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

C++ dla programistów gier. Wydanie II



Autor: Michael J. Dickheiser
Tłumaczenie: Przemysław Szeremiota
ISBN: 978-83-246-0967-3
Tytuł oryginału: C++ For Game Programmers (Second edition)
Format: B5, stron: 480

Poznaj nowoczesne metody tworzenia gier komputerowych

- Wykorzystaj najefektywniejsze techniki oferowane przez C++
- Popraw czytelność kodu i wydajność programów
- Zastosuj wzorce projektowe

Rynek gier komputerowych jest niezwykle wymagający. Gracze stawiają tego rodzaju programom coraz wyższe wymagania, co z kolei przekłada się na konieczność stosowania coraz doskonalszych technik ich tworzenia. Będąc programistą gier komputerowych, na pewno doskonale zdajesz sobie z tego sprawę. Jeśli chcesz, aby kolejna stworzona przez Ciebie gra spełniała oczekiwania nawet najbardziej wybrednych graczy, wykorzystaj język C++. Jego możliwości sprawiają, że jest doskonałym narzędziem do tworzenia gier.

„C++ dla programistów gier. Wydanie II" to przewodnik po języku C++ opisujący go z punktu widzenia programowania specyficznych aplikacji, jakimi są gry. Książka przedstawia najefektywniejsze techniki C++ i metody rozwiązywania problemów, przed którymi stają programiści gier. Czytając ją, dowiesz się, jak zarządzać pamięcią i stosować wzorce projektowe oraz STL. Poznasz możliwości wykorzystania języków skryptowych do usprawnienia procesu tworzenia gry komputerowej. Każde z rozwiązań opatrzone jest przykładem, dzięki czemu łatwo będzie Ci zaimplementować je w swoich pracach.

- Podstawy języka C++
- Korzystanie z szablonów
- Obsługa wyjątków
- Zarządzanie pamięcią
- Poprawa wydajności aplikacji
- Wzorce projektowe
- Biblioteka STL
- Stosowanie języków skryptowych
- Zarządzanie obiektami
- Serializacja

Dołącz do elitarnej grupy programistów gier komputerowych

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

Spis treści

Wprowadzenie **13**

Część I Elementy C++ **21**

Rozdział 1. Dziedziczenie **23**
Klasy 23
Dziedziczenie 24
Polimorfizm i funkcje wirtualne 27
Dziedziczyć czy nie dziedziczyć? 30
Zasada 1. Dziedziczenie kontra zawieranie 30
Zasada 2. Zachowanie kontra dane 31
Kiedy stosować dziedziczenie, a kiedy go unikać 31
Implementacja dziedziczenia (zaawansowane) 33
Analiza kosztowa (zaawansowane) 35
Alternatywy (zaawansowane) 38
Dziedziczenie a architektura programu (zaawansowane) 39
Wnioski 41
Zalecana lektura 41

Rozdział 2. Dziedziczenie wielobazowe **43**
Stosowanie dziedziczenia wielobazowego 43
Wszystko w jednym 44
Wersja z zawieraniem 44
Dziedziczenie zwyczajne 46
Ratunek w dziedziczeniu wielobazowym 47
Problemy dziedziczenia wielobazowego 47
Niejednoznaczność 48
Topografia 48
Architektura programu 51
Polimorfizm 51
Kiedy stosować dziedziczenie wielobazowe, a kiedy go unikać 53
Implementacja dziedziczenia wielobazowego (zaawansowane) 54
Analiza kosztowa (zaawansowane) 55
Rzutowanie 55
Funkcje wirtualne z drugiej (kolejnej) klasy nadrzędnej 57
Wnioski 58
Zalecana lektura 58

Rozdział 3. O stałości, referencjach i o rzutowaniu ... 59
Stałość ... 59
Koncepcja stałości ... 60
Stałość a wskaźniki ... 60
Stałość a funkcje ... 61
Stałość a klasy ... 64
Stałe, ale zmienne — słowo mutable ... 65
Słowo porady odnośnie const ... 66
Referencje ... 67
Referencje kontra wskaźniki ... 67
Referencje a funkcje ... 68
Zalety referencji ... 69
Kiedy stosować referencje ... 71
Rzutowanie ... 72
Potrzeba rzutowania ... 72
Rzutowanie w konwencji C++ ... 73
Wnioski ... 76
Zalecana lektura ... 76

Rozdział 4. Szablony ... 77
W poszukiwaniu kodu uniwersalnego ... 77
Podejście pierwsze: lista wbudowana w klasę ... 78
Podejście drugie: makrodefinicja ... 79
Podejście trzecie: dziedziczenie ... 80
Podejście czwarte: uniwersalna lista wskaźników void ... 81
Szablony ... 83
Szablony klas ... 83
Szablony funkcji ... 85
Powrót do problemu listy — próba z szablonem ... 85
Wady ... 87
Złożoność ... 87
Zależności ... 87
Rozdęcie kodu ... 88
Obsługa szablonów przez kompilator ... 88
Kiedy stosować szablony ... 88
Specjalizacje szablonów (zaawansowane) ... 89
Pełna specjalizacja szablonu ... 90
Częściowa specjalizacja szablonu ... 91
Wnioski ... 91
Zalecana lektura ... 92

