaży fikcyjnej firmy ABC — odczytują kolejne rekordy tych dzienników za pośrednictwem obiektu klasy `System.IO.BinaryReader` i umieszczają ich zawartość w obiektach klasy implementującej interfejs `IJournalRecord`.

`XYZJournalRecordFactory`. Egzemplarze tej klasy odpowiadają za właściwą interpretację formatu plików dzienników generowanych przez systemy sprzedaży fikcyjnej firmy XYZ — odczytują kolejne rekordy tych dzienników za pośrednictwem obiektu klasy `System.IO.BinaryReader` i umieszczają ich zawartość w obiektach klasy implementującej interfejs `IJournalRecord`.

`RecordFactoryFactory`. Konstruktor tej klasy otrzymuje na wejściu nazwę typu systemu sprzedaży. Obiekt konstruowany przez ten konstruktor jest następnie wykorzystywany do tworzenia egzemplarzy właściwej klasy implementującej interfejs `IJournalRecordFactory` dla wskazanego typu systemu sprzedaży.

Po lekturze kolejnego podrozdziału być może zwróciłeś uwagę na pewną zbieżność strukturalną pomiędzy tym przykładem, w którym zastosowano obie formy wzorca projektowego Factory Method, a omówionym bezpośrednio po nim wzorcem projektowym Abstract Factory. Cechą charakterystyczną wzorca projektowego Abstract Factory jest traktowanie obiektów klienckich przez pryzmat interfejsów implementowanych przez klasy tworzonych obiektów, a nie przez zawartość tych obiektów.

Przeanalizujmy teraz kod implementujący omówiony projekt. W pierwszej kolejności przyjrzymy się listingowi klasy `RecordFactoryFactory`:

```
Imports System
Imports System.IO
Imports System.Reflection

' Podczas tworzenia egzemplarza tej klasy jej konstruktor
' otrzymuje na wejściu łańcuch określający typ systemu
' stosowanego w danym punkcie sprzedaży, dla którego
' tworzony obiekt będzie pełnił funkcję fabryki. Nowa fabryka
' wykorzystuje obiekt klasy ConstructorInfo, za pomocą
' którego można konstruować obiekty odpowiedniej klasy.
Public Class RecordFactoryFactory
    Private factoryConstructor As ConstructorInfo

    ' Typy systemów punktów sprzedaży.
    Public Enum PosTypes
        ABC
    End Enum

    ' Konstruktor
    '
    ' posType - typ systemu punktu sprzedaży, dla którego ten obiekt
    ' będzie tworzył egzemplarz interfejsu IJournalRecordFactory.
    Public Sub New(ByVal posType As PosTypes)
        Dim args() As Type = {GetType(StreamReader)}
        Dim factoryClass As Type
        If posType = PosTypes.ABC Then
            factoryClass = GetType(ABCJournalRecordFactory)
        Else
            Dim msg As String = "Nieznany typ punktu sprzedaży: " &
            posType.ToString
            Throw New ApplicationException(msg)
        End If
        Try
            factoryConstructor = factoryClass.GetConstructor(args)
```

```
        Catch e As Exception
            Dim msg As String = "Błąd podczas konstruowania obiektu fabryki"
            Throw New ApplicationException(msg, e)
        End Try
    End Sub

    ' Metoda tworzy obiekty klasy implementującej interfejs IJournalRecordFactory.
    Public Function CreateFactory(ByVal input As StreamReader) _
                    As IJournalRecordFactory
        Dim args() As Object = {input}
        Dim factory As Object
        Try
            factory = factoryConstructor.Invoke(args)
        Catch e As Exception
            Dim msg As String = "Błąd podczas tworzenia fabryki"
            Throw New ApplicationException(msg, e)
        End Try
        Return CType(factory, IJournalRecordFactory)
    End Function
End Class 'RecordFactoryFactory

' Ten interfejs jest implementowany przez klasy obiektów reprezentujących
' zawartość rekordu dziennika.
Public Interface IJournalRecord
End Interface
```

Poniższy listing przedstawia kod interfejsu `IJournalRecordFactory`:

```
' Ten interfejs jest implementowany przez klasy odpowiedzialne
' za tworzenie obiektów, których obiekty reprezentują zawartość
' pojedynczych rekordów pliku dziennika.
Public Interface IJournalRecordFactory
        ' Zwraca obiekt reprezentujący następny rekord
        ' pliku dziennika.
        Function NextRecord() As IJournalRecord
End Interface
```

Poniżej przedstawiono kod klasy `ABCJournalRecordFactory`, która implementuje interfejs `IJournalRecordFactory` dla typu systemu punktu sprzedaży nazwanego `"ABC"`:

```
' Ta klasa implementuje interfejs IJournalRecordFactory
' dla typu systemu punktu sprzedaży nazwanego "ABC".
Public Class ABCJournalRecordFactory
    Implements IJournalRecordFactory

    ' Typy rekordów.
    Private Shared SALE_LINE_ITEM As String = "17"
    Private Shared START_OF_SALE As String = "4"

    Private input As StreamReader

    ' Licznik rekordów.
    Private sequenceNumber As Integer = 0

    Sub New(ByVal i As StreamReader)
        input = i
    End Sub
```

```
    ' Zwraca obiekt reprezentujący następny rekord zapisany
    ' w pliku dziennika.
    Public Function NextRecord() As IJournalRecord _
      Implements IJournalRecordFactory.NextRecord
        Dim record As String = input.ReadLine()
        Dim tokens As String() = record.Split(New Char() {","c})
        sequenceNumber += 1

        If tokens(0) = START_OF_SALE Then
            Return CType(constructStartOfSale(tokens), _
                         IJournalRecord)
        ElseIf tokens(0) = SALE_LINE_ITEM Then
            Return CType(constructSaleLineItem(tokens), _
                         IJournalRecord)
        End If
        Throw New IOException("Unknown record type")
    End Function 'NextRecord

    Private _
    Function constructStartOfSale(ByVal tok() As String) _
             As StartOfSale
        Dim index As Integer = 1
        Dim transactionID As String = tok(index)
        index += 1 'Pomija identyfikator.
        index += 1 'Pomija wskaźnik trybu.
        Dim timestamp As DateTime = DateTime.Parse(tok(index))
        index += 1 'Pomija egzemplarz klasy DateTime.
        Dim terminalID As String = tok(index)
        Return New StartOfSale(terminalID, sequenceNumber, _
                               timestamp, transactionID)
    End Function 'constructStartOfSale

    Private _
    Function constructSaleLineItem(ByVal tok() As String) _
             As SaleLineItem
        Dim index As Integer = 1
        Dim transactionID As String = tok(index)
        index += 1 'Pomija identyfikator.
        index += 1 'Pomija wskaźnik trybu.
        Dim timestamp As DateTime = DateTime.Parse(tok(index))
        index += 1 'Pomija egzemplarz klasy DateTime.
        Dim terminalID As String = tok(index)
        Return New SaleLineItem(terminalID, sequenceNumber, _
                                timestamp, transactionID)
    End Function 'constructSaleLineItem
End Class 'ABCJournalRecordFactory
```

Wzorce pokrewne

Hashed Adapter Objects. Wzorzec projektowy Hashed Adapter Objects może być stosowany w implementacjach wzorca projektowego Factory Method. Takie rozwiązanie jest korzystne, jeśli zbiór klas, których egzemplarze są tworzone przez obiekt fabryki, może ulegać zmianom w czasie wykonywania programu.

Abstract Factory. Wzorzec projektowy Factory Method często jest wykorzystywany do konstruowania pojedynczych obiektów realizujących określone zadania w sytuacji, gdy obiekt żądający tych obiektów nie dysponuje wiedzą o ich klasach. Jeśli musisz stworzyć spójny zbiór tego rodzaju obiektów, najlepszym wyjściem jest użycie wzorca projektowego Abstract Factory.

Template Method. Kiedy wzorzec projektowy Factory Method jest implementowany jako mechanizm pozwalający określać typy tworzonych obiektów na podstawie informacji zawartych w ustawieniach konfiguracyjnych, zwykle jest stosowany łącznie ze wzorcem Template Method.

Prototype. Wzorzec projektowy Prototype opisuje alternatywny funkcjonowania obiektów klienckich, które nie dysponują szczegółowymi informacjami na temat procedur konstruowania obiektów przez siebie wykorzystywanych.

Strategy. Jeśli rozważasz użycie wzorca projektowego Factory Method do modyfikowania zachowań obiektów, być może lepszym rozwiązaniem będzie zastosowanie wzorca projektowego Strategy.

Abstract Factory

Wzorzec projektowy Abstract Factory, znany także jako Kit lub Toolkit, po raz pierwszy został zaproponowany w książce *Design Patterns: Elements of Reusable Object-Oriented Software* [Gamma, 1995].

Streszczenie

Mając do dyspozycji zbiór wzajemnie powiązanych interfejsów, musimy znaleźć sposób tworzenia obiektów implementujących te interfejsy z wykorzystaniem odpowiedniego zbioru konkretnych klas. Wzorzec projektowy Abstract Factory (wytwórni abstrakcji) może się bardzo przydać, jeśli chcemy, by nasz program współpracował ze zróżnicowanym zbiorem skomplikowanych środowisk zewnętrznych takich jak różne systemy okien oferujące zbliżoną funkcjonalność.

Kontekst

Przypuśćmy, że stoimy przed zadaniem budowy szkieletu interfejsu użytkownika, który będzie pracował ponad wieloma różnymi systemami okien, np. oknami systemu operacyjnego Windows, Motif czy MacOS. Szkielet musi prawidłowo współpracować z każdą platformą i obsługiwać właściwy dla tej platformy wygląd i sposób obsługi. Budowany szkielet zawiera po jednej klasie abstrakcyjnej dla każdego typu kontrolek

(pola tekstowego, przycisku, listy itp.) oraz konkretnych podklas tych klas abstrakcyjnych dla wszystkich obsługiwanych platform. Aby mieć pewność, że oprogramowanie budowane na bazie tego szkieletu będzie niezawodne, wszystkie tworzone obiekty kontrolek muszą być zgodne z docelową platformą. Wzorzec projektowy Abstract Factory umożliwia efektywną realizację właśnie tego wymagania.

Klasa fabryki abstrakcyjnej definiuje funkcje obsługujące operacje tworzenia egzemplarzy wszystkich klas abstrakcyjnych reprezentujących kontrolki graficznego interfejsu użytkownika. Konkretne wytwórnie (fabryki) to po prostu konkretne podklasy klasy abstrakcyjnej fabryki, które implementują jej funkcje tworzące egzemplarze konkretnych klas kontrolek dla określonych platform.

Jeśli spojrzymy na tę strukturę bez wchodzenia w szczegóły, zauważymy, że klasa abstrakcyjnej fabryki i jej konkretne podklasy tworzą zbiory konkretnych klas właściwe dla różnych, ale wzajemnie powiązanych produktów. Analizę szerszej perspektywy zilustrujemy przykładem nieco innej sytuacji.

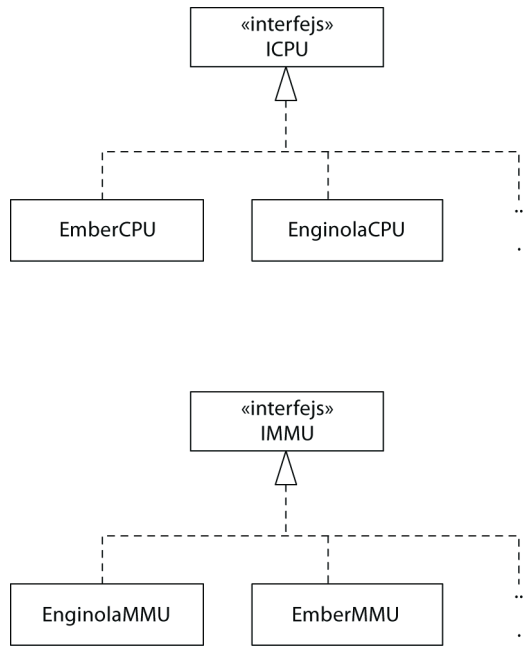
Przypuśćmy, że piszemy program, którego zadaniem będzie zdalna diagnostyka komputerów wyprodukowanych przez firmę nazwaną Stellar Microsystems. W związku z rozwojem technologicznym architektura kolejnych modeli komputerów oferowanych przez tę firmę nieznacznie różni się od modeli sprzedawanych wcześniej. Najstarsze komputery tej firmy budowano na bazie procesorów koncernu Enginola, które obsługiwały tradycyjny, dość skomplikowany zbiór rozkazów. Od tego czasu firma wprowadziła na rynek wiele kolejnych generacji komputerów wykorzystujących jej własne architektury RISC, nazwane odpowiednio ember, super-ember oraz ultra-ember. Najważniejsze komponenty stosowane podczas budowy tych komputerów odpowiadały za podobne funkcje, ale każdy model składał się z nieco innego zbioru tych komponentów.

Aby nasz program „wiedział", które testy powinien wykonać i jak należy interpretować uzyskane wyniki, będzie musiał tworzyć egzemplarze klas właściwych dla każdego z kluczowych komponentów diagnozowanego komputera. Poszczególne klasy tych obiektów odpowiadają konkretnym typom testowanych komponentów. Oznacza to, że musimy dysponować osobnym zbiorem klas dla każdej ze stosowanych przez tę firmę architektur sprzętowych. Dla każdego z typów komponentów składających się na te architektury musi istnieć odpowiednia klasa w każdym z tych zbiorów.

Na rysunku 4.6 przedstawiono diagram klas ilustrujący organizację klas składających się na opisany system diagnostyczny, który zaprojektowano z myślą o różnych rodzajach komponentów komputerowych. W prezentowanym diagramie uwzględniono co prawda tylko dwa typy komponentów, jednak organizacja klas dla większej liczby rodzajów komponentów byłaby identyczna. Dla każdego typu komponentu zdefiniowano osobny interfejs. Dla każdej obsługiwanej architektury komputerowej muszą istnieć osobne zbiory klas, które implementują interfejsy właściwe dla poszczególnych komponentów.

Struktura systemu diagnostycznego, którą przedstawiono na rysunku 4.6, dobrze ilustruje problem rozwiązywany za pomocą wzorca projektowego Abstract Factory. Musimy znaleźć sposób właściwego zorganizowania procesu tworzenia obiektów diagnostycznych. Chcemy, aby klasy korzystające z mechanizmów zaimplementowanych w klasach

Rysunek 4.6. *Klasy systemu diagnostycznego*



diagnozujących były od tych klas niezależne. Moglibyśmy oczywiście użyć do tworzenia obiektów diagnostycznych wzorca projektowego Factory Method, jednak wciąż stalibyśmy przed problemem, którego wzorzec Factory Method w żaden sposób nie rozwiązuje.

Chcemy mieć pewność, że wszystkie obiekty wykorzystywane przez nas do diagnozowania komputera będą odpowiednio dostosowane do jego architektury. Gdybyśmy użyli wzorca projektowego Factory Method, klasy korzystające z klas diagnozujących musiałyby każdorazowo przekazywać poszczególnym fabrykom informacje, która architektura komputerowa ma być przedmiotem działania tworzonych obiektów diagnostycznych. Naszym celem jest znalezienie spójnego sposobu zapewniania, że wszystkie tego rodzaju obiekty wykorzystywane dla pojedynczego komputera będą odpowiadały właściwej architekturze, ale bez konieczności wprowadzania dodatkowych zależności do klas, które tych obiektów używają.

Zalety

- ☺ System pracujący z wieloma różnymi produktami powinien funkcjonować w sposób całkowicie niezależny od poszczególnych produktów.
- ☺ Powinniśmy mieć możliwość takiego konfigurowania naszego systemu, aby prawidłowo współpracował z jednym lub wieloma wybranymi składowymi szerszych rodzin produktów.
- ☺ Egzemplarze klasy, które w założeniu mają obsługiwać konkretny produkt, powinny być stosowane łącznie i tylko w odniesieniu do produktu, z myślą o którym zostały zaprojektowane i zaimplementowane. To ograniczenie powinno być ściśle przestrzegane i odpowiednio wymuszane.

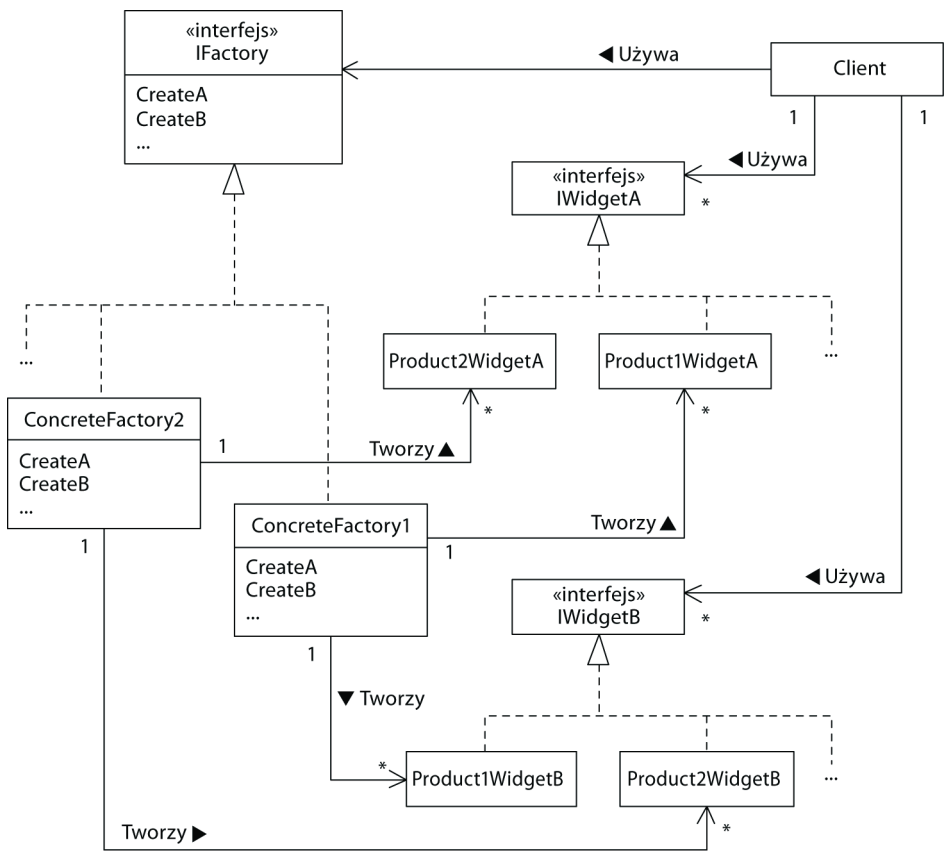
☺ Mechanizmy obsługujące produkty zaimplementowane w pozostałych częściach systemu powinny funkcjonować możliwie niezależnie od tego, które konkretne klasy bezpośrednio komunikują się z tymi produktami.

☺ System powinien być rozszerzalny do tego stopnia, aby w razie potrzeby radzić sobie z kolejnymi produktami przez stosowanie dodatkowych zbiorów klas — tego rodzaju rozbudowa nie powinna wymagać więcej niż dopisania kilku wierszy kodu.

☹ Interfejs implementowany przez klasę nie wystarczy do wyróżnienia tej klasy wśród innych klas, których egzemplarze mogą tworzyć obiekty.

Rozwiązanie

Na rysunku 4.7 przedstawiono diagram klas, który pokazuje role odgrywane przez poszczególne klasy i interfejsy wchodzące w skład wzorca projektowego Abstract Factory.



Rysunek 4.7. *Przykład wzorca projektowego Abstract Factory*

Poniżej opisano poszczególne role odgrywane we wzorcu projektowym Abstract Factory przez klasy i interfejsy z rysunku 4.7.

`IWidgetA, IWidgetB, ...` Interfejsy w tej roli odpowiadają usłudze lub elementowi funkcjonalności danego produktu. Klasy, które implementują jeden z tych interfejsów, automatycznie muszą obsługiwać usługę lub element funkcjonalności reprezentowaną przez implementowany interfejs.

Z uwagi na ograniczoną ilość miejsca na stronie przedstawiony diagram zawiera tylko dwa takie interfejsy. W większości implementacji wzorca projektowego Abstract Factory stosuje się znacznie więcej niż dwa tego rodzaju interfejsy. Liczba tych interfejsów odpowiada liczbie usług lub elementów funkcjonalności wykorzystywanego produktu.

`Product1WidgetA, Product2WidgetA, ...` Klasy występujące w tej roli odpowiadają konkretnemu elementowi funkcjonalności określonego produktu. Ogólnie, można te klasy interpretować jak konkretne struktury reprezentujące poszczególne elementy funkcjonalności.

`Client`. Klasy w tej roli wykorzystują konkretne klasy elementów funkcjonalności do żądania lub uzyskiwania usług ze strony produktu, który jest przedmiotem działań danego klienta. Klasy klienckie współpracują z konkretnymi klasami elementów funkcjonalności wyłącznie za pośrednictwem interfejsu `IWidget`. Oznacza to, że klasy klienckie są niezależne od konkretnych klas elementów funkcjonalności, z którymi współpracują.

`IFactory`. Interfejsy pełniące tę rolę deklarują metody odpowiedzialne za tworzenie egzemplarzy konkretnych klas elementów funkcjonalności. Każda z tych metod zwraca egzemplarz klasy implementującej inny interfejs występujący w roli `IWidget`. Klasy odpowiedzialne za tworzenie konkretnych obiektów reprezentujących elementy funkcjonalności koniecznie muszą implementować jeden z interfejsów `IFactory`.

`ConcreteFactory1, ConcreteFactory2, ...` Klasy występujące w tej roli implementują interfejs `IFactory` i odpowiadają za tworzenie egzemplarzy konkretnych klas elementów funkcjonalności produktów. Każda z klas występujących w tej roli odpowiada innemu produktowi, z którym może współpracować klasa kliencka (występująca w roli `Client`). Każda funkcja klasy `ConcreteFactory` tworzy inny rodzaj konkretnego elementu funkcjonalności. Warto jednak pamiętać, że zadaniem wszystkich tworzonych w ten sposób konkretnych obiektów elementów funkcjonalności jest współpraca z produktem właściwym dla danej klasy `ConcreteFactory`.

Klasy klienckie, które wywołują te funkcje, nie powinny dysponować żadną bezpośrednią wiedzą na temat konkretnych klas fabryk — dostęp do egzemplarzy tych klas zawsze musi się odbywać za pośrednictwem interfejsu `IFactory`.

Jeśli nadal nie są dla Ciebie jasne relacje łączące poszczególne elementy, możesz skorzystać z zestawienia przedstawionego w tabeli 4.1. Tabela jest w istocie macierzą reprezentującą poszczególne produkty, z którymi nasz program będzie musiał współpracować, oraz wspólne dla tych produktów elementy funkcjonalności.

Tabela 4.1. *Macierz produktów i elementów funkcjonalności*

	Product1	Product2	Product3	
WidgetA	Product1WidgetA	Product2WidgetA	Product3WidgetA	...
WidgetB	Product1WidgetB	Product2WidgetB	Product3WidgetB	...
WidgetC	Product1WidgetC	Product2WidgetC	Product3WidgetC	...
...	...	...	...	

Każda kolumna tabeli 4.1 odpowiada innemu produktowi. Każdy wiersz tej tabeli odpowiada innemu elementowi funkcjonalności produktów. Dla wszystkich elementów funkcjonalności produktu należy zdefiniować osobny interfejs, który będzie pośredniczył we współpracy z danym produktem. Ciało tabeli 4.1 zawiera nazwy klas implementujących konkretne interfejsy właściwe dla konkretnych produktów. Celem wzorca projektowego Abstract Factory jest zagwarantowanie korzystania wyłącznie z klas wchodzących w skład kolumny tej macierzy, czyli klas właściwych dla produktu, z którym współpracujemy.

Nasze przekonanie odnośnie korzystania wyłącznie z klas należących do właściwej kolumny macierzy wynika z faktu, że za tworzenie tych obiektów odpowiada obiekt klasy `ConcreteFactory` właściwy dla produktu, z którym nasz program w danej chwili współpracuje. Każda klasa `ConcreteFactory` definiuje metody tworzące obiekty, które implementują interfejsy rozmaitych elementów funkcjonalności tylko do jednego produktu.

Implementacja

Największym wyzwaniem, przed jakim staje programista implementujący wzorzec projektowy Abstract Factory, jest opracowanie mechanizmu wykorzystywanego przez klasy klienckie do tworzenia i uzyskiwania dostępu do egzemplarzy interfejsu `IFactory`. Z najprostszą sytuacją mamy do czynienia wtedy, gdy przez cały czas działania programu obiekty klienckie muszą współpracować tylko z jednym produktem. W takim przypadku wystarczy użyć klasy definiującej zmienną statyczną reprezentującą pojedynczy obiekt klasy `ConcreteFactory` wykorzystywany przez wszystkie obiekty klienckie programu. Taka zmienna albo powinna być składową publiczną, albo jej wartość powinna być udostępniana za pośrednictwem publicznej właściwości statycznej.

Jeśli obiekt abstrakcyjnej fabryki musi wybrać jeden z wielu konkretnych obiektów fabryk na podstawie informacji przekazanych przez obiekt klienta, do zaimplementowania mechanizmu dokonującego tego rodzaju wyboru możemy użyć wzorca projektowego Factory Method.

Skutki stosowania

☺ Klasy klienckie są niezależne od wykorzystywanych przez siebie konkretnych klas elementów funkcjonalności.

☺ Dodawanie (celowo nie użyto słowa „pisanie") klas współpracujących z kolejnymi produktami jest wyjątkowo proste. Klasa obiektu konkretnej fabryki zwykle wymaga stosowania odpowiedniej referencji w jednym miejscu. Oznacza to, że ewentualna wymiana konkretnej fabryki na inną (współpracującą z określonym produktem) jest bardzo proste.

☺ Ponieważ wzorzec projektowy Abstract Factory wymusza na klasach klienckich korzystanie z pośrednictwa interfejsu `IFactory` podczas tworzenia konkretnych obiektów reprezentujących elementy funkcjonalności, możemy być pewni, że obiekty klienckie korzystają ze spójnego zbioru obiektów współpracujących z produktem.

☹ Najważniejszą wadą wzorca projektowego Abstract Factory jest zwiększona pracochłonność związana z pisaniem nowego zbioru klas dla każdego produktu reprezentowanego przez osobny interfejs. Okazuje się, że czasochłonne może być także rozszerzanie zbioru elementów funkcjonalności, który będzie sprawdzany przez istniejący zbiór klas w ramach wykonywanych testów produktów.

Dodanie obsługi nowego produktu wiąże się z koniecznością napisania kompletnego zbioru klas dla konkretnych elementów funkcjonalności oferowanych przez ten nowy produkt. Dla każdego interfejsu `IWidget` należy napisać konkretną klasę elementu funkcjonalności. Im więcej istnieje interfejsów `IWidget`, tym więcej pracy czeka programistę dodającego obsługę dodatkowego produktu.

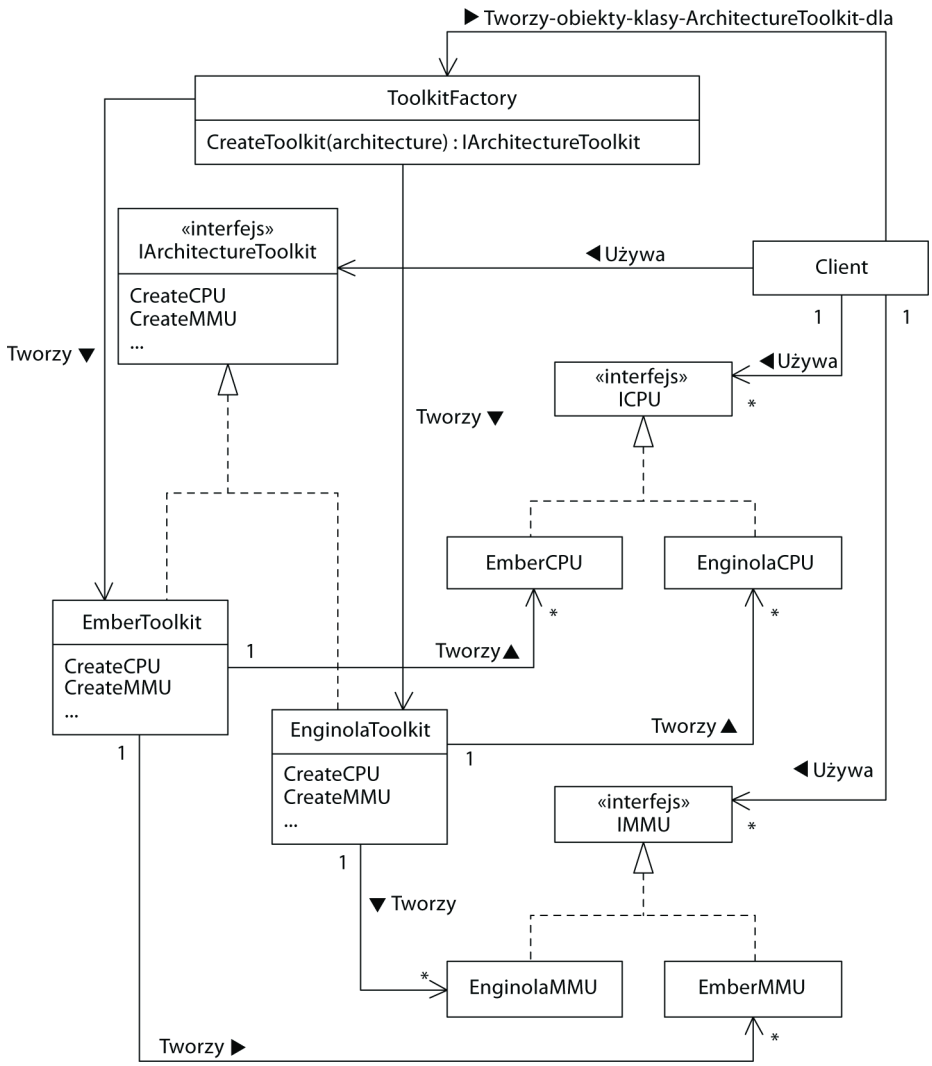
Także zapewnienie dostępu do dodatkowego elementu funkcjonalności produktów reprezentowanych przez interfejsy może wymagać sporego wysiłku, jeśli takich obsługiwanych produktów jest wiele. Implementacja dostępu do nowego elementu funkcjonalności polega na napisaniu interfejsu `IWidget` (właściwego tylko dla tego elementu) oraz po jednej konkretnej klasie elementu funkcjonalności dla każdego produktu.

☹ Obiekty klienckie mogą dodatkowo wymagać organizowania klas reprezentujących elementy funkcjonalności w formie odpowiednich hierarchii. Wzorzec projektowy Abstract Factory w swojej podstawowej wersji nie przewiduje takiej możliwości, ponieważ wymaga, by konkretne klasy elementów funkcjonalności były organizowane w ramach hierarchii klas, która jest niezależna od obiektów klienckich. Można tę trudność przezwyciężyć łącząc ten wzorzec ze wzorcem projektowym Bridge (wzorcem mostu).

Należy utworzyć taką hierarchię klas elementów funkcjonalności niezależnych od produktów, która będzie spełniała kryteria narzucone przez klasy klienckie. Każda z takich klas niezależnych od produktu i reprezentujących element funkcjonalności deleguje logikę właściwą dla produktu do odpowiednich klas implementujących interfejs `WidgetIF`.

Przykład kodu źródłowego

Podczas analizy przykładu ilustrującego praktyczne zastosowania wzorca projektowego Abstract Factory wrócimy do problemu omówionego w punkcie „Kontekst". Na rysunku 4.8 przedstawiono rozbudowany diagram klas, w którym uwzględniono organizację właściwą dla tego wzorca projektowego.



Rysunek 4.8. *Klasy diagnostyczne zbudowane z wykorzystaniem wzorca projektowego Abstract Factory*

Egzemplarz klasy `Client` zarządza zdalnym procesem diagnostycznym. Po określeniu architektury komputera będącego przedmiotem diagnozy egzemplarz klasy `Client` przekazuje informacje o typie architektury do funkcji `CreateToolkit` obiektu klasy `ToolkitFactory`. Funkcja zwraca egzemplarz jednej z klas implementujących interfejs

IArchitectureToolkit, czyli klasę EmberToolkit lub EnginolaToolkit — wybór właściwej klasy zależy od wykrytej architektury diagnozowanego komputera. Obiekt klasy Client może następnie użyć egzemplarza interfejsu IArchitectureToolkit do tworzenia obiektów modelujących procesory (CPU), jednostki zarządzania pamięcią (MMU) i innych komponentów diagnozowanej architektury.

Poniższy listing zawiera kod implementujący projekt systemu zdalnego diagnozowania komputerów, którego ogólny schemat przedstawiono na rysunku 4.8.

```
' Klasy testujące wszystkie typy procesorów muszą implementować
' ten interfejs.
Public Interface ICpu
    ' Typ architektury obsługiwanej przez ten obiekt.
    ReadOnly Property Type() As String

    ...
End Interface

' Klasy zaprojektowane z myślą o testowaniu wszystkich rodzajów jednostek
' zarządzania pamięcią muszą implementować ten interfejs.
Public Interface IMmu
    ' Typ architektury testowanej przez ten obiekt.
    ReadOnly Property Type() As String

    ...
End Interface

' Klasa zaprojektowana z myślą o testowaniu procesora komputera, który
' zbudowano na bazie architektury Ember.
Public Class EmberCpu
    Implements ICpu

    Public ReadOnly Property Type() As String _
        Implements ICpu.Type
        Get
            Return "Ember"
        End Get
    End Property

    ...
End Class

' Klasa zaprojektowana z myślą o testowaniu jednostki zarządzania pamięcią (MMU)
' komputera, który zbudowano na bazie architektury Ember.
Public Class EmberMmu
    Implements IMmu

    Public ReadOnly Property Type() As String _
        Implements IMmu.Type
        Get
            Return "Ember"
        End Get

        ...
    End Property
End Class
```

```
' Klasa zaprojektowana z myślą o testowaniu komputerów zbudowanych
' na bazie architektury Ember.
Public Class EmberToolkit
    Implements IArchitectureToolkit

    ' Zwraca obiekt przystosowany do testowania procesora komputera
    ' zbudowanego na bazie architektury Ember.
    Public Function CreateCpu() As ICpu _
        Implements IArchitectureToolkit.CreateCpu
        Return New EmberCpu()
    End Function

    '  Zwraca obiekt przystosowany do testowania jednostki zarządzania
    ' pamięcią komputera zbudowanego na bazie architektury Ember.
    Public Function CreateMmu() As IMmu _
        Implements IArchitectureToolkit.CreateMmu
        Return New EmberMmu()
    End Function
    ...
End Class

' Klasa zaprojektowana z myślą o testowaniu procesora komputera, który
' zbudowano na bazie architektury Enginola.
Public Class EnginolaCpu
    Implements ICpu

    Public ReadOnly Property Type() As String _
        Implements ICpu.Type
        Get
            Return "Enginola"
        End Get
    End Property
End Class

' Klasa zaprojektowana z myślą o testowaniu jednostki zarządzania pamięcią (MMU)
' komputera, który zbudowano na bazie architektury Enginola.
Public Class EnginolaMmu
    Implements IMmu

    Public ReadOnly Property Type() As String _
        Implements IMmu.Type
        Get
            Return "Enginola"
        End Get
    End Property

    ...
End Class

' Klasa zaprojektowana z myślą o testowaniu komputerów zbudowanych
' na bazie architektury Enginola.
Public Class EnginolaToolkit
    Implements IArchitectureToolkit

    ' Zwraca obiekt przystosowany do testowania procesora komputera
    ' zbudowanego na bazie architektury Enginola.
    Public Function CreateCpu() As ICpu _
```

```
        Implements IArchitectureToolkit.CreateCpu
        Return New EnginolaCpu()
    End Function

    ' Zwraca obiekt przystosowany do testowania jednostki zarządzania
    ' pamięcią komputera zbudowanego na bazie architektury Enginola.
    Public Function CreateMmu() As IMmu _
        Implements IArchitectureToolkit.CreateMmu
        Return New EnginolaMmu()
    End Function

    ...
End Class

' Abstrakcyjne klasy narzędziowe odpowiedzialne za tworzenie obiektów
' testujących różne komponenty tej samej architektury muszą implementować
' ten interfejs.
Public Interface IArchitectureToolkit
    Function CreateCpu() As ICpu
    Function CreateMmu() As IMmu
    ...
End Interface

' Poniższa klasa odpowiada za tworzenie egzemplarzy klasy, która
' implementuje interfejs IArchitectureToolkit dla konkretnej
' architektury.
'
' Klasa ToolkitFactory jest singletonem.
Public Class ToolkitFactory
    ' Jedyny egzemplarz tej klasy.
    Private Shared myInstance As New ToolkitFactory()

    ' Symboliczne nazwy identyfikujące obsługiwane architektury komputerów.
    Public Enum ComputerArchitecture
        ENGINOLA = 900
        EMBER = 901
    End Enum

    ' Zwraca jedyny egzemplarz tej klasy.
    Public Shared ReadOnly Property Instance() As ToolkitFactory
        Get
            Return myInstance
        End Get
    End Property

    '
    ' Zwraca nowo utworzony obiekt klasy implementującej interfejs
    ' ArchitectureToolkitIF dla danej architektury komputera.
    '
    Public Function CreateToolkit(ByVal architecture As ComputerArchitecture) _
        As IArchitectureToolkit
        Select Case architecture
            Case ComputerArchitecture.ENGINOLA
                Return New EnginolaToolkit()

            Case ComputerArchitecture.EMBER
                Return New EmberToolkit()
```

```
        End Select
        Throw New System.ArgumentException("ArchId=" & architecture.ToString)
    End Function
End Class

' Bardzo prosta klasa kliencka.
Public Class Client
    Public Shared Sub Main()
        Dim myFactory As ToolkitFactory
        myFactory = ToolkitFactory.Instance
        Dim af As IArchitectureToolkit
        af = myFactory.CreateToolkit(ToolkitFactory.ComputerArchitecture.EMBER)
        Dim cpu As ICpu = af.CreateCpu()
        System.Console.WriteLine("Utworzono obiekt narzędziowy dla procesora " &
cpu.Type)
    End Sub
End Class
```

Wzorce pokrewne

Factory Method. W przykładzie przedstawionym w poprzednim punkcie abstrakcyjna klasa fabryki wykorzystywała wzorzec projektowy Factory Method podczas decydowania o wyborze konkretnego obiektu fabryki dla klasy klienckiej.