Rozdział 5. Obsługa wyjątków ... 93
Jak sobie radzić z błędami ... 93
Ignorować! ... 93
Stosować kody błędów ... 94
Poddać się (z asercjami)! ... 95
Stosować wywołania setjmp() i longjmp() ... 95
Stosować wyjątki C++ ... 96
Stosowanie wyjątków ... 97
Wprowadzenie ... 98
Zrzucanie wyjątków ... 98
Przechwytywanie wyjątków ... 100

Odporność na wyjątki ... 103
Pozyskiwanie zasobów ... 104
Konstruktory ... 107
Destruktory ... 109
Analiza kosztowa ... 109
Kiedy stosować wyjątki ... 111
Wnioski ... 112
Zalecana lektura ... 112

Część II Wydobywanie mocy C++ ... 113

Rozdział 6. Wydajność ... 115
Wydajność i optymalizacje ... 115
W czasie programowania ... 117
Pod koniec programowania ... 118
Rodzaje funkcji ... 119
Funkcje globalne ... 119
Statyczne funkcje klas ... 120
Niewirtualne składowe klas ... 120
Funkcje wirtualne przy dziedziczeniu pojedynczym ... 121
Funkcje wirtualne przy dziedziczeniu wielobazowym ... 123
Rozwijanie funkcji w miejscu wywołania ... 124
Potrzeba rozwijania w funkcji ... 124
Funkcje rozwijane w miejscu wywołania ... 125
Kiedy stosować rozwijanie zamiast wywołania? ... 127
Jeszcze o narzutach wywołań funkcji ... 128
Parametry funkcji ... 128
Wartości zwracane ... 130
Funkcje puste ... 132
Unikanie kopiowania ... 133
Argumenty wywołań ... 133
Obiekty tymczasowe ... 134
Jawność intencji ... 135
Blokowanie kopiowania ... 135
Dopuszczanie kopiowania ... 136
Przeciążanie operatorów ... 137
Konstruktory i destruktory ... 139
Pamięci podręczne i wyrównywanie danych w pamięci (zaawansowane) ... 143
Wzorce odwołań do pamięci ... 144
Rozmiar obiektów ... 145
Rozmieszczanie składowych w obiektach ... 146
Wyrównanie pamięci ... 146
Wnioski ... 147
Zalecana lektura ... 148

Rozdział 7. Przydział pamięci ... 149
Stos ... 149
Sterta ... 150
Wydajność przydziału ... 151
Fragmentacja pamięci ... 152
Inne problemy ... 154
Przydziały statyczne ... 155
Zalety i wady przydziału statycznego ... 156
Kiedy korzystać z przydziałów statycznych ... 157

Przydziały dynamiczne ... 158
Łańcuch wywołań ... 158
Globalne operatory new i delete ... 159
Operatory new i delete dla klas ... 161
Własny menedżer pamięci ... 163
Kontrola błędów ... 163
Przeglądanie stert ... 166
Zakładki i wycieki ... 166
Sterty hierarchiczne ... 167
Inne rodzaje przydziałów ... 169
Narzut mechanizmu zarządzania pamięcią ... 170
Pule pamięci ... 171
Implementacja ... 172
Podłączanie puli do sterty ... 175
Pule uniwersalne ... 176
W nagłych wypadkach… ... 177
Wnioski ... 178
Zalecana lektura ... 179

Rozdział 8. Wzorce projektowe w C++ ... 181
Czym są wzorce projektowe? ... 181
Wzorzec Singleton ... 183
Przykład: menedżer plików ... 183
Implementacja Singletona ... 184
Wzorzec Façade ... 185
Fasada dla systemu „w budowie” ... 188
Fasada dla przeróbek ... 189
Wzorzec Observer ... 190
Wzorzec Visitor ... 195
Wnioski ... 199
Zalecana lektura ... 199

Rozdział 9. Kontenery STL ... 201
Przegląd STL ... 201
Korzystać czy nie korzystać? ... 203
Wykorzystanie gotowego kodu ... 203
Wydajność ... 204
Wady ... 205
Kontenery sekwencyjne ... 206
Kontener vector ... 206
Kontener deque ... 211
Kontener list ... 214
Kontenery asocjacyjne ... 217
Kontenery set i multiset ... 218
Kontenery map i multimap ... 222
Kontenery haszowane ... 226
Adaptory kontenerów ... 230
Stos ... 231
Kolejka ... 231
Kolejka priorytetowa ... 232
Wnioski ... 233
Zalecana lektura ... 235

Rozdział 10. STL: algorytmy i zagadnienia zaawansowane 237
Obiekty funkcyjne (funktory) 237
Wskaźniki funkcji 237
Funktory 238
Adaptory funktorów 240
Algorytmy 241
Algorytmy niemodyfikujące 242
Algorytmy modyfikujące 245
Algorytmy sortujące 248
Uogólnione algorytmy numeryczne 249
Ciągi znaków 250
Ciągle bez ciągów 250
Klasa string 252
Wydajność 254
Pamięć 255
Alternatywy 256
Alokatory (zaawansowane) 257
Kiedy STL nie wystarcza (zaawansowane) 260
Wnioski 262
Zalecana lektura 262