Singleton. Konkretne klasy fabryk zwykle są implementowane w formie singletonów (reifikacji wzorca projektowego Singleton).

Builder

Wzorzec opisano po raz pierwszy w książce *Design Patterns: Elements of Reusable Object-Oriented Software* [Gamma, 1995].

Streszczenie

Wzorzec projektowy Builder umożliwia obiektom klienckim konstruowanie złożonych obiektów przez samo określanie ich typu i docelowej zawartości. Klient jest więc skutecznie izolowany od szczegółów związanych z procesem konstruowania obiektów.

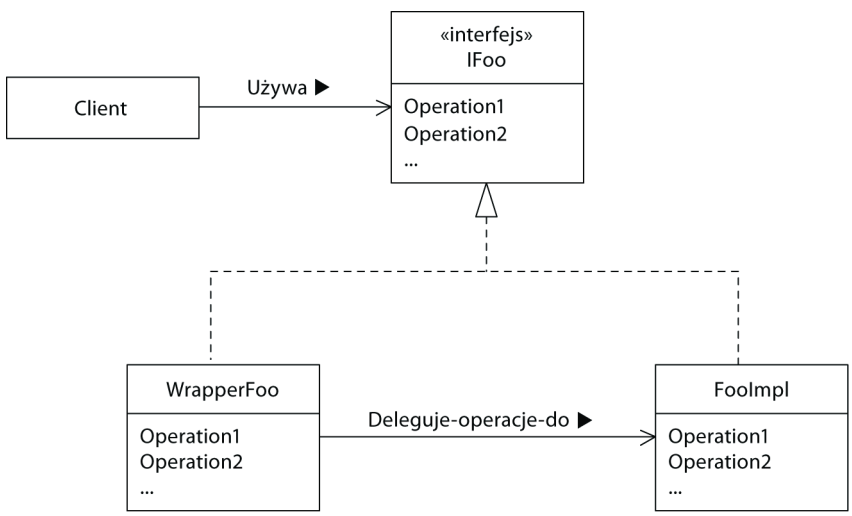
Kontekst

Kontekst wzorca projektowego Builder zilustrujemy przykładem narzędzia, które automatycznie generuje kod źródłowy klasy nazywanej *klasą opakowania* (ang. *wrapper class*).

Implementacje części spośród wzorców projektowych omawianych w tej książce, w tym wzorców Proxy i Façade, wymagały stosowania specjalnych klas opakowań. Cechą charakterystyczną klasy opakowania jest to, że implementuje interfejs przez delegowanie implementacji wszystkich funkcji, procedur i właściwości do innego obiektu, który implementuje ten sam interfejs.

Klasy opakowań zwykle definiują konstruktor, którego argument reprezentuje obiekt klasy implementujący interfejs implementowany przez samą klasę opakowania. Konstruktor ustawia wewnętrzną zmienną egzemplarza, która odwołuje się do przekazanego na wejściu obiektu. Funkcje i procedury tak skonstruowanego egzemplarza klasy opakowania przekazują swoje parametry do odpowiednich funkcji i procedur tego obiektu. Każda funkcja obiektu klasy opakowania zwraca wynik zwrócony jej przez wywołanie odpowiedniej funkcji tego wewnętrznego obiektu. Zwykle mówi się, że są to funkcje i procedury propagujące, ponieważ ich rolą jest wyłącznie przekazywanie parametrów i wyników odpowiednio od klasy klienckiej do wskazywanego obiektu i od wskazywanego obiektu do klasy klienckiej.

Obiekt klasy opakowania pełni funkcję pośrednika pomiędzy klientem a właściwym obiektem udostępniającym usługi lub dane wykorzystywane przez tego klienta (patrz rysunek 4.9). W przykładzie przedstawionym na rysunku 4.9 obiekt kliencki wykorzystuje obiekt klasy implementującej interfejs `IFoo`. Tym wykorzystywanym obiektem może być egzemplarz klasy `WrapperFoo`, która deleguje wszystkie otrzymywane wywołania do obiektu klasy `FooImpl`.



Rysunek 4.9. *Przykład klasy opakowania*

Klasa opakowania sama w sobie nie przedstawia zbyt dużej wartości. Egzemplarz klasy opakowania oferuje jedynie zachowanie obiektu, do którego się odwołuje, czyli tzw. obiektu opakowywanego. Oznacza to, że klasa opakowania jako taka nie wprowadza żadnych nowych rozwiązań. Mimo to klasy opakowania okazują się niezwykle przydatne podczas implementowania wzorca projektowego Decorator.

Wzorzec Decorator opisuje sposób tworzenia klasy implementującej ten sam interfejs co inna klasa, dzięki czemu może modyfikować *wybrane* elementy funkcjonalności i zachowań tej klasy. Implementacja wzorca projektowego Decorator może wymagać zbudowania klasy, która o tyle przypomina klasę opakowania, że większość jej funkcji i procedur przekazuje tylko odpowiednie wywołania do funkcji i procedur opakowywanego obiektu, jednak pozostałe realizują swoje zadania w sposób mniej lub bardziej odbiegający od rozwiązań zaimplementowanych w opakowywanym obiekcie. Większość pracy związanej z implementacją tego rodzaju klas może polegać na pisaniu funkcji i procedur propagujących, chyba że klasa dekoratora ma postać podklasy klasy opakowania.

Jeśli klasa dekoratora jest implementowana w formie podklasy klasy opakowania, dziedziczy wszystkie definiowane przez tę klasę opakowania niezbędne funkcje i procedury propagujące. Oznacza to, że programista implementujący klasę dekoratora nie musi tracić czasu na żmudne pisanie kolejnych metod propagujących.

Implementacja klasy dekoratora polegająca na rozszerzaniu klasy opakowania nie przyniesie programiście żadnych oszczędności czasowych, jeśli i tak będzie musiał napisać samą klasę opakowania. Istnieje na szczęście możliwość napisania narzędzia, które będzie automatycznie generowało klasy opakowań na podstawie obiektów klasy `System.Type` opisujących właściwe interfejsy.

Poniższy listing zawiera fragment kodu klasy pełniącej funkcję opakowania dla obiektów, które implementują interfejs `System.IAppDomainSetup`:

```
Imports System
Namespace Wrappers

    Public Class WrapperIAppDomainSetup
        Implements System.IAppDomainSetup

        ' Referencja do opakowywanego obiektu.
        Private myWrappedObject As IAppDomainSetup

        Private
        Sub New(ByVal myIAppDomainSetup As IAppDomainSetup)
            MyBase.New()
            myWrappedObject = myIAppDomainSetup
        End Sub

        ReadOnly Property WrappedObject() As IAppDomainSetup
            Get
                Return myWrappedObject
            End Get
        End Property

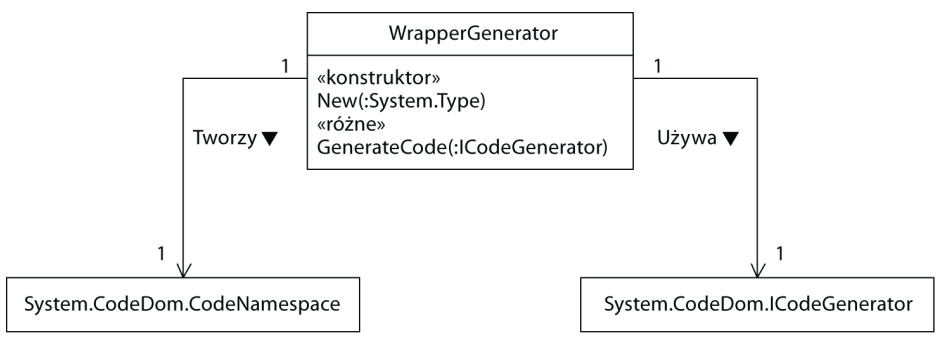
        Property ApplicationBase() As String _
                    Implements IAppDomainSetup.ApplicationBase
            Get
                Return myWrappedObject.ApplicationBase
            End Get
            Set(ByVal Value As String)
                myWrappedObject.ApplicationBase = Value
            End Set
        End Property
```

```
        Property ApplicationName() As String _
                    Implements IAppDomainSetup.ApplicationName
            Get
                Return myWrappedObject.ApplicationName
            End Get
            Set(ByVal Value As String)
                myWrappedObject.ApplicationName = Value
            End Set
        End Property

        ...
    End Class
End Namespace
```

Przypuśćmy, że projektujemy narzędzie generujące kod źródłowy klasy opakowania na podstawie przekazanego na wejściu obiektu klasy `Type`. Chcemy, aby nasze narzędzie generowało kod w języku VB.NET, C# lub innym języku dodanym do naszego oprogramowania w późniejszym terminie. Okazuje się, że zadanie budowy takiego narzędzia będzie znacznie prostsze, jeśli skorzystamy właśnie ze wzorca projektowego Builder.

Projekt narzędzia zorganizowano w taki sposób, aby w pierwszej kolejności były identyfikowane składowe przyszłej klasy opakowania. Następnie wykorzystujemy obiekt generatora kodu do wygenerowania samej klasy opakowania. Pozostałe składniki naszego narzędzia uzyskują dostęp do tego generatora za pośrednictwem odpowiedniego interfejsu. Dzięki temu narzędzie może używać różnych obiektów generatorów kodu i — tym samym — funkcjonować niezależnie od języka programowania, którego kod jest generowany. Większość klas potrzebnych do budowy takiego narzędzia jest dostępna w bibliotece platformy .NET Framework. Projekt naszego narzędzia przedstawiono na rysunku 4.10.



Rysunek 4.10. *Przykład współpracy klas implementujących wzorzec projektowy Builder*

Na diagramie przedstawionym na rysunku 4.10 widać między innymi klasę `WrapperGenerator`. Obiekt klasy `Type`, który opisuje odpowiedni interfejs, jest przekazywany do konstruktora wspomnianej klasy `WrapperGenerator`. W procesie konstruowania obiektu tej klasy konstruktor tworzy egzemplarz klasy `CodeNamespace`, który opisuje klasę opakowania i jej składowe. Po skonstruowaniu obiektu klasy `WrapperGenerator` możemy przekazać na wejściu metody `GenerateCode` tego obiektu egzemplarz interfejsu `ICodeGenerator`. Metoda `GenerateCode` wykorzysta ten egzemplarz do wygenerowania kodu źródłowego — okazuje się, że metoda nie musi nawet „wiedzieć", w którym języku programowania zostanie zapisany ten kod źródłowy.

Możemy uzyskać egzemplarz interfejsu `ICodeGenerator`, który będzie odpowiadał za generowanie kodu języka C#, wywołując funkcję `CreateGenerator` obiektu klasy `CSharpCodeProvider`. Egzemplarz interfejsu `ICodeGenerator`, który będzie odpowiadał za generowanie kodu języka VB.NET, możemy uzyskać, wywołując funkcję `CreateGenerator` obiektu klasy `VBCodeProvider`.

Zalety

- ☺ Program musi oferować możliwość generowania wielu zewnętrznych reprezentacji pewnych danych.
- ☺ Klasy odpowiedzialne za zapewnianie zawartości powinny być niezależne od wszelkich zewnętrznych reprezentacji danych oraz klas, które te dane budują. Jeśli klasy dostarczające zawartość nie są ograniczane przez żadne zależności od zewnętrznych reprezentacji danych, ewentualne zmiany klas tych zewnętrznych reprezentacji nie będą wymagały dostosowywania klas dostarczających zawartość.
- ☺ Klasy odpowiedzialne za budowę zewnętrznych reprezentacji danych są niezależne od klas dostarczających niezbędną zawartość, która jest podstawą procesu budowy tych reprezentacji. Egzemplarze tych klas mogą współpracować z dowolnymi obiektami dostarczającymi zawartość nawet wtedy, gdy nie dysponują żadną wiedzą o tych obiektach.

Rozwiązanie

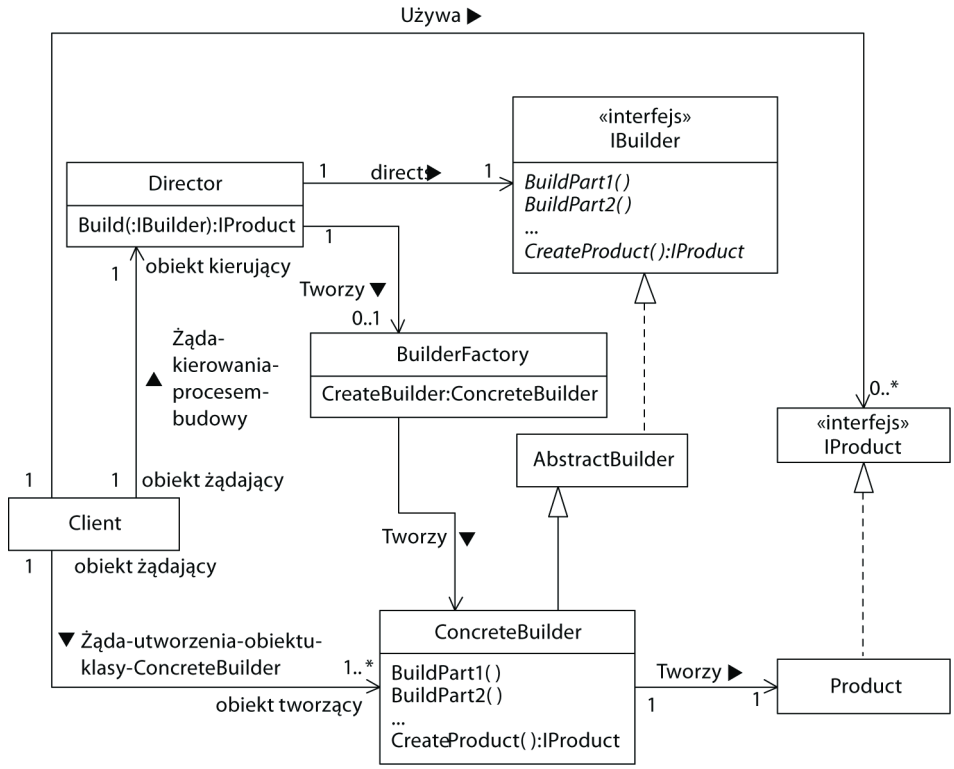
Na rysunku 4.11 przedstawiono diagram klas ilustrujący uczestników (klasy i interfejsy) wzorca projektowego Builder.

Role odgrywane przez te klasy i interfejsy we wzorcu projektowym Builder opisano poniżej:

`Product`. Klasa występująca w tej roli definiuje typ reprezentacji danych. Wszystkie klasy w tej roli powinny implementować interfejs `IProduct`, dzięki czemu obiekty zawierające referencje do egzemplarzy tych klas mogą korzystać z pośrednictwa tego interfejsu i — tym samym — unikać ścisłych zależności do konkretnych implementacji.

`IProduct`. Wzorzec projektowy Builder jest wykorzystywany do budowy rozmaitych rodzajów obiektów klasy `Product`, które są następnie wykorzystywane przez obiekty klasy `Client`. Aby zwolnić obiekty klienckie z obowiązku znajomości konkretnych klas budowanych dla nich obiektów `Product`, wszystkie te klasy powinny implementować interfejs `IProduct`. Obiekty klasy `Client` odwołują się do zbudowanych obiektów klasy `Product` właśnie za pośrednictwem interfejsu `IProduct`, dzięki czemu nie muszą dysponować wiedzą o konkretnej klasie budowanych obiektów.

`Client`. Egzemplarz klasy klienckiej inicjuje działania wzorca projektowego Builder. Właśnie te obiekty wywołują funkcję `CreateBuilder` klasy `BuilderFactory` i przekazują do metody `getInstance` informacje o typie



Rysunek 4.11. *Wzorzec projektowy Builder*

oczekiwanego produktu. Funkcja `CreateBuilder` identyfikuje właściwą klasę `ConcreteBuilder`, której obiekt należy utworzyć, i zwraca odpowiedni egzemplarz obiektowi klasy `Client`. Obiekt klasy `Client` przekazuje następnie uzyskany w ten sposób egzemplarz klasy `ConcreteBuilder` do funkcji `Build` obiektu klasy `Director`, który buduje odpowiedni obiekt produktu.

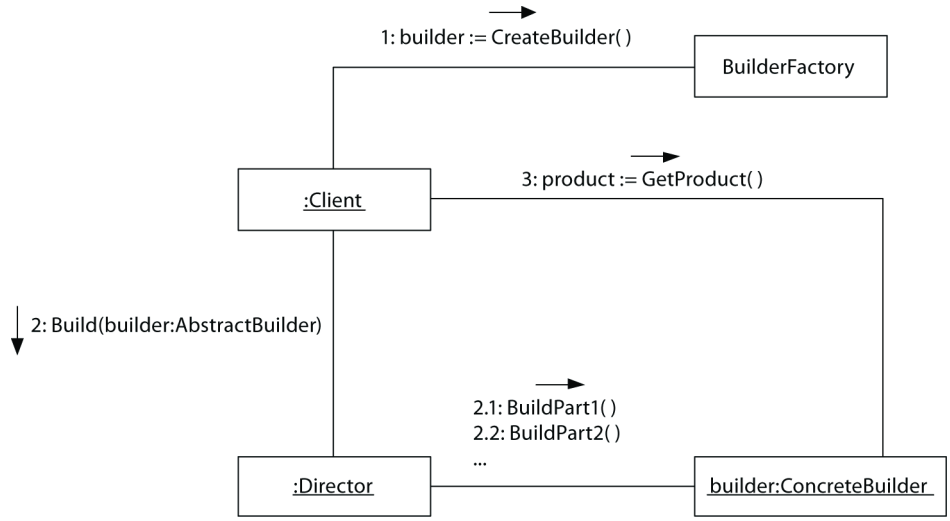
`ConcreteBuilder`. Klasa występująca w tej roli jest nie tylko konkretną podklasą abstrakcyjnej klasy bazowej `AbstractBuilder`, ale też implementacją interfejsu `IBuilder`. Za pomocą tej klasy można budować konkretne rodzaje reprezentacji danych dla obiektu klasy `Director`.

`AbstractBuilder`. Klasa występująca w tej roli jest abstrakcyjną klasą bazową dla klas `ConcreteBuilder`. Klasa `AbstractBuilder` oferuje wspólną logikę, która bardzo się przydaje podczas budowy logicznie spójnej implementacji klas `ConcreteBuilder`.

`IBuilder`. Obiekty klienckie uzyskują dostęp do konkretnych obiektów budowniczych właśnie za pośrednictwem interfejsu występującego w tej roli. Interfejsy `IBuilder` definiują funkcje i właściwości wykorzystywane przez obiekty klas `Client` i `Director` do komunikacji z obiektami klasy `ConcreteBuilder`.

`Director`. Obiekt klasy `Director` wywołuje funkcje i procedury obiektu klasy `ConcreteBuilder`, aby zapewnić dostęp do konkretnego budowniczego wraz z zawartością dla budowanego obiektu produktu.

Na rysunku 4.12 przedstawiono diagram współpracy, który pokazuje, jak egzemplarze opisanych powyżej klas powinny wspólnie funkcjonować.



Rysunek 4.12. *Przykład współpracy uczestników wzorca projektowego Builder*

Implementacja

Niektóre implementacje wzorca projektowego Builder nie budują obiektów, tylko — jak w przykładzie omówionym w podrozdziale „Kontekst" — generują np. jakieś pliki. W tego rodzaju zastosowaniach wzorca projektowego Builder nie używa się ani interfejsu `IProduct`, ani klas `Product`.

Proces budowy

Istnieją dwa najważniejsze rozwiązania w zakresie organizowania procesu budowy produktu. Jednym z nich jest budowa w pierwszej kolejności obiektów reprezentujących szczegółowe informacje o końcowym produkcie (z wyjątkiem danych właściwych dla konkretnego rodzaju produktu przeznaczonego do budowy). Przykładowo, jeśli naszym celem jest utworzenie określonego typu pliku zawierającego sformatowane dane tekstowe, możemy utworzyć sekwencję łańcuchów i innych obiektów opisujących docelowy sposób formatowania tych łańcuchów. Sekwencję tych obiektów można następnie przekazać do właściwego obiektu budowniczego, który wygeneruje — w zależności od potrzeb — plik w formacie RTF, PostScript lub dowolnym innym.

Jeśli budowany produkt jest stosunkowo prosty, początkowe tworzenie produktu pośredniego, półproduktu, który będzie niezależny od produktu końcowego, może rodzić więcej komplikacji niż korzyści. W takim przypadku prostszym rozwiązaniem jest przekazanie

opisu produktu przeznaczonego do budowy bezpośrednio do obiektu klasy `ConcreteBuilder`. Jeśli zdecydujemy się na takie rozwiązanie, największym wyzwaniem podczas implementacji wzorca projektowego Builder będzie zaprojektowanie zbioru funkcji i procedur definiowanych przez interfejs `IBuilder`, które będą zapewniały zawartość konkretnym obiektom budowniczych. Dobór tych funkcji i procedur jest o tyle trudny, że ich liczba może być bardzo duża. Zbiór składowych interfejsu `IBuilder` powinien być na tyle ogólny, aby możliwe było konstruowanie wszystkich rozważanych reprezentacji danych. Z drugiej strony, zbyt ogólny zbiór metod może utrudniać proces ich implementacji i używania. Znalezienie złotego środka pomiędzy uniwersalnością a utrudnieniami procesu implementacji rodzi następujące problemy, które należy rozwiązać w fazie implementowania programu:

- Każda funkcja opisująca zawartość, którą zadeklarowano w interfejsie `IBuilder`, może być dostarczana wraz z domyślną, pustą (pozbawioną konkretnych działań) implementacją w formie odpowiedniej klasy `AbstractBuilder`. Warto jednak pamiętać, że brak tego rodzaju implementacji w klasie `AbstractBuilder` umożliwia konkretnym klasom budowniczych dziedziczenie funkcji i procedur zadeklarowanych ze słowem kluczowym `MustOverride`, co z kolei powoduje, że kompilator będzie zmuszał programistów do implementowania tych wymaganych składowych. Wymuszanie na konkretnych klasach budowniczych zawierania implementacji funkcji i procedur jest dobrą praktyką zawsze wtedy, gdy tego rodzaju składowe zapewniają kluczowe informacje o zawartości. W ten sposób skutecznie zapobiegamy sytuacjom, w których programiści odpowiedzialni za implementację tych klas zapominają o niezbędnych funkcjach lub procedurach.
- Z drugiej strony, funkcja i procedury, które dostarczają opcjonalną zawartość lub dodatkowe informacje o strukturze tej zawartości, mogą się okazać niepotrzebne lub wręcz niepożądane w przypadku niektórych reprezentacji danych. Opracowanie domyślnych, pustych implementacji dla tego rodzaju funkcji i procedur pozwala oszczędzić mnóstwo czasu wymaganego do zaimplementowania konkretnych klas budowniczych, które tych funkcji i procedur po prostu nie potrzebują. Więcej informacji na temat tej technik znajdziesz w podrozdziale poświęconym wzorcowi projektowemu Null Object.
- W wielu przypadkach okazuje się, że organizacja konkretnych klas budowniczych w sposób umożliwiający proste dodawanie danych do obiektu produktu przez funkcje i procedury dostarczające zawartość w zupełności wystarczy. W pewnych sytuacjach nie ma prostego sposobu wskazywania budowniczym punktów ukończonych produktów, w których powinny się znajdować określone fragmenty tych produktów. Wówczas najprostszym rozwiązaniem jest takie zaprojektowanie funkcji dostarczającej zawartość, aby zwracała do egzemplarza klasy `Director` obiekt reprezentujący odpowiedni fragment produktu. Egzemplarz klasy `Director` może następnie przekazać ten obiekt na wejściu innej funkcji dostarczającej zawartość i wskazać pozycję odpowiedniego fragmentu w ramach całego produktu.

Skutki stosowania

☺ Procesy identyfikacji zawartości i konstruowania konkretnych reprezentacji danych są od siebie niezależne. Ewentualne zmiany reprezentacji danych produktu nie mają żadnego wpływu na obiekty dostarczające zawartość. Także obiekty budowniczych mogą korzystać z różnych obiektów dostarczających zawartość i nie wymagają dostosowywania do ewentualnych modyfikacji dokonywanych na tych obiektach.

- Wzorzec projektowy Builder daje nam większą kontrolę nad procesem konstruowania obiektów niż inne wzorce, np. Factory Method, ponieważ sterowanie wykonywaniem kolejnych operacji składających się na tworzenie obiektu produktu należy do egzemplarza klasy występującej w roli `Director`. W pozostałych wzorcach tworzenie całych obiektów ma postać jednej, niepodzielnej operacji.

Zastosowania w technologii .NET

Przykład omówiony w punkcie „Kontekst" pokazał, że sama platforma .NET Framework oferuje obsługę wzorca projektowego Builder. Odpowiednie mechanizmy są wykorzystywane między innymi przez środowisko programowania Visual Studio .NET.

Klasy `VBCodeProvider` i `CSharpCodeProvider` występują w roli `BuilderFactory` (fabryki budowniczych). Obie klasy definiują funkcje odpowiedzialne za tworzenie egzemplarzy klas implementujących interfejs `ICodeGenerator`. Sam interfejs `ICodeGenerator`, który we wzorcu projektowym Builder występuje w roli `IBuilder`, deklaruje funkcje odpowiedzialne za generowanie kodu źródłowego. Klasy implementujące interfejs `ICodeGenerator` występują w roli `ConcreteBuilder`.

Implementacja wzorca projektowego Builder obsługiwana za pośrednictwem wymienionych klas i interfejsu nie obejmuje żadnego interfejsu, któremu można by przypisać rolę `IProduct`, ani klas występujących w roli `Product`. Egzemplarze interfejsu `ICodeGenerator` nie tworzą obiektów reprezentujących kod źródłowy. Zamiast tego zapisują kod źródłowy za pośrednictwem obiektu klasy `TextWriter`, który przekazuje dane wyjściowe do pliku. Podobnie platforma .NET Framework nie zawiera klasy występującej w roli `Director`. W tym przypadku taka klasa jest niepotrzebna, ponieważ nazwa, atrybuty, składowe, wyrażenia składowych itd. są w pełni opisywane przez już istniejące obiekty, które przekazujemy do metod egzemplarzy interfejsu `ICodeGenerator`.

Przykład kodu źródłowego

Przeanalizujmy teraz prosty fragment kodu implementującego projekt opisany w punkcie „Kontekst" i przedstawiony na rysunku 4.10. Poniżej przedstawiono wybraną część klasy `WrapperGenerator`:

```
' Klasa WrapperGenerator odpowiada za generowanie kodu źródłowego
' klas opakowujących interfejsy lub inne klasy, które umożliwiają
' odpowiednie zakodowanie klasy pośrednika lub dekoratora.
```

```
Public Class WrapperGenerator
    ' Model generowanego kodu.
    Private codeModel As CodeNamespace

    ' Nazwa prywatnej zmiennej egzemplarza wygenerowanej klasy.
    Private Const VAR_NAME As String = "_myWrappedObject_"

    'Nazwa chronionej, dostępnej tylko do odczytu właściwości, która wchodzi
    'w skład wygenerowanej klasy i zapewnia dostęp do opakowanego obiektu.
    Private Const PROP_NAME As String = "_WrappedObject_"

    'Argumentem tego konstruktora jest typ (egzemplarz klasy Type)
    'klasy lub interfejsu, którego egzemplarz będzie opakowywany
    'przez wygenerowaną klasę.
    Public Sub New(ByVal base As Type)
        codeModel = GenerateCodeModel(base)
    End Sub
```

Konstruktor klasy `WrapperGenerator` wywołuje funkcję nazwaną `GenerateCodeModel`, która tworzy obiekty opisujące klasę opakowania dla interfejsu reprezentowanego przez argument tej funkcji. Odpowiednie obiekty są reprezentowane przez zmienną egzemplarza `codeModel` klasy `WrapperGenerator`.

Po skonstruowaniu obiektu klasy `WrapperGenerator` pozostałe obiekty mogą przekazywać egzemplarze interfejsu `ICodeGenerator` do procedury `GenerateCode` tego obiektu. Procedura `GenerateCode` przekazuje do otrzymanego egzemplarza interfejsu `ICodeGenerator` opis klasy opakowania. Egzemplarz interfejsu `ICodeGenerator` wykorzystuje opis klasy opakowania do wygenerowania odpowiedniego kodu źródłowego. Język kodu źródłowego klasy opakowania zależy wyłącznie od konkretnej klasy egzemplarza interfejsu `ICodeGenerator`, zatem obiekt klasy `WrapperGenerator` w ogóle nie musi dysponować tego rodzaju informacjami ani tym bardziej uzależniać sposobu pracy od poszczególnych języków programowania.

```
    Public Sub GenerateCode(ByVal generator As ICodeGenerator)
        Dim options As CodeGeneratorOptions = New CodeGeneratorOptions()
        generator.GenerateCodeFromNamespace(codeModel, Console.Out, options)
    End Sub
    ...
End Class
```

Poniżej przedstawiono kod źródłowy bardzo prostej klasy klienckiej wykorzystującej klasę `WrapperGenerator`:

```
Sub Main()
    Dim w As WrapperGenerator = New
WrapperGenerator(GetType(System.IAppDomainSetup))
    w.GenerateCode(New VBCodeProvider().CreateGenerator())
    w.GenerateCode(New CSharpCodeProvider().CreateGenerator())
End Sub
```

Wzorce pokrewne

Interface. Wzorzec projektowy Builder wykorzystuje wzorzec Interface do ukrywania konkretnych klas egzemplarzy interfejsu `IProduct`.

Composite. Obiekty klasy `Product`, które zbudowano z wykorzystaniem wzorca projektowego Builder, najczęściej mają postać kompozytów; zdarza się też, że obiekty opisujące produkty przekazywane do obiektów klasy `ConcreteBuilder` są kompozytami.

Factory Method. Wzorzec projektowy Builder wykorzystuje wzorzec Factory Method podczas decydowania o wyborze konkretnej klasy budowniczego dla tworzonego egzemplarza.

Template Method. Klasa abstrakcyjnego budowniczego często jest implementowana z wykorzystaniem wzorca projektowego Template Method.

Null Object. Implementacje wzorca projektowego Builder, które obejmują między innymi „puste" implementacje metod, muszą wykorzystywać wzorzec projektowy Null Object.

Visitor. W implementacjach wzorca Builder można stosować wzorzec projektowy Visitor. Wzorzec projektowy Visitor może też być rozwiązaniem w pełni alternatywnym względem wzorca Builder.

Strategy. Wzorzec projektowy Builder jest wyspecjalizowaną formą wzorca projektowego Strategy.

Prototype

Wzorzec opisano po raz pierwszy w książce *Design Patterns: Elements of Reusable Object-Oriented Software* [Gamma, 1995].

Streszczenie

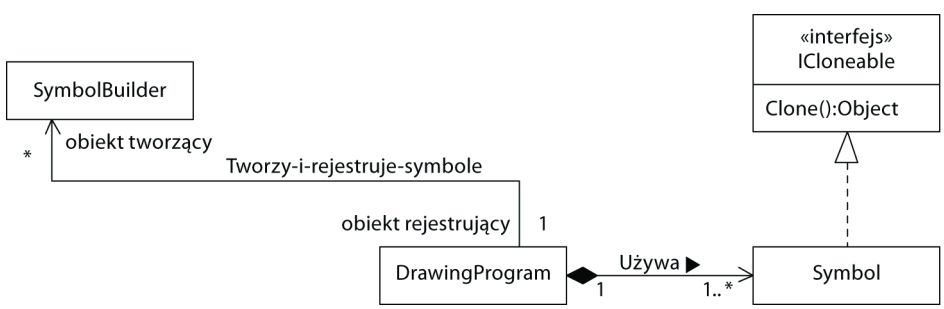
Wzorzec projektowy Prototype umożliwia obiektom tworzenie innych obiektów dostosowanych do bieżącego kontekstu bez znajomości ich konkretnej klasy lub bez wiedzy o szczegółach związanych z procesem ich konstruowania. Działanie tego wzorca polega na przekazywaniu odpowiednich obiektów prototypowych do egzemplarzy inicjujących procesy tworzenia właściwych obiektów. Obiekty odpowiedzialne za inicjowanie tych procesów tworzą właściwe obiekty, żądając od otrzymanych obiektów prototypowych sporządzania ich kopii.

Kontekst

Przypuśćmy, że projektujemy program CAD, który będzie oferował swoim użytkownikom możliwość rysowania diagramów złożonych z symboli dostępnych na specjalnej palecie. Program będzie zawierał wbudowany zbiór kluczowych symboli. Warto jednak pamiętać, że użytkownikami programu będą ludzie o różnych zainteresowaniach i specjalizujących się w odmiennych zadaniach projektowych. Podstawowy zbiór symboli nie będzie wystarczający dla tych bardziej zaawansowanych użytkowników. Każdy taki użytkownik będzie mógł skorzystać z mechanizmu dodawania symboli właściwych dla swojego obszaru zainteresowań. Większość użytkowników naszego programu będzie zainteresowana konkretnymi zastosowaniami i — tym samym — zbiorami specjalistycznych symboli, zatem implementacja mechanizmu dodawania tych zbiorów jest kluczowym wymaganiem stawianym tej aplikacji.

To wymaganie stawia nas przed problemem zapewnienia palet dodatkowych symboli. Można stosunkowo łatwo tak zorganizować składniki naszego programu, aby wszystkie symbole (zarówno wbudowane symbole kluczowe, jak i wyspecjalizowane symbole dodatkowe) dziedziczyły po wspólnej klasie macierzystej. Takie rozwiązanie spowoduje, że pozostałe mechanizmy naszego programu do rysowania diagramów będą mogły przetwarzać obiekty symboli w spójny sposób. Pozostaje jednak kwestia właściwego zaimplementowania mechanizmu tworzącego te obiekty. W takich przypadkach tworzenie obiektów jest zdecydowanie bardziej skomplikowane niż standardowe konstruowanie egzemplarzy pojedynczych klas. Cały ten proces może wymagać ustawiania wartości pewnych właściwości lub łączenia obiektów do postaci obiektów kompozytowych.

Dobrym rozwiązaniem jest dostarczenie do programu rysującemu utworzonych wcześniej obiektów, które będą wykorzystywane w roli prototypów podczas tworzenia podobnych obiektów już na potrzeby konkretnych operacji. Najważniejszym wymaganiem stawianym obiektom występującym w roli prototypów jest implementowanie przez ich klasy interfejsu `ICloneable`. Interfejs `ICloneable` deklaruje funkcję nazwaną `Clone`, której zadaniem jest zwracanie nowych obiektów będących wiernymi kopiami obiektów oryginalnych. Organizację naszego programu z użyciem obiektu prototypowego przedstawiono na rysunku 4.13.



Rysunek 4.13. *Przykład zastosowania wzorca projektowego Prototype*

Program rysujący utrzymuje kolekcję prototypowych obiektów klasy `Symbol`. Właściwe obiekty tej klasy są uzyskiwane przez klonowanie odpowiednich obiektów prototypowych. Obiekty klasy `SymbolBuilder` tworzą obiekty klasy `Symbol` i rejestrują je w naszym programie rysującym diagramy.

Zalety

☺ System musi oferować możliwość tworzenia obiektów nawet wtedy, gdy nie dysponuje precyzyjnymi informacjami o ich klasach, sposobach ich tworzenia lub reprezentowanymi przez nie danych.

☺ System „nie wie" z góry, dla których klas będą tworzone egzemplarze — odpowiednie informacje trafiają do systemu dopiero w czasie wykonywania programu, kiedy odpowiednie dane są pozyskiwane „w locie", np. z wykorzystaniem techniki dynamicznego łączenia.

☺ Poniższe strategie umożliwiają tworzenie rozmaitych obiektów w tym przypadku nie mają zastosowania lub z pewnych względów są niewskazane:

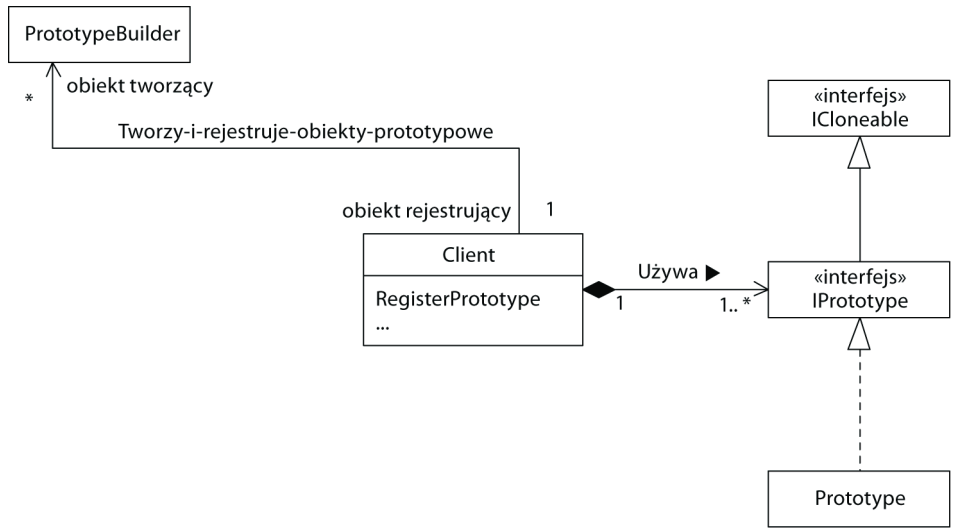
- Klasy inicjujące procesy tworzenia obiektów bezpośrednio konstruują odpowiednie egzemplarze. Oznacza to, że muszą dysponować wiedzą o klasach tych obiektów i — tym samym — są związane zależnościami od wielu innych obiektów.
- Klasy inicjujące procesy tworzenia obiektów konstruują odpowiednie egzemplarze za pośrednictwem klasy zawierającej metodę wytwórczą (metodę fabrykującą). Metoda wytwórcza zaprojektowana z myślą o tworzeniu wielu różnych obiektów może być bardzo duża i — tym samym — trudna w konserwacji.
- Klasy inicjujące procesy tworzenia obiektów konstruują odpowiednie egzemplarze za pośrednictwem abstrakcyjnej klasy fabryki. Aby abstrakcyjna fabryka mogła tworzyć wiele różnych typów obiektów, należy zaimplementować dla niej szereg klas konkretnych fabryk, które razem tworzą hierarchię reprezentującą klasy konstruowanych obiektów.