Rozdział 11. Poza STL: własne struktury i algorytmy 265
Grafy — studium przypadku 265
Powtórka z grafów 265
Ogólniej o grafie 267
W kwestii kosztów 267
Koszt przebycia krawędzi a jakość rozgrywki 268
Grafy w C++ 269
Zaprząc grafy do pracy 271
„Inteligencja" grafów 276
Życie na krawędzi 276
Aktualizacja, wracamy do C++ 279
Implementacja wyznaczania trasy 285
A może A* (zaawansowane) 293
Inne zastosowania grafów 294
Optymalizacja przejść w interfejsie użytkownika 296
Brakujące ścieżki powrotne 298
Zagubione plansze menu 298
Wnioski 299
Zalecana lektura 299

Część III Techniki specjalne 301

Rozdział 12. Interfejsy abstrakcyjne 303
Interfejsy abstrakcyjne 303
Implementacja interfejsów w C++ 305
Interfejsy abstrakcyjne w roli bariery 306
Nagłówki i wytwórnie 308
Z życia 310
Interfejsy abstrakcyjne w roli charakterystyk klas 312
Implementacje 313
Pytanie o interfejs 315
Rozszerzanie gry 317
Nie wszystko złoto… 319
Wnioski 320
Zalecana lektura 321

Rozdział 13. Wtyczki 323
Po co komu wtyczki 323
Wtyczki do cudzych programów 324
Wtyczki do własnych programów 325
Architektura wtyczek 326
Interfejs wtyczek 326
Tworzenie konkretnych wtyczek 327
Obsługa różnych rodzajów wtyczek 328
Ładowanie wtyczek 329
Menedżer wtyczek 331
Komunikacja dwukierunkowa 333
Żeby miało ręce i nogi… 335
Wtyczki w praktyce 335
Stosowanie wtyczek 336
Wady 336
Inne platformy 337
Wnioski 338
Zalecana lektura 338

Rozdział 14. C++ a skrypty 339
Po co jeszcze jeden język, i to skryptowy? 339
Złe wieści… 340
I wieści dobre 341
Rozważania o architekturze 344
Engine kontra gameplay 345
Zintegrowany interpreter skryptów — nie tylko w sterowaniu rozgrywką 349
Konsola 349
Interaktywne sesje diagnostyczne 350
Szybkie prototypowanie 352
Automatyzacja testów 353
Wnioski 355
Zalecana lektura 356
Lua 356
Python 357
GameMonkey Script 357
AngelScript 358

Rozdział 15. Informacja o typach w czasie wykonania 359
Praca bez RTTI 359
Używanie i nadużywanie RTTI 361
Standardowe RTTI w C++ 363
Operator dynamic_cast 363
Operator typeid 365
Analiza RTTI w wydaniu C++ 366
Własny system RTTI 368
Najprostsze rozwiązanie 368
Z obsługą dziedziczenia pojedynczego 372
Z obsługą dziedziczenia wielobazowego 375
Wnioski 378
Zalecana lektura 378

Rozdział 16. Tworzenie obiektów i zarządzanie nimi 379
Tworzenie obiektów 379
Kiedy new nie wystarcza 380
Wielka selekcja 381
Wytwórnie obiektów 382
Prosta wytwórnia 382
Wytwórnia rozproszona 383
Jawne rejestrowanie obiektów wytwórczych 384
Niejawne rejestrowanie obiektów wytwórczych 385
Identyfikatory typów obiektów 387
Od szablonu 388
Obiekty współużytkowane 389
Bez wspólnych obiektów 390
Ignorowanie problemu 391
Niech się właściciel martwi 392
Zliczanie odwołań 393
Uchwyty 396
Inteligentne wskaźniki 398
Wnioski 401
Zalecana lektura 402

Rozdział 17. Utrwalanie obiektów 405
Przegląd zagadnień dotyczących utrwalania jednostek gry 405
Jednostki kontra zasoby 406
Najprostsze rozwiązanie, które nie zadziała 406
Czego potrzebujemy 407
Implementacja utrwalania jednostek gry 409
Strumienie 409
Zapisywanie 412
Wczytywanie 416
Wespół w zespół 419
Wnioski 420
Zalecana lektura 421

Rozdział 18. Postępowanie z dużymi projektami 423
Struktura logiczna a struktura fizyczna 423
Klasy i pliki 425
Pliki nagłówkowe 426
Co ma się znaleźć w pliku nagłówkowym? 426
Bariery włączania 427
Dyrektywy #include w plikach implementacji 430
Dyrektywy #include w plikach nagłówkowych 432
Wstępnie kompilowane pliki nagłówkowe 434
Wzorzec implementacji prywatnej 437
Biblioteki 440
Konfiguracje 443
Konfiguracja diagnostyczna 444
Konfiguracja dystrybucyjna 444
Konfiguracja diagnostyczna zoptymalizowana 445
Wnioski 445
Zalecana lektura 446

Rozdział 19. Zbrojenie gry 447
Stosowanie asercji 447
Kiedy stosować asercje 448
Kiedy nie stosować asercji 450
Własne asercje 451
Co powinno się znaleźć w ostatecznej wersji 452
Własna przykładowa implementacja asercji 454
Zawsze świeżo 455
Wycieki pamięci 456
Fragmentacja pamięci 456
Dryf zegara 456
Kumulacja błędu 457
Co robić? 457
Postępowanie ze „złymi" danymi 458
Wykrywać asercjami 459
Radzić sobie samemu 459
Kompromis 461
Wnioski 463
Zalecana lektura 463

Skorowidz 465

Rozdział 8.