☺ Rozmaite obiekty tworzone przez system mogą być egzemplarzami jednej klasy, która zawiera różne informacje o stanach lub odmienną zawartość danych.

Rozwiązanie

Budowana klasa powinna mieć możliwość tworzenia obiektów klas implementujących znany interfejs na podstawie przekazywanych na wejściu prototypowych egzemplarzy (po jednym dla każdego konstruowanego obiektu). Samo tworzenie nowych obiektów polegałoby wówczas na klonowaniu tych prototypowych egzemplarzy.

Na rysunku 4.14 przedstawiono diagram ilustrujący organizację wzorca projektowego Prototype. Opis ról, w których występują poszczególne klasy i interfejsy widoczne na tym diagramie, znajdziesz poniżej.



Rysunek 4.14. *Wzorzec projektowy Prototype*

`Client`. Klasa kliencka reprezentuje na potrzeby wzorca projektowego Prototype pozostałe składniki danego programu. Klasy klienckie musi tworzyć obiekty, o których „wiedzą" bardzo niewiele. W związku z tym klasy klienckie muszą zawierać metody odpowiedzialne za dodawanie do wewnętrznych kolekcji kolejnych obiektów prototypowych. Na przedstawionym diagramie odpowiednią metodę oznaczono nazwą `RegisterPrototype`. W rzeczywistych implementacjach częściej stosuje się nazwy odpowiadające typom dodawanych obiektów prototypowych, np. `RegisterSymbol`.

`Prototype`. Klasy występujące w tej roli implementują interfejs `IPrototype`, a ich egzemplarze są tworzone wyłącznie celem ich późniejszego klonowania przez obiekty klienckie. Klasy `Prototype` zwykle mają postać klas abstrakcyjnych, dla których definiuje się wiele konkretnych podklas.

`IPrototype`. Klasy wszystkich obiektów prototypowych muszą implementować interfejs występujący w roli `IPrototype`. Klasy klienckie komunikują się z obiektami prototypowymi właśnie za pośrednictwem tego interfejsu. Interfejsy występujące w tej roli powinny rozszerzać interfejs `Cloneable` — dzięki temu egzemplarze wszystkich klas implementujących te interfejsy będzie można klonować.

`PrototypeBuilder`. Ta rola reprezentuje wszystkie klasy, których egzemplarze odpowiadają za przekazywanie obiektów prototypowych do obiektów klienckich. Tego rodzaju klasy należy oznaczać nazwami, które jednoznacznie wskazują na typy budowanych obiektów prototypowych, np. `SymbolBuilder`.

Obiekt klasy `PrototypeBuilder` tworzy obiekty klasy `Prototype`. Każdy taki nowo utworzony egzemplarz klasy Prototype jest przekazywany do funkcji `RegisterPrototype` obiektu klienckiego (egzemplarza klasy `Client`).

Implementacja

Największym problemem w zakresie implementacji opisanego systemu jest wybór właściwego sposobu dodawania przez egzemplarze klasy `PrototypeBuilder` nowych obiektów do palety obiektu klienckiego względem obiektów prototypowych. Najprostszym rozwiązaniem jest oczywiście udostępnienie odpowiedniej funkcji przez klasę kliencką — taka metoda mogłaby być wywoływana przez obiekty klasy `PrototypeBuilder`. Wadą tego podejścia jest wymagana znajomość konkretnej klasy obiektów klienckich po stronie obiektów klasy `PrototypeBuilder`. Jeśli takie wymaganie stanowi problem, można odizolować obiekty klasy `PrototypeBuilder` od wiedzy o poszczególnych klasach obiektów klienckich za pomocą dodatkowego interfejsu lub abstrakcyjnej klasy bazowej odpowiednio implementowanej lub dziedziczonej przez klasy klienckie.

Innym ważnym problemem jest znalezienie sposobu implementacji operacji klonowania obiektów prototypowych. Istnieją dwie podstawowe strategie implementowania tego rodzaju operacji:

- Płytkie kopiowanie oznacza, że zmienne sklonowanego obiektu zawierają te same wartości co zmienne obiektu oryginalnego i że wszystkie reprezentowane przez te zmienne referencje do obiektów wskazują na te same obiekty w pamięci. Innymi słowy, płytka kopia jest kopią samego klonowanego obiektu, a więc bez obiektów, do których się odwołuje. Zarówno oryginał, jak i jego płytka kopia odwołują się do tych samych obiektów.
- Głębokie kopiowanie oznacza, że zmienne sklonowanego obiektu zawierają te same wartości co zmienne obiektu oryginalnego — wyjątkiem są zmienne odwołujące się do innych obiektów, ponieważ w sklonowanym obiekcie odwołują się do kopii obiektów wskazywanych przez zmienne obiektu oryginalnego. Innymi słowy, głęboka kopia jest nie tylko kopią samego sklonowanego obiektu, ale też wszystkich obiektów, do których ten obiekt się odwołuje. Głęboka kopia odwołuje się do kopii obiektów wskazywanych przez obiekt oryginalny.

  Proces implementacji mechanizmu głębokiego kopiowania jest dużo bardziej skomplikowany niż w przypadku płytkiego kopiowania. Należy zdecydować, czy dla pośrednio klonowanych obiektów (wskazywanych przez obiekt oryginalny) mają być tworzone kopie płytkie czy kopie głębokie. Mechanizm musi sobie radzić także z sytuacjami, w których referencje mają charakter cykliczny.

Płytkie kopiowanie jest łatwiejsze w implementacji, ponieważ wszystkie klasy dziedziczą po klasie `Object` chronioną funkcję `MemberwiseClone`, która zwraca ich płytkie kopie. Implementacja funkcji `Clone` interfejsu `ICloneable` może się więc sprowadzać do wywołania funkcji `MemberwiseClone` klasy implementującej.

Niektóre rodzaje obiektów, np. wątki czy gniazda, nie mogą być po prostu kopiowane lub współdzielone przez wiele obiektów klienckich. Niezależnie od stosowanej strategii kopiowania ewentualne zawieranie referencji do tego typu obiektów w oryginalnym egzemplarzu będzie się wiązało z koniecznością skonstruowania ich odpowiedników w egzemplarzu sklonowanym.

Jeśli paleta obiektów prototypowych obiektu klasy `Client` nie składa się ze stałej liczby obiektów charakteryzujących się jednym, niezmiennym zakresem działań, stosowanie osobnych zmiennych do reprezentowania każdego z tych prototypów byłoby dość kłopotliwe. W takim przypadku dużo lepszym rozwiązaniem będzie użycie obiektu kolekcji, który może zawierać dynamicznie zmieniającą się liczbę prototypowych obiektów wchodzących w skład palety. Tego rodzaju obiekty kolekcji stosowane w ramach wzorca projektowego Prototype bywają nazywane *menedżerami prototypów* (ang. *prototype managers*). Obiekty występujące w tej roli nie muszą oczywiście mieć postaci prostych kolekcji — można sobie wyobrazić menedżera prototypów, który umożliwia odnajdywanie składowanych obiektów według wartości atrybutów lub innych kluczy.

Jeśli nasz program będzie współpracował z wieloma obiektami klienckimi, będziemy musieli się zmierzyć z jeszcze jednym istotnym problemem. Obiekty klienckie mogą albo dysponować własnymi paletami prototypowych obiektów, albo współdzielić jedną paletę. Decyzja w tym zakresie powinna zależeć od wymagań stawianych budowanej aplikacji.

Skutki stosowania

- ☺ Program może dynamicznie dodawać i usuwać prototypowe obiekty w czasie wykonywania. Ta niewątpliwa zaleta jest cechą wyróżniającą wzorzec projektowy Prototype wśród wszystkich prezentowanych w tej książce wzorców konstrukcyjnych.
- ☺ Obiekt klasy `PrototypeBuilder` może zawierać stały, niezmienny zbiór obiektów prototypowych.
- ☺ Obiekt klasy `PrototypeBuilder` może oferować dodatkową elastyczność polegającą na możliwości tworzenia nowych obiektów prototypowych przez łączenie innych obiektów oraz modyfikowania wartości reprezentowanych przez właściwości tych obiektów.
- ☺ Obiekt kliencki może dodatkowo oferować możliwość tworzenia nowych rodzajów obiektów prototypowych. Przykładowo, w przypadku programu wspomagającego rysowanie diagramów, który omówiono w tym podrozdziale, obiekt klienta mógłby np. umożliwiać użytkownikom identyfikację rysunków wewnętrznych reprezentowanych na bardziej ogólnych diagramach przez odpowiedni symbol.
- ☺ Klasa kliencka jest niezależna od konkretnej klasy wykorzystywanych przez siebie obiektów prototypowych. Oznacza to, że klasa kliencka nie musi dysponować szczegółowymi informacjami o procesie konstruowania tych obiektów.
- ☺ Wszelkie szczegóły związane z konstruowaniem obiektów prototypowych są reprezentowane przez obiekty klasy `PrototypeBuilder`.
- ☺ Wymaganie dotyczące implementowania przez klasy obiektów prototypowych implementowanie takiego interfejsu jak `IPrototype` powoduje, że wzorzec projektowy Prototype gwarantuje zgodność zbiorów funkcji i procedur udostępnianych przez te obiekty prototypowe obiektom klas klienckich.

- Obiekty prototypowe nie muszą być organizowane w ramach żadnych specjalnych hierarchii klas.

☹ Wadą wzorca projektowego Prototype są dodatkowe wymagania czasowe związane z koniecznością pisania klas `PrototypeBuilder`.

☹ Programy wykorzystujące wzorzec projektowy Prototype bazują na mechanizmie dynamicznego łączenia lub innym, podobnym rozwiązaniu. Instalacja programów wykorzystujących tego rodzaju mechanizmy zwykle jest procesem dość skomplikowanym, którego realizacja wymaga szczegółowej wiedzy o środowisku docelowym (zupełnie zbędnej podczas wdrażania tradycyjnego oprogramowania).

Zastosowania w technologii .NET

Jednym z przykładów obsługi wzorca projektowego Prototype w technologii .NET jest połączenie architektury komponentowej z mechanizmem serializacji[2]. Istnieje możliwość budowy aplikacji, która będzie konfigurowała i inicjalizowała swoje komponenty, by następnie serializować je w plikach dyskowych. Kiedy takie komponenty zostaną już umieszczone w odpowiednich plikach, mogą być wielokrotnie wykorzystywane przez inne aplikacje, które automatycznie są zwolnione z obowiązku dysponowania informacji o przebiegu oryginalnego procesu tworzenia tych komponentów.

Przykład kodu źródłowego

Przypuśćmy, że piszemy interaktywną grę z gatunku RPG, czyli grę, której uczestnik może stosować rozmaite zachowania interaktywne względem symulowanych postaci. Okazuje się, że użytkownicy tego programu skarżą się na znużenie wielokrotnie powtarzanymi interakcjami dotyczącymi wciąż tych samych postaci i wykazują zainteresowanie ewentualnymi nowymi postaciami. W odpowiedzi na oczekiwania graczy próbujemy opracować dodatek do gry, który będzie się składał nie tylko z kilku z góry wygenerowanych postaci, ale też podprogramu automatycznie generującego dodatkowe postacie.

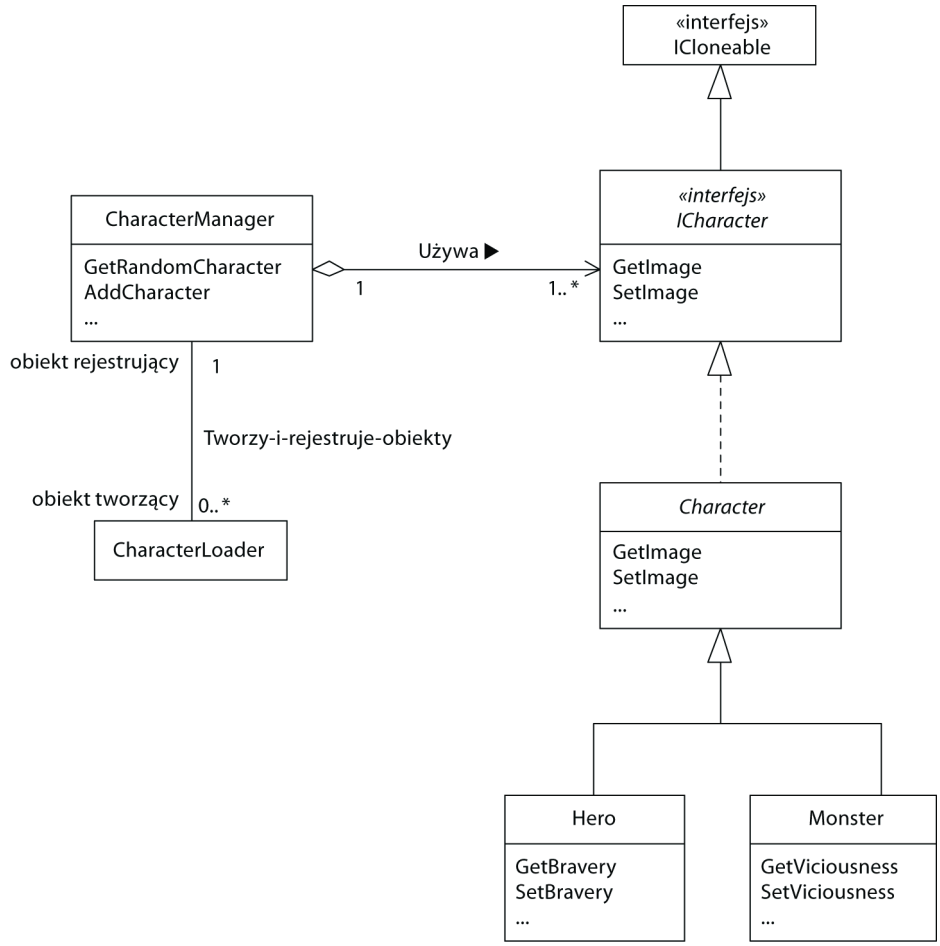
Postacie występujące w grze są reprezentowane przez egzemplarze stosunkowo niewielkiej liczby klas: Hero, Fool, Villain oraz Monster. Tym, co odróżnia od siebie egzemplarze tej samej klasy, są odmienne wartości atrybutów przypisanych reprezentowanym postaciom z danej kategorii, w tym reguły stosowane podczas wizualizacji postaci, wzrost, waga, inteligencja oraz sprawność.

Kilka spośród klas składających się na naszą grę przedstawiono na rysunku 4.15.

Poniżej przedstawiono kod źródłowy tego przykładu:

```
Imports System
Imports System.Drawing
```

[2] Serializacja umożliwia zapisywanie obiektów w formie strumieni bajtów, na podstawie których można tworzyć kolejne obiekty.



Rysunek 4.15. *Przykład użycia wzorca projektowego Prototype*

```
Imports System.Collections
Imports System.IO
Imports System.Runtime.Serialization.Formatters.Binary

' Wszystkie klasy, których egzemplarze mogą występować w grze w roli postaci,
' muszą implementować ten interfejs.
Public Interface ICharacter
    Inherits ICloneable
    Property Name() As String
    Property Image() As Image
    Property Strength() As Integer
    ...
End Interface 'ICharacter

' Klasa abstrakcyjna wykorzystywana do implementowania postaci występujących w grze.
Public MustInherit Class Character
    Implements ICharacter
    Private myName As String
```

```
    Private myImage As Image
    Private myStrength As Integer

    ' Nazwa postaci
    Public Property Name() As String Implements ICharacter.Name
        Get
            Return myName
        End Get
        Set(ByVal Value As String)
            myName = Value
        End Set
    End Property

    ' Obraz postaci
    Public Property Image() As Image Implements ICharacter.Image
        Get
            Return myImage
        End Get
        Set(ByVal Value As Image)
            myImage = Value
        End Set
    End Property

    ' Siła fizyczna postaci
    Public Property Strength() As Integer _
                    Implements ICharacter.Strength
        Get
            Return myStrength
        End Get
        Set(ByVal Value As Integer)
            myStrength = Value
        End Set
    End Property

    ' Zwraca kopię tego obiektu postaci.
    Public Function Clone() As Object _
                    Implements ICloneable.Clone
        Return Me.MemberwiseClone()
    End Function

    ...
End Class 'Character

' Klasa reprezentująca postacie należące do kategorii bohaterów.
Public Class Hero
    Inherits Character
    Private myBravery As Integer

    Public Property Bravery() As Integer
        Get
            Return myBravery
        End Get
        Set(ByVal Value As Integer)
            myBravery = Value
        End Set
    End Property

End Class 'Hero
```

```
' Egzemplarz tej klasy utrzymuje wewnętrzną kolekcję prototypowych
' obiektów postaci — nowe obiekty postaci są tworzone przez kopiowanie
' odpowiednich obiektów prototypowych.
Public Class CharacterManager
    Private characters As New ArrayList()
    Private Shared rand As New Random()

    '
    ' Zwraca kopię losowego obiektu postaci składowanego w wewnętrznej kolekcji.
    '
    ReadOnly Property RandomCharacter() As Character
        Get
            Dim i As Integer = rand.Next(characters.Count)
            Dim c As Character = CType(characters(i), Character)
            Return CType(c.Clone(), Character)
        End Get
    End Property

    '
    ' Dodaje do kolekcji nowy obiekt prototypowy.
    '
    Public Sub AddCharacter(ByVal character As Character)
        characters.Add(character)
    End Sub

End Class 'CharacterManager

' Klasa wczytuje obiekty postaci i dodaje je do menedżera postaci
' (obiektu klasy CharacterManager).
'
Class CharacterLoader
    Private mgr As CharacterManager

    ' cm - obiekt klasy CharacterManager, z którym współpracuje obiekt wczytujący.
    Sub New(ByVal cm As CharacterManager)
        mgr = cm
    End Sub

    ' Wczytuje obiekty postaci ze wskazanego pliku. Ponieważ ewentualny
    ' błąd będzie miał wpływ tylko na te części programu, które
    'w  założeniu miały korzystać z tego nowego obiektu postaci,
    ' generowanie wyjątku jest w tym przypadku niepotrzebne.
    Function loadCharacters(ByVal fname As String) As Integer
        Dim formatter As New BinaryFormatter()
        Dim reader As Stream = File.OpenRead(fname)
        Dim objectCount As Integer = 0 'Liczba wczytanych obiektów
        'Jeśli proces konstruowania obiektu klasy InputStream zakończy się niepowodzeniem, funkcja
        ' zwraca sterowanie.
        Try
            While True
                Dim o As Object = formatter.Deserialize(reader)
                If TypeOf o Is Character Then
                    mgr.AddCharacter(CType(o, Character))
                    objectCount += 1
                End If
            End While
        Finally
```

```
            If reader IsNot Nothing Then
                reader.Close()
            End If
        End Try
        Return objectCount
    End Function 'loadCharacters

End Class 'CharacterLoader
```

Wzorce pokrewne

Composite. Wzorzec projektowy Prototype często jest stosowany łącznie z wzorcem Composite. Wzorzec projektowy Composite ułatwia organizowanie obiektów prototypowych.