Wzorce projektowe w C++

W tym rozdziale przygotujemy grunt na najbardziej zaawansowane elementy C++, które przyjdzie nam omawiać w miarę zbliżania się do końca książki. Odejdziemy na chwilę od szczegółów implementacyjnych i wejdziemy na nieco wyższy poziom abstrakcji projektowej, którą stanowią właśnie **wzorce projektowe** (ang. *design patterns*). Lektura tego rozdziału zaowocuje lepszym zrozumieniem tego, jak bardzo język C++ odszedł od starego C; łatwiej też będzie docenić to, w jaki sposób język programowania wspiera projektowanie na wysokim, naturalniejszym dla projektanta poziomie.

Czym są wzorce projektowe?

Każdy pamięta, że kiedy pierwszy raz uczył się programowania, to jego pierwsze wrażenie z kontaktu z językiem programowania — dowolnym językiem programowania — było wrażeniem obcości i niezrozumiałości: z początku widać tylko niejasne symbole i formacje tylko szczątkowo przypominające jakieś konstrukcje języka naturalnego. Wkrótce jednak przychodzi oswojenie z symbolami i składnią, a także świadomość możliwości algorytmicznych wynikających z tej składni. Po opanowaniu podstaw języka okazuje się, że można czytać i postrzegać kod programu niejako w sposób naturalny, bez biedzenia się nad rozbiorem poszczególnych instrukcji. Skrótowce takie jak `x += 1` tracą swoją obcość i stają się naturalnym sposobem inkrementacji zmiennej.

Po przyswojeniu pojęć zmiennych i operatorów zaczynamy oswajać się ze strukturami sterującymi: w miarę zaznajamiania się z koncepcjami sterowania wykonaniem programu okazuje się, że można dość swobodnie rozmawiać z innymi programistami o naturze pętli i instrukcji wyboru. Potem poznaje się podstawowe struktury, najważniejsze funkcje standardowe i coraz bardziej zaawansowane elementy języka. Na tym etapie zrozumienie struktury programu osiąga nowy poziom, tak samo jak zdolność do komunikowania się ze współpracownikami prawie że w samym języku programowania — w każdym razie przekazywanie współpracownikom koncepcji programistycznych przychodzi coraz łatwiej. Pojawia się przyspieszenie postępów nauki, ponieważ co łatwiejsze koncepcje stają się intuicyjne, a te coraz bardziej skomplikowane również w miarę postępów stanowią coraz mniejsze łamigłówki.

Taka jest natura poznawania i opanowywania każdego systemu, który złożoność i wyrafinowanie opiera na zestawie prostszych, niepodzielnych dalej elementów; programowanie jest znakomitym przykładem takiego systemu. Szczególnie dobrymi przykładami są języki przewidziane do programowania obiektowego, bo ze swojej natury przenoszą uczącego się na coraz wyższe poziomy abstrakcji, pozwalając zaszczepiać zrozumienie niskopoziomowych konstrukcji do łatwiej przyswajalnych (albo bardziej logicznych) struktur wysokiego poziomu.

I wtedy dochodzimy do **wzorców**. **Wzorce projektowe** są całkiem już abstrakcyjnymi (oderwanymi od języka programowania) strukturami, które zarówno ujmują naturę typowych problemów projektowych, rozwiązywanych od nowa w każdym kolejnym projekcie, jak i proponują powszechnie przyjęte i wypróbowane rozwiązania tych problemów. Przykładami takich problemów są:

- Jak zapewnić globalny dostęp do zestawu klas bez zaśmiecania globalnej przestrzeni nazw i z utrzymaniem pożądanej dla projektowania obiektowego hermetyzacji abstrakcji.
- Jak napisać kod operujący na wspólnych elementach radykalnie odmiennych klas, ze wspólnym, dobrze pomyślanym interfejsem do nowych funkcji.
- Jak umożliwić obiektom różnych klas utrzymywanie referencji do siebie wzajemnie bez ryzykowania wiszących wskaźników i znikania obiektów na granicach zasięgów.

W tym rozdziale skupimy się na czterech popularnych wzorcach projektowych, szczególnie przydatnych w projektowaniu i programowaniu gier. Wzorce te zostały wybrane do omówienia ze względu na to, że można je z powodzeniem zastosować w praktycznie każdym podsystemie gry. I najpewniej stosowaliśmy je już z dobrym skutkiem, choć bez świadomości, że nasze techniki zostały skatalogowane i opisane jako wzorce projektowe. Jeśli tak, to wypada tylko się cieszyć, bo to znaczy, że kod już jest przynajmniej w jakichś częściach organizowany zgodnie z najlepszą wiedzą. Z drugiej strony nieświadomość korzystania z wzorców projektowych oznacza, że nie było dotąd okazji do dołączenia opisanych rozwiązań do własnego słownika projektowego. To z kolei uniemożliwia przekazywanie wysoce abstrakcyjnych koncepcji projektowych innym programistom, co było podstawowym motywem do katalogowania wzorców projektowych.

Ale nic straconego. Po lekturze tego rozdziału będziemy już dysponowali solidną dawką wiedzy o kilku ważnych i popularnych wzorcach:

- Singleton
- Façade („fasada”)
- Observer („obserwator”)
- Visitor („wizytator”)

Każdy z tych wzorców projektowych zostanie omówiony w osobnym podrozdziale. Omówienie będzie z konieczności i celowo uproszczone, dla łatwiejszego przetrawienia przedstawianych pojęć. Zachęcam jednak Czytelników do uzupełnienia wiedzy

lekturą książek wymienianych pod koniec rozdziału; w nich można znaleźć znacznie więcej informacji o wszystkich omawianych wzorcach, istotnych dla podjęcia decyzji o ich ewentualnym wdrożeniu do własnych projektów.

Wzorzec Singleton

Na pierwszy ogień pójdzie **wzorzec Singleton**. Jak sama nazwa wskazuje, Singleton to klasa, która ma tylko (najwyżej) jeden egzemplarz. Ogólne zadanie Singletona to udostępnianie centralnego miejsca dla zestawu funkcji, które mają być dostępne globalnie dla całej reszty programu. Singleton powinien udostępniać otoczeniu dobrze zdefiniowany interfejs i hermetyzować jego implementację, co doskonale się zgadza z filozofią C++.

Kolejną cechą Singletona jest to, że jedyny egzemplarz jest chroniony również przed ingerencjami z zewnątrz. Jak się wkrótce okaże, jedynym sposobem na utworzenie egzemplarza jest publiczny interfejs jego klasy — nie można samowolnie utworzyć Singletona za pomocą któregoś z jego konstruktorów! Ale zanim przejdziemy do szczegółów tworzenia egzemplarza, przyjrzymy się przykładowemu systemowi stanowiącemu idealnego kandydata na implementację wedle wzorca Singleton.

Przykład: menedżer plików

Praktycznie każda gra posiada własny podsystem obsługi plików w zakresie otwierania, zamykania i modyfikowania plików. Taki podsystem jest zwykle określany mianem **menedżera plików** i zazwyczaj dla całej gry jest jeden wspólny taki system, bo plików zwykle potrzeba mnóstwo, ale do zarządzania nimi wystarczy pojedynczy menedżer. Podejrzewamy już, że taki menedżer może zostać zaimplementowany w postaci klasy, na przykład takiej jak poniższa:

```
class FileManager
{
public:
   FileManager();
   ~FileManager();
    bool FileExists(const char* strName) const;
    File* OpenFile(const char* strName, eFileOpenMode mode);
    bool CloseFile(File* pFile);
    // itd...
protected:
    // Tu „wnętrzności" menedżera
};
```

Aby skorzystać z menedżera plików, wystarczy utworzyć obiekt klasy:

```
FileManager fileManager;
```

A potem można już z niego korzystać:

```
File* pFP = FileManager.OpenFile("ObjectData.xml", eReadOnly);
```

Kod będzie działał zupełnie dobrze, ale takie podejście kryje kilka problemów, zwłaszcza w większych, bardziej rozbudowanych projektach. Przede wszystkim nie ma żadnego zabezpieczenia przed tworzeniem wielu obiektów menedżera plików. A ponieważ taki menedżer nie jest zwykle nijak powiązany z konkretną hierarchią klas ani z konkretnym podsystemem programu, tworzenie kilku menedżerów jest doprawdy marnotrawstwem. A co gorsza, operacje na plikach zwykle angażują jakąś pulę zasobów sprzętowych i w grze zaimplementowanej wielowątkowo obecność wielu menedżerów plików odczytujących i zapisujących te potencjalnie wspólne zasoby to proszenie się o kłopoty.

To, czego nam trzeba, to prosty menedżer, który będzie wyłącznym odpowiedzialnym za dostęp do owych zasobów, tak aby zawsze można było z niego bezpiecznie korzystać również w kodzie wielowątkowym; dobrze by było także, gdyby był elegancko hermetyzowany. Można by pomyśleć, że rozwiązanie jest banalne: wystarczy przecież, że ograniczymy się do utworzenia pojedynczego egzemplarza klasy menedżera plików! Cóż, może i wystarczy, ale takie założenie nie daje **żadnej** gwarancji, że kto inny nie utworzy innego egzemplarza. A nawet jeśli uda się wymusić na użytkownikach obecność tylko jednego egzemplarza, to gdzie niby powinien zostać utworzony? I jak użytkownicy mają się do niego odwoływać? Na pewno nasuwa się odpowiedź: wystarczy, żeby obiekt menedżera plików znajdował się w zasięgu globalnym. Może nasuwają się też inne rozwiązania, ale przejdźmy od razu do najlepszego: niech menedżer plików zostanie Singletonem.