Abstract Factory. Wzorzec projektowy Abstract Factory stanowi kuszącą alternatywę dla wzorca projektowego Prototype w sytuacji, gdy nie są potrzebne dynamiczne zmiany palety prototypowych obiektów (tego rodzaju przypadki są obsługiwane przez wzorzec Prototype).

Klasa `PrototypeBuilder` wykorzystuje wzorzec projektowy Abstract Factory do tworzenia zbioru obiektów prototypowych.

Façade. Klasa kliencka często pełni funkcję fasady oddzielającej pozostałe klasy wchodzące w skład implementacji wzorca projektowego Prototype od reszty programu.

Factory Method. Wzorzec projektowy Factory Method może być alternatywą dla wzorca projektowego Prototype w sytuacji, gdy paleta prototypowych obiektów nigdy nie zawiera więcej niż jeden taki obiekt.

Decorator. Wzorzec projektowy Prototype często jest wykorzystywany łącznie ze wzorcem Decorator, który ułatwia konstruowanie obiektów prototypowych.

Singleton

Wzorzec opisano po raz pierwszy w książce *Design Patterns: Elements of Reusable Object-Oriented Software* [Gamma, 1995].

Streszczenie

Wzorzec projektowy Singleton gwarantuje nam, że zostanie utworzony tylko jeden egzemplarz danej klasy. Wszystkie obiekty korzystające z danej klasy używają tego samego egzemplarza.

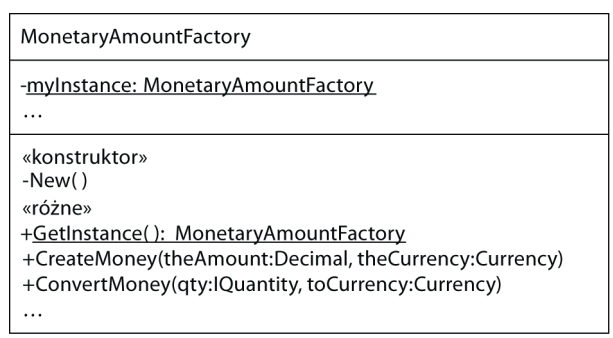
Kontekst

Przypuśćmy, że projektujemy część szkieletu systemów handlu elektronicznego odpowiedzialną za zarządzanie kwotami pieniężnymi. Każda taka kwota składa się z liczby i waluty. Zdecydowaliśmy, że użyjemy klasy, która będzie odpowiadała przede wszystkim za tworzenie obiektów reprezentujących kwoty pieniężne, ale także za konwersje tych kwot pomiędzy różnymi obsługiwanymi walutami. Uznaliśmy, że najlepszą nazwą dla tej klasy będzie `MonetaryAmountFactory`.

Początkowa implementacja klasy `MonetaryAmountFactory` będzie zawierała niewielki zbiór trwale zakodowanych walut wraz ze stałymi relacjami cenowymi. Ponieważ jednak w przyszłości klasa będzie musiała obsługiwać kolejne waluty oraz sytuacje, w których poszczególne kraje zmieniają lub denominują swoje waluty, przewidujemy, że ostateczna wersja tej klasy powinna pobierać niezbędne dane z odpowiedniej tabeli bazy danych.

Przewidywania, zgodnie z którymi klasa `MonetaryAmountFactory` będzie w przyszłości zarządzała danymi zewnętrznymi, rodzi pewne poważne problemy już na etapie jej projektowania. Chcemy być pewni, że wszystkie obiekty klienckie względem tej klasy będą stosowały ten sam zbiór walut. Z uwagi na oszczędność pamięci chcemy też uniknąć sytuacji, w której tabela walut będzie składowana w pamięci w formie wielu kopii.

Na rysunku 4.16 przedstawiono projektowaną strukturę klasy `MonetaryAmountFactory`. Konstruktor klasy `MonetaryAmountFactory` jest składową prywatną. W ten sposób eliminujemy możliwość bezpośredniego tworzenia egzemplarzy tej klasy z poziomu innych klas. Zamiast publicznego konstruktora klasa `MonetaryAmountFactory` udostępnia funkcję `GetInstance`, za pośrednictwem której pozostałe obiekty mogą uzyskiwać egzemplarz tej klasy. Funkcja `GetInstance` zawsze zwraca ten sam egzemplarz klasy `MonetaryAmountFactory` reprezentowany przez prywatną zmienną dzieloną `myInstance`.



Rysunek 4.16. *Fabryka obiektów reprezentujących kwoty pieniężne*

Pozostałe funkcje klasy `MonetaryAmountFactory` odpowiadają za tworzenie kwot pieniężnych i konwersje kwot już istniejących pomiędzy obsługiwanymi walutami. Ponieważ zawsze istnieje tylko jeden egzemplarz klasy `MonetaryAmountFactory`, wszystkie jej obiekty klienckie muszą korzystać z tego samego egzemplarza. Oznacza to, że program potrzebuje dokładnie jednej kopii tabeli reprezentującej relacje cenowe pomiędzy obsługiwanymi walutami.

Zalety

☺ Musi istnieć przynajmniej jeden egzemplarz klasy singletonu. Nawet jeśli funkcje naszej klasy nie potrzebują do właściwego działania żadnych danych egzemplarza lub jeśli korzystają wyłącznie z danych współdzielonych, egzemplarz takiej klasy może być niezbędny z kilku ważnych powodów. Jednym z nich jest możliwy dostęp do danej klasy za pośrednictwem interfejsu.

☺ Nigdy nie może istnieć więcej niż jeden egzemplarz klasy singletonu. Klasy singletonów stosuje się między innymi w sytuacjach, gdy chcemy dysponować dokładnie jednym źródłem pewnych informacji. Przykładowo, możemy zdecydować, że za generowanie sekwencji numerów seryjnych powinien odpowiadać pojedynczy obiekt właściwej klasy.

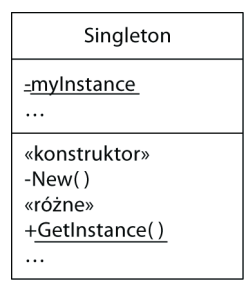
☺ Jedyny egzemplarz klasy singletonu musi być dostępny z poziomu wszystkich obiektów klienckich względem tej klasy.

☹ Nawet jeśli koszty operacji utworzenia pojedynczego obiektu są relatywnie niskie, nie można wykluczyć, że w przyszłości taki obiekt będzie zajmował znaczne ilości pamięci operacyjnej lub stale (przez cały czas życia) korzystał z innych, równie cennych zasobów.

Rozwiązanie

Wzorzec projektowy Singleton jest relatywnie prosty, ponieważ obejmuje tylko jedną klasę. Przykładową strukturę takiej klasy przedstawiono na rysunku 4.17.

Rysunek 4.17. *Singleton*



Klasa singletonu zawiera zmienną dzieloną, która reprezentuje jedyny egzemplarz tej klasy. Egzemplarz klasy singletonu jest tworzony w momencie wczytywania klasy do pamięci. Należy implementować tego rodzaju klasy w sposób wykluczający możliwość tworzenia dodatkowych egzemplarzy przez pozostałe klasy. Oznacza to, że wszystkie konstruktory klasy singletonu muszą być prywatne.

Aby uzyskać dostęp do jedynego egzemplarza klasy singletonu, należy używać odpowiedniej metody dzielonej (zwykle nazywanej `GetInstance` lub `GetClassname`), która zwraca referencję do tego egzemplarza.

Implementacja

Chociaż wzorzec projektowy Singleton nie opisuje żadnych szczególnie skomplikowanych rozwiązań, podczas jego implementowania warto pamiętać o kilku pozornie nieważnych, ale w praktyce dość istotnych zagadnieniach:

Prywatny konstruktor. Wymuszanie specyficznej natury klasy singletonu wymaga takiego jej zakodowania, które uniemożliwi pozostałym klasom bezpośrednie tworzenie egzemplarzy. Najlepszym sposobem realizacji tego wymagania jest zadeklarowanie konstruktora klasy singletonu w formie składowej prywatnej. Należy zachować ostrożność i zadeklarować przynajmniej jeden taki konstruktor — jeśli klasa nie będzie zawierała żadnego konstruktora, kompilator automatycznie wygeneruje domyślny konstruktor publiczny.

„Leniwe" tworzenie egzemplarzy. Typowym zastosowaniem wzorca projektowego singletonu są sytuacje, w których jedyne egzemplarze klas singletonów mogą się okazać niepotrzebne. Nie wiemy, czy dany program faktycznie będzie korzystał z egzemplarza singletonu do momentu pierwszego wywołania funkcji `GetInstance`. W takich przypadkach dobrą praktyką jest odkładanie procesu tworzenia jedynego egzemplarza singletonu właśnie do chwili, w której nastąpi takie wywołanie.

Możliwe stosowanie więcej niż jednego egzemplarza. Inna odmiana wzorca projektowego Singleton ma swoje źródło w fakcie zawierania polityki, strategii tworzenia jedynego egzemplarza klasy w kodzie samej klasy singletonu. Ponieważ politykę tworzenia egzemplarza należy zaimplementować w ciele funkcji `GetInstance` klasy singletonu, można tę politykę stosunkowo łatwo dostosowywać do zmieniającej się sytuacji. Przykładowo, taka polityka może przewidywać zwracanie przez funkcję `GetInstance` jednego z dwóch potencjalnych egzemplarzy lub tworzenia nowych egzemplarzy w stałych odstępach czasu.

Tego typu rozwiązania mogą być szczególnie korzystne w sytuacji, gdy używamy niewielkiej stałej liczby pojedynczych obiektów reprezentujących np. zewnętrzne zasoby i chcemy — na tyle, na ile jest to możliwe — zrównoważyć ich obciążenie. Jeśli rozciągniemy tę strategię na dowolną lub zmienną liczbę obiektów, otrzymamy wzorzec projektowy Object Pool (puli obiektów).

Tworzenie kopii singletonu. Obiekt klasy singletonu zwykle ma być jedynym egzemplarzem swojej klasy. Nawet w sytuacji, gdy dopuścimy możliwość stosowania więcej niż jednego egzemplarza klasy singletonu, będziemy dążyli do zachowania pełnej kontrolki nad procesem tworzenia tych egzemplarzy w ramach funkcji `GetInstance` klasy singletonu. Oznacza to, że pozostałe klasy nadal nie będą mogły tworzyć kopii obiektów singletonów.

Klasa singletonu z oczywistych względów nie powinna implementować interfejsu `System.ICloneable`; nie powinna też udostępniać swoim klientom żadnych innych mechanizmów umożliwiających kopiowanie jej egzemplarzy.

Serializacja jest mechanizmem obsługującym konwersję zawartości obiektów składowanych w pamięci na strumienie bajtów lub kod języka XML (od ang. *eXtensible Markup Language*). Deserializacja jest mechanizmem odwrotnym, czyli rozwiązaniem umożliwiającym konwersję strumienia bajtów lub kodu języka XML (utworzonego wcześniej w procesie serializacji) na obiekty, których zawartość jest identyczna jak w przypadku serializowanych obiektów oryginalnych.

Serializacja może być wykorzystywana do kopiowania obiektów, chociaż to nie kopiowanie jest podstawowym celem tego mechanizmu. Serializację najczęściej stosuje się do realizacji dwóch zadań:

- Serializacja jest wykorzystywana do utrwalania obiektów przez zapisywanie ich zawartości w plikach dyskowych (oczywiście po uprzednim przekonwertowaniu na strumień bajtów). Dzięki temu mamy możliwość wiernego odtwarzania zawartości utrwalonych w ten sposób obiektów.
- Serializacja jest wykorzystywana do obsługi zdalnych wywołań procedur za pośrednictwem protokołu SOAP (od ang. *Simple Object Access Protocol*). Protokół SOAP wykorzystuje mechanizm serializacji do przekazywania wartości argumentów na wejściu zdalnych procedur oraz do przekazywania zwracanych wartości do zdalnych obiektów wywołujących.

Ogólnie, żadne z tych rozwiązań nie znajduje zastosowania w przypadku obiektów klas singletonów. Obiekty tych klas z reguły nie powinny wchodzić w skład serializowanych strumieni obiektów. Można wykluczyć taką możliwość, rezygnując z bezpośredniego serializowania obiektów singletonów oraz nie zapisując referencji do obiektów singletonów w zmiennych egzemplarzy klas klienckich. Zamiast tego klasy klienckie powinny wywoływać metody `GetInstance` wykorzystywanych klas singletonów za każdym razem, gdy potrzebują dostępu do obiektów singletonów.

Obsługa jednoczesnych wywołań funkcji GetInstance

Jeśli istnieje nawet najmniejsze ryzyko jednoczesnego wywołania metody `GetInstance` klasy singletonu przez wiele wątków, musimy zagwarantować, że w odpowiedzi na te wywołania nasza metoda nie utworzy wielu egzemplarzy klasy singletonu. Przeanalizujmy następujący fragment kodu:

```
Public Class Foo
    Private Shared myInstance As Foo
    ...
    Public Shared Function GetInstance() As Foo
        If myInstance Is Nothing Then
            myInstance = New Foo()
        End If
        Return myInstance
    End Function
    ...
End Class
```

Jeśli dwa wątki jednocześnie wywołają funkcję `GetInstance` i jeśli będą to dwa pierwsze wywołania tej funkcji (jeśli nie będzie istniał utworzony wcześniej egzemplarz klasy singletonu), instrukcje warunkowe wykonane w odpowiedzi na oba wywołania wykażą,

że zmienna `myInstance` ma wartość `Nothing`, zatem w obu przypadkach zostanie utworzony nowy egzemplarz klasy `Foo`. Można tego problemu uniknąć, stosując wyrażenie `SyncLock`, które wykluczy możliwość jednoczesnego sprawdzania wartości zmiennej `myInstance` przez więcej niż jeden wątek. Dzięki temu możemy być pewni, że niezależnie od liczby jednoczesnych wywołań zostanie utworzony tylko jeden egzemplarz klasy `Foo`.

Jeśli zdecydujesz się na użycie wyrażenia `SyncLock` do ograniczenia liczby egzemplarzy klasy singletonu, zwykle powinieneś zastosować blokadę na klasie, która zawiera funkcję `GetInstance`. Poniższy kod ilustruje możliwy sposób przebudowy poprzedniego przykładu — tym razem użyto wyrażenia `SyncLock` blokującego dostęp do klasy `Foo`:

```
Public Class Foo
    Private Shared myInstance As Foo
    ...
    Public Shared Function GetInstance() As Foo
        SyncLock GetType(Foo)
            If myInstance Is Nothing Then
                myInstance = New Foo()
            End If
        End SyncLock
        Return myInstance
    End Function
    ...
End Class
```

Warto pamiętać, że stosowanie wyrażenia `SyncLock` może powodować pewne opóźnienia związane z koniecznością uzyskania i założenia blokady przed przystąpieniem do dalszego przetwarzania.

Skutki stosowania

☺ Nie może istnieć więcej niż jeden egzemplarz klasy singletonu.

☺ Metoda `GetInstance` klasy singletonu zawiera implementację strategii tworzenia jedynego egzemplarza tej klasy. Klasy wykorzystujące klasę singletonu (klasy klienckie względem klasy singletonu) nie są w żaden sposób uzależnione od szczegółowych rozwiązań zastosowanych w metodzie tworzącej egzemplarz singletonu.

- Pozostałe klasy, które zawierają odwołania do jedynego egzemplarza klasy singletonu, muszą uzyskiwać ten egzemplarz za pośrednictwem funkcji `GetInstance`, ponieważ nie mogą tego egzemplarza konstruować samodzielnie przez wywołanie konstruktora.

☹ Pomysł tworzenia podklas klas singletonów jest dość niefortunny, ponieważ musiałby skutkować naruszeniem izolacji oddzielającej politykę tworzenia pojedynczych egzemplarzy tego rodzaju klas od klas klienckich. Tworzenie podklas klasy singletonu jest możliwe tylko w sytuacji, gdy klasa singletonu udostępnia konstruktor, który nie jest prywatny. Warto też pamiętać, że z uwagi na brak możliwości przykrywania dzielonych funkcji podklasy klasy singletonu musiałyby udostępniać funkcję `GetInstance` swojej klasy bazowej.

Zastosowania w technologii .NET

Przykładem singletonu (implementacji wzorca projektowego Singleton) jest klasa `System.DbNull`. Jedyny egzemplarz tej klasy jest składowany w jej dzielonej właściwości nazwanej `Value`.

Przykład kodu źródłowego

Przykładowy fragment kodu źródłowego ilustrujący praktyczne zastosowania wzorca projektowego Singleton pochodzi ze szkieletu aplikacji internetowej obsługującej operacje handlu elektronicznego, który krótko opisano w punkcie „Kontekst".

Użytej w tym przykładzie klasy `Currency` nie należy mylić z typem `Currency` obsługiwanym w starszych wersjach języka Visual Basic.

```
' Klasa MonetaryAmountFactory jest singletonem odpowiedzialnym za tworzenie
' obiektów klasy Quantity reprezentujących kwoty pieniężne.
Public Class MonetaryAmountFactory
    Inherits AbstractAmountFactory

    ' Jedyny egzemplarz klasy MonetaryAmountFactory.
    Private Shared ReadOnly myInstance As MonetaryAmountFactory _
        = New MonetaryAmountFactory()

    Private Shared ReadOnly myConverter As ICurrencyConversion _
        = New HardwiredCurrencyConversion()
    ...
    ' Prywatny konstruktor daje nam pewność, że tylko metoda GetInstance
    ' tej klasy będzie mogła tworzyć jej egzemplarze.
    '
    Private Sub New()
        ...
    End Sub
    ...
    ' Zwraca jedyny egzemplarz tej klasy.
    Public Shared Function GetInstance() As MonetaryAmountFactory
        Return myInstance
    End Function

    ' Zwraca obiekt klasy Quantity, który reprezentuje kwotę pieniężną
    ' będącą połączeniem przekazanej na wejściu wartości i waluty.
    Public Function CreateMoney(ByVal theAmount As Decimal, _
                                ByVal theCurrency As Currency) _
                    As IQuantity
        Dim unit As MeasurementUnit = CurrencyToMeasurementUnit(theCurrency)
        Return CreateQuantity(theAmount, unit)
    End Function
    ...
    ' Konwertuje daną na wejściu kwotę pieniężną (egzemplarz interfejsu
    ' IQuantity) do wskazanej waluty docelowej.
    Public Function ConvertMoney(ByVal theQty As IQuantity, _
                                 ByVal theCurrency As Currency, _
                                 ByVal theMaxPrecision As Integer, _
                                 ByVal time As DateTime) As IQuantity
```

```
        ...
    End Function
    ...
End Class
```

Wzorce pokrewne

Wzorzec projektowy Singleton może być łączony z wieloma innymi wzorcami. Szczególnie często stosuje się ten wzorzec łącznie z takimi wzorcami projektowymi jak Abstract Factory, Builder czy Prototype.