Implementacja Singletona

Wiemy już, dlaczego należałoby wdrożyć do projektu właśnie ten wzorzec, więc pora go zaimplementować. Zaczniemy od podstawowej wersji klasy.

```
class Singleton
{
public:
    static Singleton * GetInstance()
    {
        return s_pInstance;
    }

    static void Create();
    static void Destroy();
protected:
    static Singleton * s_pInstance;
    Singleton();  // ukryty konstruktor!
};
```

A oto jej implementacja:

```
// Inicjalizacja wskaźnika jedynego egzemplarza wartością NULL
Singleton* Singleron::s_pInstance = NULL;

// Funkcja Create()
static void Singleton::Create()
{
    if (!s_pInstance)
```

```
    {
        s_pInstance = new Singleton;
    }
}

// Funkcja Destroy()
static void Singleton::Destroy()
{
    delete s_pInstance;
    s_pInstance = NULL;
}
```

Zwróćmy uwagę na jawne zastosowanie metod `Create()` i `Destroy()`. Można by co prawda ukryć proces tworzenia jedynego egzemplarza klasy za wywołaniem `GetInstance()`, zdając się na „opóźnione" (ang. *lazy*) utworzenie tego egzemplarza, odłożone do momentu pojawienia się pierwszego odwołania do egzemplarza. Ale ponieważ w Singletonach elegancja zaleca posiadanie zewnętrznego obiektu tworzonego i usuwanego jawnie, zastosujemy się do tego zalecenia. Kontynuując przykład z menedżerem plików, możemy teraz zrealizować go jako Singleton i utworzyć na początku egzemplarz menedżera:

```
FileManager::Create();
```

i od tego momentu korzystać z niego jak poprzednio:

```
FileManager::GetInstance()->OpenFile("ObjectData.xml", eReadOnly);
```

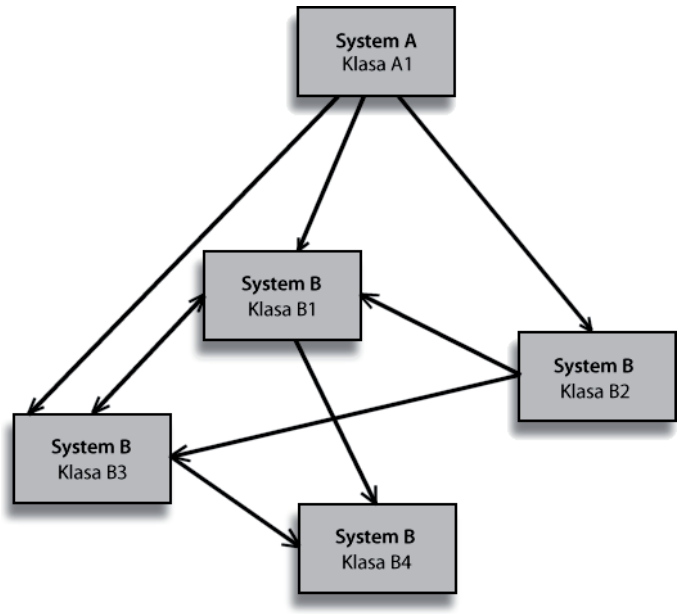
Po zakończeniu korzystania z egzemplarza menedżera plików przy kończeniu programu należy usunąć egzemplarz:

```
FileManager::Destroy();
```

Wzorzec Façade

Następny wzorzec na naszej rozkładówce to **wzorzec Façade** (fasada). Z nazwy wynikałoby, że to jakby „sztuczny fronton"— przykrywka czegoś, co jest ukryte przed okiem widza. W programowaniu fasada jawi się zwykle jako interfejs do zestawu systemów albo klas, niekoniecznie ściśle ze sobą powiązanych. Zasadniczo fasada stanowi więc otoczkę ujednolicającą różnorakie interfejsy wszystkich tych systemów składowych do postaci interfejsu na przykład prostszego albo bardziej dostępnego.

Dla przykładu weźmiemy klasę ilustrowaną rysunkiem 8.1. Zauważmy, że klasa w systemie A jest powiązana z kilkoma klasami w systemie B. W takim układzie w systemie B nie można mówić praktycznie o hermetyzacji systemu, bo o klasach `B1`, `B2` i `B3` musi wiedzieć klasa `A1`. Jeśli system B ulegnie w przyszłości zmianie w jakimkolwiek niepowierzchownym zakresie, jest wielce prawdopodobne, że klasa `A1` również będzie musiała zostać zmieniona.



Rysunek 8.1.
Klasa w systemie A jest powiązana z potencjalnie wieloma klasami w systemie B

Wyobraźmy sobie, że system B to podsystem generowania grafiki, a system A to klient tego systemu, na przykład interfejs użytkownika (menu gry). Klasami w podsystemie graficznym mogą być na przykład: klasa zarządzająca teksturami, klasa odrysowująca, klasa zarządzająca czcionkami czy klasa zarządzająca nakładkami (ang. *overlay*) 2D. Jeśli interfejs użytkownika będzie miał na ekranie wyświetlić komunikat tekstowy w jakimś przyjemnym dla oka formacie, będzie zapewne musiał odwoływać się do usług wszystkich tych czterech klas:

```
Texture* pTexture = GfxTextureMgr::GetTexture("CoolTexture.tif");

Font* pFont = FontMgr::GetFont("UberRoman.fnt");

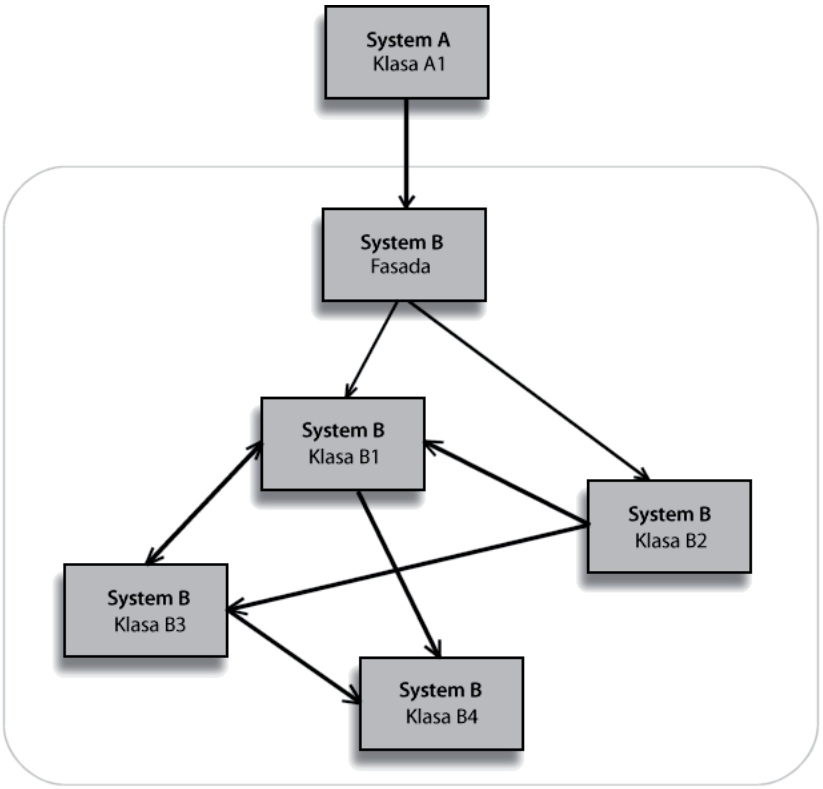
Overlay2d* pMessageBox = OverlayManager::CreateOverlay(pTexture,
                              pFont, "Hello World!");

GfxRenderer::AddScreenElement(pMEssageBox);
```

Jeśli do systemu B wprowadzimy fasadę, będziemy mogli uzyskać wreszcie tak pożądany efekt hermetyzacji systemu i udostępnić jednolity, wspólny interfejs, lepiej i bezpośrednio dostosowany do potrzeb systemu A. Pomysł ten ilustrowany jest rysunkiem 8.2. Zauważmy, że wprowadzenie do systemu B wzorca Façade powoduje, że w systemie A możemy korzystać z usług B za pośrednictwem pojedynczej klasy.

Kontynuujmy przykład z podsystemem graficznym. Interfejs użytkownika może teraz zrealizować zadanie wyświetlenia okienka dialogowego za pośrednictwem pojedynczego wywołania:

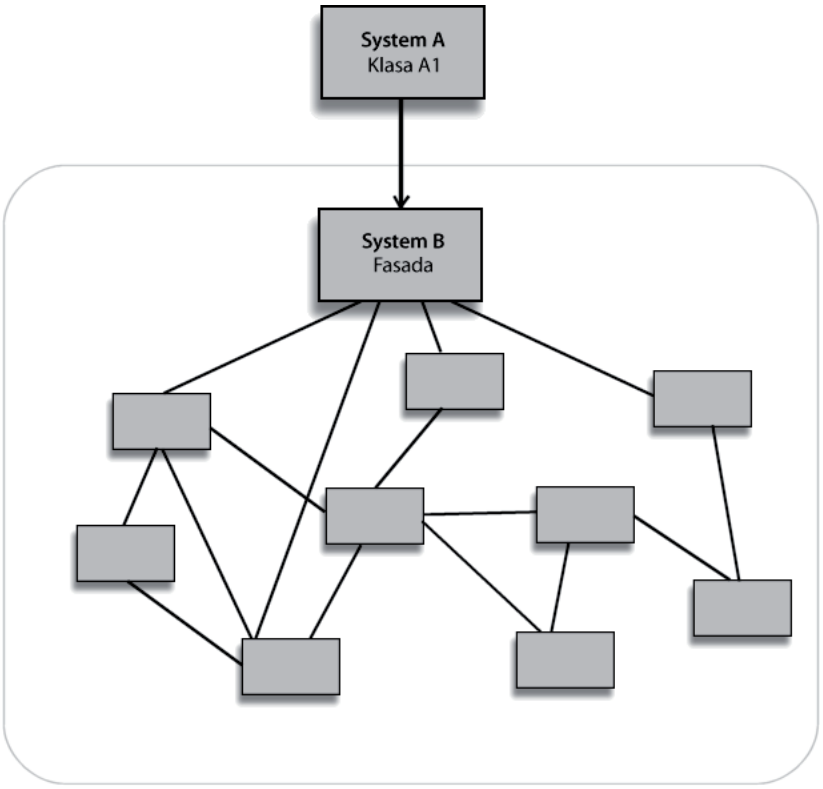
```
// GraphicsInterface to nasza nowa fasada przesłaniająca
// usługi podsystemu graficznego
GraphicsInterface::DisplayMsgBox("CoolTexture.tif",
                                 "UberRoman.fnt",
                                 "Hello World!");
```