Cache Management. Wzorzec projektowy Singleton jest pod pewnymi względami podobny do wzorca projektowego Cache Management. Funkcjonalnie wzorzec Singleton przypomina pamięć podręczną zawierającą tylko jeden obiekt.

Object Pool. Wzorzec projektowy Object Pool opisuje sposób zarządzania dowolnie dużą kolekcją podobnych obiektów, a nie — jak w przypadku wzorca projektowego Singleton — tylko jednym obiektem.

Object Pool

Wzorzec po raz pierwszy zaproponowano w książce *Patterns in Java* [Grand, 2002].

Streszczenie

Wzorzec projektowy Object Pool jest stosowany do zarządzania procesem wielokrotnego wykorzystywania tych samych obiektów w sytuacji, gdy tworzenie nowych egzemplarzy danych klas jest szczególnie kosztowne lub gdy istnieje górne ograniczenie liczby takich egzemplarzy.

Kontekst

Przypuśćmy, że pracujemy w firmie wykorzystującej znaczną liczbę dużych, w wielu przypadkach przestarzałych aplikacji typu mainframe — naszym zadaniem jest poprawa dostępu do przetwarzanych danych w oparciu o technologię ADO.NET. Celem tego projektu jest zapewnienie aplikacjom .NET dostępności do danych przetwarzanych w ramach systemów mainframe na poziomie porównywalnym z dostępnością do zasobów bazy danych SQL Server i innych baz danych zgodnych z technologią .NET.

Budowa mechanizmów ADO.NET odpowiedzialnych za udostępnianie danych wiąże się z koniecznością napisania klas implementujących kilka kluczowych interfejsów. Jednym z nich jest interfejs `System.Data.IDbConnection`. Klasy implementujące ten interfejs odpowiadają za tworzenie połączeń ze źródłami danych, przekazywanie poleceń do tych źródeł, zwracanie wyników generowanych przez źródła danych oraz zapewnianie kontekstu dla przetwarzania transakcyjnego.

Otrzymaliśmy zadanie zaprojektowania klas, które będą implementowały wspomniany interfejs `IDbConnection`.

Przewidujemy, że najbardziej kosztownym spośród wszystkich działań podejmowanych przez egzemplarz interfejsu `IDbConnection` będzie otwieranie połączeń pomiędzy naszymi klasami a niezbędnymi źródłami danych. Nasze prognozy co do kosztów tworzenia połączeń wynikają z kilku powodów:

- Otwarcie każdego połączenia ze źródłem danych może wymagać nawet kilku sekund. Proces otwierania połączenia może się wiązać z koniecznością ustanowienia połączenia sieciowego, zalogowania w aplikacji mainframe oraz wykonania wszystkich operacji w zakresie odnajdywania i wydobywania danych, które są wymagane przez aplikację przed właściwym uzyskaniem żądanych danych.
- Im więcej nawiążemy połączeń z daną aplikacją, tym więcej zasobów będzie czasowo zajętych po stronie oprogramowania mainframe.
- Każde otwarte połączenie zajmuje cenne zasoby także po stronie klienta — najczęściej są to takie zasoby jak pamięć oraz cykle procesora niezbędne do utrzymywania aktywnych połączeń.

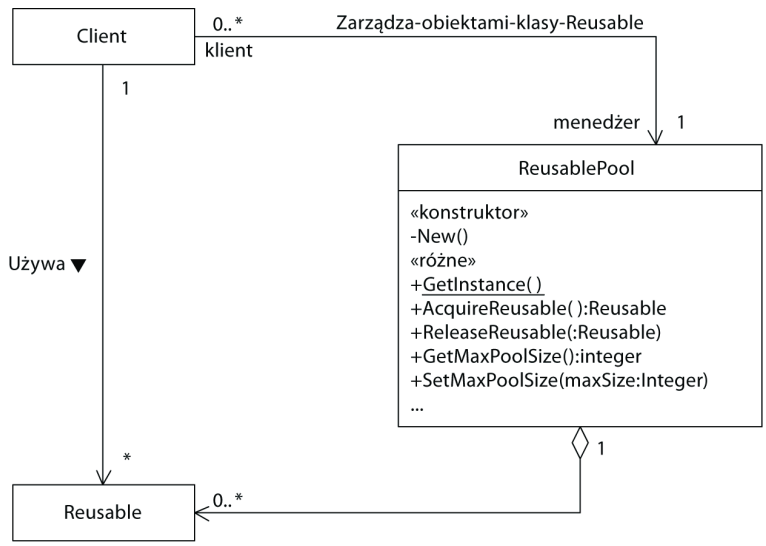
Z uwagi na wysokie koszty operacji otwierania połączeń ze źródłami danych chcemy, by wszędzie tam, gdzie jest to możliwe, nasze klasy implementujące interfejs `IDbConnection` wielokrotnie wykorzystywały połączenia już istniejące, zamiast za każdym razem tworzyć nowe. Nasza implementacja mechanizmu ponownego wykorzystywania utworzonych wcześniej połączeń musi funkcjonować w sposób całkowicie przezroczysty dla uniwersalnych rozwiązań technologii ADO.NET.

W naszej implementacji interfejsu `IDbConnection` zastosujemy strategię zarządzania połączeniami opartą na założeniu, że połączenia otwarte z użyciem tego samego łańcucha połączeń mogą być stosowane zamiennie. Dopóki połączenie z bazą danych znajduje się w stanie umożliwiającym wykonywanie zapytań za jego pośrednictwem, nie ma znaczenia, z którego połączenia skonstruowanego na bazie tego samego łańcucha korzystamy. Zgodnie z tą obserwacją zdecydowaliśmy, że projektowane przez nas implementacje interfejsu `IDbConnection` będą się składały z dwóch warstw.

Klasa nazwana `Connection` będzie implementowała górną warstwę, zatem właśnie z obiektów tej klasy będą bezpośrednio korzystały mechanizmy technologii ADO.NET. Za identyfikację aplikacji będzie odpowiadała właściwość `ConnectionString` obiektu klasy `Connection`. Obiekty tej klasy nie będą jednak bezpośrednio obsługiwały połączeń ze źródłami danych. Bezpośredni związek obiektu klasy `Connection` z egzemplarzem interfejsu `IConnectionImpl` ma miejsce tylko w czasie wysyłania polecenia do aplikacji i odbierania zwracanych wyników. Właściwą reprezentacją połączenia jest właśnie egzemplarz interfejsu `IConnectionImpl`, który występuje w roli dolnej części naszej implementacji.

Klasa `Connection` będzie delegowała do innych klas zadania związane z tworzeniem i ponownym wykorzystywaniem obiektów klasy `ConnectionImpl`. Przykładowo, klasa nazwana `ConnectionPool` będzie odpowiadała za utrzymywanie puli tych obiektów klasy

ConnectionImpl, które w danej chwili nie są związane z żadnymi obiektami klasy Connection. Obiekt klasy ConnectionImpl będzie tworzony tylko wtedy, gdy któryś z obiektów klasy Connection będzie żądał przydziału obiektu klasy ConnectionImpl i gdy pula dostępnych obiektów klasy ConnectionImpl będzie pusta. Projekt naszego systemu przedstawiono na rysunku 4.18.



Rysunek 4.18. *Zarządzanie pulą obiektów klasy ConnectionImpl*

Obiekt klasy Connection wywołuje funkcję AcquireImpl obiektu klasy ConnectionPool w momencie, w którym potrzebuje egzemplarza interfejsu IConnectionImpl. Funkcja AcquireImpl otrzymuje na wejściu łańcuch połączenia. Jeśli któryś z obiektów klasy ConnectionImpl składowanych w wewnętrznej kolekcji obiektu klasy ConnectionPool (puli połączeń) utworzono z tym samym łańcuchem połączenia, funkcja AcquireImpl zwróci właśnie ten obiekt. Jeśli jednak się okaże, że obiekt klasy ConnectionPool nie zawiera żądanego egzemplarza, funkcja AcquireImpl spróbuje utworzyć i zwrócić nowy obiekt. Operacja tworzenia egzemplarza interfejsu IConnectionImpl w tym przypadku polega na przekazaniu łańcucha połączenia do funkcji CreateConnectionImpl obiektu klasy ConnectionFactory. Jeśli próba utworzenia nowego egzemplarza zakończy się niepowodzeniem, obiekt klasy ConnectionPool będzie czekał na zwrócenie (za pomocą wywołania procedury ReleaseImpl) do puli istniejącego obiektu klasy ConnectionImpl i zwróci żądany obiekt dopiero kiedy to nastąpi.

Klasa ConnectionPool jest singletonem. Oznacza to, że powinien istnieć tylko jeden egzemplarz tej klasy. W związku z tym konstruktor klasy ConnectionPool zadeklarowano jako składową prywatną. Pozostałe klasy uzyskują dostęp do jedynego egzemplarza klasy ConnectionPool za pośrednictwem dzielonej funkcji GetInstance.

Istnieje wiele potencjalnych powodów, dla których funkcja AcquireImpl klasy ConnectionPool może nie mieć możliwości utworzenia żądanego obiektu klasy ConnectionImpl. Najczęstszym powodem jest górne ograniczenie maksymalnej liczby połączeń (reprezentowanych przez te obiekty) nawiązanych pomiędzy naszym systemem a daną aplikacją. Źródłem tego rodzaju limitów zwykle są wymagania w zakresie zapewnienia

efektywnej obsługi określonej liczby klientów. Ponieważ każda aplikacja może obsługiwać ograniczoną liczbę połączeń jednocześnie, definiując maksymalną liczbę połączeń ustanawianych pomiędzy taką aplikacją o pojedynczymi klientami, można w prosty sposób zagwarantować jednoczesną obsługę minimalnej liczby klientów.

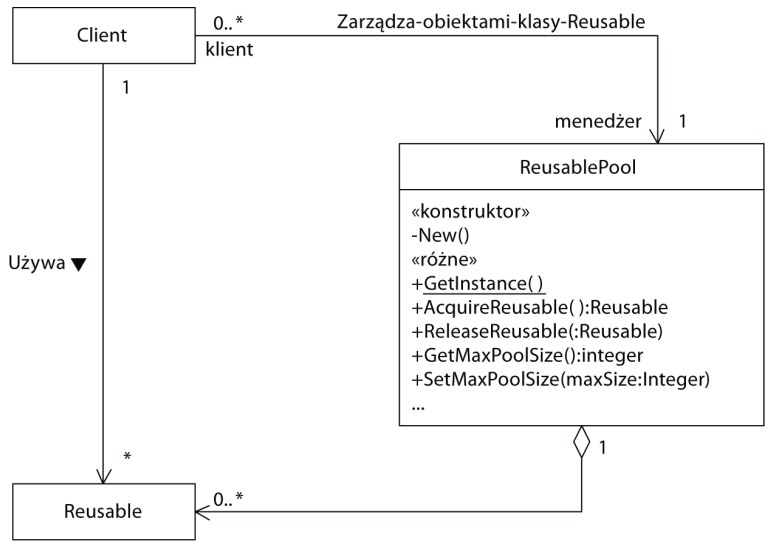
Zalety

- ☺ Program nie może tworzyć więcej egzemplarzy określonej klasy niż przewiduje zdefiniowane dla tej klasy ograniczenie.
- ☺ Tworzenie egzemplarzy danej klasy bywa na tyle kosztowne, że warto czasem stosować mechanizmy wielokrotnego stosowania tych samych egzemplarzy, aby zminimalizować liczbę tych operacji.
- ☺ Można wyeliminować znaczną część operacji tworzenia pewnych obiektów przez ponowne stosowanie utworzonych wcześniej, ale już niepotrzebnych obiektów (zamiast pozwalać na ich usuwanie przez mechanizm odzyskiwania pamięci).
- ☺ Niektóre egzemplarze klasy można stosować zamiennie. Jeśli dysponujesz wieloma egzemplarzami danej klasy, które spełniają określone kryteria, możesz w dowolny sposób wybierać spośród dostępnych obiektów i wykorzystywać je do realizacji kolejnych zadań. W takim przypadku nie ma znaczenia, który konkretnie obiekt wybierzesz do dalszej pracy.
- ☺ Zasoby mogą być zarządzane centralnie (na poziomie pojedynczego obiektu) lub w sposób zdecentralizowany (wówczas za odpowiednie działania odpowiada wiele obiektów). Osiąganie przewidywalnych rezultatów jest dużo prostsze, jeśli za zarządzanie zasobami odpowiada jeden obiekt.
- ☹ Niektóre obiekty wykorzystują najcenniejsze zasoby, których z reguły brakuje. Pewne obiekty mogą zajmować duże ilości pamięci, inne mogą cyklicznie sprawdzać prawdziwość jakiegoś warunku i — tym samym — wykorzystywać cykle procesora i być może przepustowość połączeń sieciowych. Jeśli ilość zasobów wykorzystywanych przez jakiś obiekt jest niewystarczająca, warto rozważyć zaimplementowanie mechanizmu zwalniającego te zasoby na czas, w którym dany obiekt nie jest używany.

Rozwiązanie

Jeśli egzemplarze klasy mogą być wykorzystywane wielokrotnie, należy unikać tworzenia nowych egzemplarzy właśnie przez ponowne stosowanie obiektów już istniejących. Na rysunku 4.19 przedstawiono diagram klas, który dobrze ilustruje role odgrywane przez poszczególne klasy we wzorcu projektowym Object Pool.

W poniższych podpunktach krótko opisano role klas w ramach wzorca projektowego Object Pool (diagramie na diagramie z rysunku 4.19).



Rysunek 4.19. *Wzorzec projektowy Object Pool*

Reusable

Egzemplarze występujących w roli `Reusable` (klas obiektów wielokrotnego użytku) współpracują z pozostałymi obiektami w ograniczonych przedziałach czasowych — kiedy się okazuje, że nie są już potrzebne dotychczasowym obiektom współpracującym, są zwracane do puli. Ważną cechą obiektów klas `Reusable` jest ich wymienność. W skrajnych przypadkach wszystkie obiekty klas `Reusable` mogą być stosowane zamiennie. Dobrym przykładem mogą być klasy sterujące dostępem do linii telefonicznych. Jeśli przyjmiemy, że wszystkie linie telefoniczne są sobie równoważne w kwestii możliwości nawiązywania połączeń, wybór obiektu przydzielanego klientowi nie będzie miał żadnego znaczenia.

Może się zdarzyć, że warunek zamienności będzie spełniony tylko w przypadku obiektów klasy `Reusable` zawierających tę samą wartość pewnej właściwości. Odpowiedni przykład opisano w punkcie „Kontekst", gdzie można było stosować zamiennie tylko te egzemplarze interfejsu `IConnectionImpl`, które reprezentowały tę samą wartość łańcucha połączenia.

Client

Egzemplarze klas występujących w tej roli (klas klienckich) wykorzystują do realizacji swoich zadań obiekty klas `Reusable`. W wielu implementacjach wzorca projektowego Object Pool obiekty klienckie nie współpracują bezpośrednio z obiektami wielokrotnego użytku. Zamiast takiej współpracy wykorzystują jakiś mechanizm pośredniczący, który skutecznie ukrywa istnienie puli obiektów. Takie rozwiązanie omówiono bardziej szczegółowo w punkcie „Implementacja".

ReusablePool

Egzemplarze klas występujących w tej roli (klas puli obiektów wielokrotnego użytku) odpowiadają za zarządzanie obiektami klas `Reusable` wykorzystywanymi przez obiekty klienckie (egzemplarze klas `Client`). Dobrą praktyką jest utrzymywanie wszystkich aktualnie nieużywanych obiektów klas `Reusable` w jednej puli obiektów — dzięki temu można tymi obiektami zarządzać zgodnie z jedną spójną strategią. Aby było to możliwe, `ReusablePool` powinna być klasą singletonu. Konstruktor lub konstruktory tej klasy powinny być prywatne, dzięki czemu pozostałe klasy będą mogły uzyskiwać jej jedyny egzemplarz za pośrednictwem metody `GetInstance` klasy `ReusablePool`.

Obiekt klasy `Client` wywołuje funkcję `AcquireReusable` obiektu klasy `ReusablePool` w momencie, w którym do dalszej pracy potrzebuje obiektu klasy `Reusable`. Obiekt klasy `ReusablePool` utrzymuje kolekcję tych obiektów klasy `Reusable`, które nie są aktualnie używane przez korzystające z tej puli obiekty klienckie. Jeśli w chwili wywołania funkcji `AcquireReusable` wspomniana kolekcja zawiera choć jeden obiekt klasy `Reusable`, funkcja usuwa obiekt z puli i zwraca go klientowi. Jeśli pula jest pusta, funkcja `AcquireReusable` próbuje skonstruować nowy egzemplarz klasy `Reusable`. Jeśli próba utworzenia nowego obiektu się nie powiedzie, funkcja poczeka na zwrócenie do puli któregoś z utworzonych wcześniej obiektów klasy `Reusable`.

Obiekty klasy `Client` przekazują obiekty klasy `Reusable` z powrotem do puli za pomocą procedury `ReleaseReusable` klasy `ReusablePool` w momencie, w którym odkryją, że ich praca z tymi obiektami dobiegła końca. Procedura `ReleaseReusable` zwraca otrzymany na wejściu egzemplarz klasy `Reusable` do puli aktualnie nieużywanych obiektów.

W wielu implementacjach wzorca projektowego Object Pool stosuje się mechanizmy ograniczania łącznej liczby istniejących obiektów klasy `Reusable`. Za to, by zdefiniowana z góry maksymalna liczba obiektów klasy `Reusable` nie została przekroczona, odpowiada wówczas obiekt klasy `ReusablePool`. Jeśli obiekt klasy `ReusablePool` musi pilnować maksymalnego limitu tworzonych przez siebie obiektów, przeważnie zawiera właściwość, której jedynym zadaniem jest reprezentowanie maksymalnej liczby konstruowanych egzemplarzy. Na diagramie przedstawionym na rysunku 4.19 odpowiednią właściwość oznaczono nazwą `MaxPoolSize`.

Implementacja

W poniższych podpunktach opisano kilka problemów, które należy mieć na uwadze podczas implementowania wzorca projektowego Object Pool.

Ukrywanie puli obiektów

W wielu implementacjach wzorca projektowego Object Pool klasy korzystające z obiektów wielokrotnego użytku są izolowane od szczegółów związanych z funkcjonowaniem samej puli obiektów — wymagania stawiane tym klasom sprowadzają się do właściwej pracy wyłącznie z obiektami wielokrotnego użytku (ich pula powinna wówczas

pozostawać przezroczysta dla klas klienckich). Dobrym sposobem zapewnienia takiej przezroczystości puli obiektów z perspektywy klas klienckich korzystających z obiektów wielokrotnego użytku jest zastosowanie wzorca projektowego Façade.

Projekt przedstawiony na rysunku 4.18 uwzględnia dodatkową implementację wzorca projektowego Façade. Klasy klienckie (które nie są reprezentowane na diagramie) komunikują się bezpośrednio z obiektami klasy `Connection` i nie wiedzą, że mają do czynienia z fasadą ukrywającą faktyczne procesy tworzenia i składowania żądanych obiektów w puli.

Delegowanie zadania tworzenia obiektów

Jednym z zadań klasy występującej w roli `ReusablePool` jest tworzenie nowych obiektów wielokrotnego użytku w odpowiedzi na żądania obiektów klienckich. Klasy `ReusablePool` bardzo często delegują zadania związane z tworzeniem obiektów wielokrotnego użytku do innych, wyspecjalizowanych klas. Takie rozwiązanie jest stosowane z dwóch powodów:

- Jeśli proces tworzenia obiektu jest dużo bardziej skomplikowany od wywołania konstruktora odpowiedniej klasy, delegowanie odpowiedzialności za realizację tego procesu do innej klasy może znacznie uprościć i ograniczyć rozmiary klasy `ReusablePool`.
- Wyprowadzanie odpowiedzialności za tworzenie obiektów wielokrotnego użytku poza klasę `ReusablePool` w pewnych sytuacjach może zwiększać szansę ponownego wykorzystywania samej klasy `ReusablePool`.

Zapewnianie maksymalnej liczby egzemplarzy

W wielu przypadkach obiekty zarządzające pulą obiektów muszą ograniczać liczbę tworzonych egzemplarzy wielokrotnego użytku. Implementacja takiego limitu na poziomie klasy, która tworzy tego rodzaju obiekty, nie stanowi oczywiście większego problemu. Warto jednak pamiętać, że naprawdę niezawodny mechanizm ograniczający łączną liczbę tworzonych egzemplarzy wymagałby, aby obiekt odpowiedzialny za zarządzanie ich pulą był jedynym, który może inicjować operacje tworzenia.

Możemy zagwarantować, że egzemplarze danej klasy będą tworzone wyłącznie przez klasę zarządzającą pulą obiektów — wystarczy zadeklarować konstruktor klasy obiektów wielokrotnego użytku jako prywatną składową tej klasy i zaimplementować klasę zarządzającą pulą w formie innej składowej tej samej klasy. Jeśli jednak struktura klasy, której egzemplarze są przedmiotem zarządzania w puli, nie zależy tylko od nas, możemy wprowadzić odpowiednie rozwiązanie, stosując mechanizm dziedziczenia.

Struktura danych

W sytuacji, gdy istnieje ograniczenie łącznej liczby tworzonych obiektów lub po prostu maksymalny rozmiar puli obiektów, najlepszym sposobem zaimplementowania samej puli jest użycie prostej tablicy. Jeśli rozmiar puli obiektów nie jest ograniczany z góry, lepszym rozwiązaniem będzie zaimplementowanie tej puli w formie egzemplarza klasy `ArrayList`.

Ograniczanie rozmiaru puli

Jeśli nie zdecydujemy się na zastosowanie jakiegoś mechanizmu ograniczającego rozmiar puli obiektów, w pewnych okolicznościach może się okazać, że nasza pula rozrośnie się do tego stopnia, że będzie zajmowała ogromną przestrzeń pamięciową, która mogłaby być wykorzystywana znacznie efektywniej przez pozostałe składniki systemu.

W pewnych sytuacjach mogą mieć miejsce chwilowe skoki liczby potrzebnych obiektów wielokrotnego użytku. Może się zdarzyć, że choć przez większość czasu obiekty klienckie potrzebują tylko pięciu lub siedmiu obiektów wielokrotnego użytku, w jednej chwili liczba żądań docierające do puli tych obiektów sięgnie aż stu egzemplarzy. W takich przypadkach składowanie w puli ponad stu obiektów, jeśli przez 99 procent czasu liczba faktycznie wykorzystywanych egzemplarzy nie przekracza dziesięciu, byłoby marnotrawstwem.

Można ten problem rozwiązać, definiując górne ograniczenie liczby egzemplarzy, które mogą być przechowywane w puli obiektów wielokrotnego użytku. Tego rodzaju limity zwykle powinny być niższe od łącznej liczby tworzonych i jednocześnie istniejących obiektów.

Kiedy się okaże, że obiekt zarządzany w ramach puli nie jest już potrzebny obiektowi klienckiemu, obiekt kliencki zwróci go do egzemplarza klasy `ReusablePool`. Jeśli pula będzie już zawierała maksymalną liczbę obiektów wielokrotnego użytku, zwolniony egzemplarz nie zostanie zwrócony do pełnej puli. Jeśli zwolniony egzemplarz wielokrotnego użytku wciąż będzie zajmował jakieś zewnętrzne zasoby, obiekt puli nakaże mu zwolnić te zasoby. Ponieważ jednak z uwagi na osiągnięcie maksymalnej liczby elementów w puli zwolniony obiekt nie zostanie do niej zwrócony, musimy się liczyć z możliwością jego usunięcia przez mechanizm odzyskiwania pamięci.

Zarządzanie obiektami stanowymi

Podstawowe założenie, które leży u podstaw stosowania puli obiektów, mówi, że zarządzane przez nią obiekty mogą być używane wymiennie. Klient uzyskujący obiekt z puli oczekuje, że stan tego obiektu będzie określony i zgodny z przyjętymi kryteriami. Jeśli istnieje możliwość modyfikowania stanów obiektów wielokrotnego użytku przez obiekty klienckie, należy zaimplementować mechanizm odpowiedzialny za przywracanie pierwotnego stanu wszystkim obiektom zwracanym do puli. Poniżej opisano kilka najczęściej stosowanych rozwiązań w tym zakresie:

- Pula wprost przywraca stan obiektu wielokrotnego użytku bezpośrednio przed jego przekazaniem obiektowi klienckiemu. W niektórych przypadkach stosowanie tego rozwiązania może się okazać niemożliwe. Przykładowo, po zamknięciu obiektu połączenia pula obiektów nie ma możliwości jego ponownego otwarcia. Z drugiej strony zamykanie połączeń przez obiekty klienckie byłoby sprzeczne z ideą puli obiektów wielokrotnego użytku.
- Jeśli nasza implementacja wzorca projektowego Object Pool wykorzystuje klasę fasady do ukrywania klas obiektów wielokrotnego użytku przed obiektami klienckimi, umieszczenie w klasie fasady mechanizmu odpowiedzialnego za modyfikowanie stanu obiektów wielokrotnego użytku byłoby dość problematyczne.

- Jeśli nie używamy klasy fasady, możemy uniemożliwić obiektom klienckim modyfikowanie stanu wykorzystywanych przez nie obiektów wielokrotnego użytku, stosując wzorzec projektowy Decorator. Przykładowo, w przypadku obiektów reprezentujących połączenia z bazami danych należałoby utworzyć obiekt opakowania, którego klasa implementowałaby interfejs `IDbConnection`. Taki obiekt delegowałby do właściwego obiektu połączenia wszystkie wywołania funkcji i procedur z wyjątkiem wywołań procedury `Close`, którą można by zaimplementować w klasie opakowania w taki sposób, aby nie podejmowała żadnych działań.

Skutki stosowania

- ☺ Wzorzec projektowy Object Pool pozwala wyeliminować wielokrotnie powtarzane operacje tworzenia nowych obiektów. Wzorzec najlepiej sprawdza się w sytuacji, gdy kolejne żądania obiektów nie różnią się od siebie.
- ☺ Umieszczenie logiki związanej z zarządzaniem procesami tworzenia i wielokrotnego wykorzystywania egzemplarzy danej klasy w innej klasie powoduje, że cały projekt jest bardziej zrozumiały i logicznie spójny. W ten sposób można wyeliminować niepotrzebne związki pomiędzy implementacją strategii tworzenia i ponownego wykorzystywania a implementacją właściwej funkcjonalności zarządzanej klasy.

Przykład kodu źródłowego

Przedstawiony poniżej kod źródłowy jest przykładem ogólnej implementacji wzorca projektowego Object Pool. Ponieważ wszystkie obiekty składowane w puli obiektów w założeniu mogą być stosowane zamiennie, samo określenie typu obiektu obsługiwanego przez pulę gwarantuje nam wystarczającą spójność na poziomie typów.

```
Imports System
Imports System.Collections
Imports System.Threading

' Interfejs implementowany przez pule obiektów.
Public Interface IObjectPool(Of objectType)
    ' Liczba obiektów w puli.
    ReadOnly Property Count() As Integer

    ' Zwraca obiekt składowany w puli lub wartość null, jeśli pula jest pusta.
    Function GetObject() As objectType

    ' Zwraca obiekt składowany w puli. Jeśli pula jest pusta, czeka
    ' do momentu, w którym jakiś obiekt zostanie do niej zwrócony.
    Function WaitForObject() As objectType

    ' Zwraca obiekt do puli.
    Sub Release(ByVal o As objectType)
End Interface 'IObjectPool
```

```
' AbstractObjectPool jest abstrakcyjną klasą bazową dla klas implementujących
' interfejs IObjectPool. Klasa definiuje wspólną logikę dla wszystkich potomnych
' pul obiektów.
Public MustInherit Class AbstractObjectPool(Of objectType)
    Implements IObjectPool(Of objectType)

    ' Ponieważ klasa AbstractObjectPool nie została napisana z myślą o żadnej
    ' konkretnej klasie, nie zaimplementowano w niej mechanizmu tworzenia obiektów
    ' tej klasy (które mają być przedmiotem zarządzania w ramach puli). Zamiast tego
    ' odpowiedzialność za tworzenie obiektów jest delegowana do obiektu klasy
    ' implementującej interfejs nazwany ICreation. Kod interfejsu ICreation przedstawiono
    ' na końcu tego podrozdziału.
    Private creator As ICreation(Of objectType)

    ' creator - obiekt, do którego dana pula będzie delegowała zadanie tworzenia
    '       obiektów zarządzanych w ramach tej puli.
    Public Sub New(ByVal creator As ICreation(Of objectType))
        Me.creator = creator
    End Sub

    ' Obiekt odpowiedzialny za blokowanie dostępu i — tym samym — gwarantujący, że
    ' dostęp do struktury danych tej puli będzie miał jednocześnie tylko jeden wątek.
    Protected MustOverride ReadOnly Property SyncRoot() _
                                            As Object

    ' Zwraca wartość true, jeśli dany obiekt puli może tworzyć inne
    ' obiekty będące przedmiotem zarządzania w ramach tej puli.
    Protected MustOverride Function OkToCreate() As Boolean

    '
    ' Tworzy obiekt, który będzie zarządzany w ramach tej puli obiektów.
    '
    Protected Overridable Function createObject() As objectType
        Return creator.Create()
    End Function 'createObject

    ' Usuwa obiekt z tablicy reprezentującej pulę obiektów i zwraca go na wyjściu.
    ' Jeśli pula jest pusta, zwraca wartość Nothing.
    Protected MustOverride Function removeObject() As objectType

    ' Reprezentuje liczbę obiektów składowanych w puli.
    Public MustOverride ReadOnly Property Count() As Integer _
                              Implements IObjectPool(Of objectType).Count

    '
    ' Zwraca obiekt składowany w puli. Jeśli w danej chwili pula nie zawiera
    ' żadnych obiektów, funkcja tworzy nowy obiekt (chyba że stosowana w tej
    ' puli polityka tworzenia obiektów uniemożliwia konstruowanie nowych
    ' egzemplarzy w bieżącym stanie). Jeśli utworzenie żądanego obiektu nie
    ' jest możliwe, funkcja zwraca wartość Nothing.
    '
    Public Function GetObject() As objectType _
        Implements IObjectPool(Of objectType).GetObject
        SyncLock SyncRoot
            Dim o As objectType = removeObject()
            If o IsNot Nothing Then Return o
            If OkToCreate() Then Return createObject()
            Return Nothing
```

```
            End SyncLock
        End Function 'GetObject

        '
        ' Zwraca obiekt składowany w puli. Jeśli w danej chwili pula nie zawiera
        ' żadnych obiektów, funkcja tworzy nowy obiekt (chyba że stosowana w tej
        ' puli polityka tworzenia obiektów uniemożliwia konstruowanie nowych
        ' egzemplarzy w bieżącym stanie). Jeśli utworzenie żądanego obiektu nie
        ' jest możliwe, funkcja czeka, aż któryś z wcześniej przydzielonych obiektów.
        ' zostanie zwrócony i będzie dostępny do ponownego użycia.
        '
        Public Function WaitForObject() As objectType _
            Implements IObjectPool(Of objectType).WaitForObject
            SyncLock SyncRoot
                Dim o As objectType = removeObject()
                If o IsNot Nothing Then Return o
                If OkToCreate() Then Return createObject()
                Do
                    ' Czeka na sygnał o zwróceniu obiektu do puli
                    ' i możliwości jego ponownego wykorzystania.
                    Monitor.Wait(SyncRoot)
                    o = removeObject()
                Loop While o Is Nothing
                Return o
            End SyncLock
        End Function 'WaitForObject

        ' Zwraca obiekt do puli.
        Public MustOverride Sub Release(ByVal o As objectType) _
                                Implements IObjectPool(Of objectType).Release
    End Class

    '
    ' Klasa SizedObjectPool ogranicza liczbę obiektów składowanych w puli.
    '
    Public Class SizedObjectPool(Of objectType)
        Inherits AbstractObjectPool(Of objectType)

        ' Ponieważ istnieje nieprzekraczalny limit obiektów oczekujących
        ' w egzemplarzu tej klasy na ponowne wykorzystanie, klasa może używać
        ' do ich składowania prostej tablicy. Klasa dodatkowo definiuje
        ' zmienną egzemplarza nazwaną count, która reprezentuje liczbę obiektów
        ' aktualnie oczekujących na ponowne wykorzystanie.
        Private myCount As Integer

        ' Tablica zawiera obiekty oczekujące na ponowne wykorzystanie.
        ' Tablica jest zarządzana zgodnie z regułami stosowanymi dla stosów.
        Private pool() As objectType

        ' Wewnętrzne operacje na tym obiekcie są synchronizowane. Dokładne
        ' wyjaśnienie zasad funkcjonowania  obiektów blokujących znajdziesz w
        ' podrozdziale poświęconym wzorcowi projektowemu Internal Lock Object.
        Private lockObject As New Object()

        ' Blokowanie obiektu ma na celu zagwarantowanie, że struktura danych
        ' puli będzie jednocześnie przetwarzana tylko przez jeden wątek.
        Protected Overrides ReadOnly Property SyncRoot() As Object
```

```
    Get
        Return lockObject
    End Get
End Property

' Zwraca wartość true, jeśli dana pula obiektów może tworzyć
' obiekty innej klasy, które będą przez tę pulę zarządzane.
' Dla tej puli nie zdefiniowano żadnej strategii ograniczania liczby
' jednocześnie składowanych obiektów wielokrotnego użytku, zatem niniejsza
' metoda zawsze będzie zwracała wartość true. Gdyby istniała taka strategia,
' wynik zwracany przez tę metodę byłby uzależniony od tego, czy liczba
' składowanych obiektów jest mniejsza od zdefiniowanego maksimum.
Protected Overrides Function OkToCreate() As Boolean
    Return True
End Function

' c - obiekt, do którego dana pula będzie delegowała zadania związane
'   z tworzeniem obiektów, które będą następnie składowane w tej puli.
' m - maksymalna liczba nieużywanych obiektów, które mogą być jednocześnie
'   składowane w tej puli.
'
Public Sub New(ByVal c As ICreation(Of objectType), ByVal m As Integer)
    MyBase.New(c)
    myCount = 0
    pool = New objectType(m) {}
End Sub

'
' Liczba obiektów aktualnie oczekujących w tej puli na
' ponowne wykorzystanie.
'
Public Overrides ReadOnly Property Count() As Integer
    Get
        Return myCount
    End Get
End Property

' Maksymalna liczba obiektów, które mogą jednocześnie oczekiwać
' w tej puli na ponowne wykorzystanie.
Public Property Capacity() As Integer
    Get
        Return pool.Length
    End Get
    Set(ByVal Value As Integer)
        If Value <= 0 Then
            Throw New ArgumentException( _
              "Pojemność puli musi być większa od zera:" & Value)
        End If
        SyncLock SyncRoot
            ReDim Preserve pool(Value)
        End SyncLock
    End Set
End Property

' Usuwa i zwraca na wyjściu obiekt składowany w tablicy pełniącej funkcję puli.
' Jeśli dana pula jest pusta, funkcja zwraca wartość Nothing.
Protected Overrides Function removeObject() As objectType
    myCount -= 1
```

```
            If Count >= 0 Then
                Return pool(Count)
            End If
            Return Nothing
        End Function 'removeObject

        '
        ' Zwraca obiekt do puli celem umożliwienia jego ponownego wykorzystania.
        '
        ' o - obiekt dostępny do ponownego wykorzystania.
        Public Overrides Sub Release(ByVal o As objectType)
            ' Procedura nie obsługuje wartości null.
            If o Is Nothing Then
                Throw New NullReferenceException()
            End If
            SyncLock SyncRoot
                If Count < Capacity Then
                    pool(Count) = o
                    myCount += 1
                    ' Informuje oczekujący wątek, że do puli trafił
                    ' obiekt gotowy do ponownego wykorzystania.
                    Monitor.Pulse(SyncRoot)
                End If
            End SyncLock
        End Sub 'Release

    End Class 'SizedObjectPool

    ' Klasy puli obiektów delegują zadanie tworzenia nowych obiektów do
    ' egzemplarzy tego interfejsu.
    Public Interface ICreation(Of objectType)
        ' Zwraca nowo utworzony obiekt.
        Function Create() As objectType
    End Interface 'ICreation
```

Wzorce pokrewne

Cache Management. Wzorzec projektowy Cache Management zarządza procesem wielokrotnego wykorzystywania konkretnych lub unikatowych egzemplarzy jakiejś klasy. Wzorzec projektowy Pool zarządza i tworzy egzemplarze klasy, które mogą być stosowane zamiennie.

Façade. Wzorzec projektowy Façade często jest wykorzystywany do ukrywania puli obiektów przed klasami, których klasy korzystają z udostępnianych w ten sposób obiektów wielokrotnego użytku. Przykładem takiego obiektu fasady jest klasa `Connection` z rysunku 4.18.

Factory Method. Wzorzec projektowy Factory Method może być wykorzystywany w implementacji logiki tworzenia obiektów. Warto jednak pamiętać, że ewentualne użycie tego wzorca nie będzie miało wpływu na zarządzanie tymi obiektami już po utworzeniu.

Singleton. Obiekty zarządzające pulami obiektów wielokrotnego użytku zwykle mają postać singletonów.

Thread Pool. Wzorzec projektowy Thread Pool (omówiony w książce *Java Enterprise Design Patterns* [Grand, 2001]) jest wyspecjalizowaną formą wzorca projektowego Object Pool.

Lock Object. W implementacji wzorca projektowego Object Pool można stosować wzorzec projektowy Lock Object.

Layered Architecture. Wzorzec projektowy Object Pool jest w istocie jedną z odmian bardziej uniwersalnego wzorca projektowego Layered Architecture, który opisano w książce *AntiPatterns* [Brown, 1998].