Rysunek 8.2. *Fasada systemu B efektywnie hermetyzuje system i udostępnia pojedynczy punkt wejścia dla klientów systemu*

Tymczasem klasa `GraphicsInterface` może za kulisami, to jest wewnętrznie, inicjować kolejno te same wywołania, którymi jawnie posługiwał się interfejs użytkownika w poprzedniej wersji. Ale niekoniecznie; zresztą implementacja wywołania `DisplayMsgBox` może się od tego momentu dowolnie niemal zmieniać, ale dzięki dodatkowemu pośrednictwu fasady nie wymusi to żadnych zmian w kodzie interfejsu użytkownika, przed którym te zmiany zostaną skutecznie ukryte. Interfejs użytkownika jako klient systemu A powinien przetrwać nawet radykalne zmiany struktury wewnętrznej systemu B. Jak na rysunku 8.3, gdzie rozbudowa systemu nie musi być wcale widoczna dla któregokolwiek z zewnętrznych użytkowników systemu B.

Przykład z podsystemem graficznym był dość oczywisty i pewnie każdy, kto implementowałby taki system samodzielnie, wpadłby na pomysł udostępnienia dla całego podsystemu wspólnego interfejsu, za którym można by skutecznie chować szczegóły implementacji; i mało kto wiedziałby w ogóle, że stawia fasadę, tak jak rzadko uzmysławiamy sobie, że mówimy prozą. Ale można też podać mniej oczywiste przykłady zastosowania wzorca Façade — o nich za chwilę.



Rysunek 8.3. *Ponieważ system B wdrożył wzorzec Façade, może się niemal dowolnie zmieniać — nawet dość radykalnie — bez wymuszania ingerencji w kodzie po stronie użytkowników systemu*

Fasada dla systemu „w budowie”

Ogólnie mówiąc, jeśli stoimy dopiero na starcie do projektu i musimy zarysować techniczny projekt systemu, najlepiej zacząć od rozważenia wszystkich wymagań z punktu widzenia użytkowników tego systemu. W ten sposób może powstać spis wszystkich klas, które należałoby zaimplementować, spis interfejsów tych klas, opis ogólnej struktura systemu i tak dalej.

Najlepiej byłoby, gdyby harmonogram projektu dawał wystarczająco dużo czasu, aby wcześniej można było przygotować znakomity projekt systemu. Ale w praktyce bywa raczej tak, że czasu jest za mało i projektowanie trzeba zarzucić na rzecz implementowania. Niedobory czasowe są zresztą typowe dla wszystkich etapów projektu i niekiedy nie można sobie pozwolić na jakiekolwiek wydłużanie poszczególnych faz i trzeba oddawać kolejne elementy tak szybko, jak się da. Jeśli do tego na udostępnienie systemu czekają nasi współpracownicy, ich oddech na naszym karku staje się palący, a każde opóźnienie powoduje groźne napięcia w zespole — bo koledzy nie będą mogli z kolei zrealizować na czas swoich zadań.

W takiej sytuacji pojawia się presja na to, aby zgodnie z regułami sztuki skupić się na dokończeniu projektowania przed przystąpieniem do implementacji systemu. Ale w większości przypadków byłoby to marnotrawstwem sporej ilości czasu. Prawda, że użytkownicy systemu polegają w swoim kodzie na stabilnym interfejsie, na którym mogliby opierać swoje implementacje i który nie wymuszałby zmian w ich kodzie w razie zmiany naszego systemu. Z drugiej strony klienci systemu nie zawsze mogą oprogramować swoje własne systemy, bo **ich** projekty mogą być uzależnione od różnorakich czynników, takich jak choćby wydajność naszego systemu i jego zakres funkcji.

Problemem jest współzależność i należałoby ją zerwać w przynajmniej jednym kierunku. Wstrzymywanie tworzenia klientów naszego systemu powoduje silne szeregowanie zadań w projekcie, które przecież po to się rozdziela, żeby je możliwie silnie zrównoleglić. Innymi słowy, trzeba się skupić na szybkim zaimplementowaniu i udostępnieniu podstawowych funkcji systemu, aby inni mogli na tej podstawie rozwijać własne podsystemy.

Rozwiązaniem jest utworzenie interfejsu tymczasowego — fasady — która pozwala na korzystanie z elementów naszego systemu w czasie, kiedy ten jest dopiero w budowie. Nawet jeśli interfejs systemu ostatecznie będzie zupełnie inny, to przynajmniej współpracownicy będą mogli dzięki niemu podjąć równoległe prace nad swoimi częściami projektu. Co prawda będą musieli może nawet kilkakrotnie reagować na zmiany w interfejsie naszego systemu, ale korzyść ze zrównoleglenia prac, a zwłaszcza postępy współpracowników w rozwijaniu implementacji wewnętrznych elementów ich podsystemów, niemających związku z naszym kawałkiem projektu, mogą zrekompensować koszty zmian interfejsu. Dodatkowe zalety tego podejścia ujawnią się w następnym przykładzie.

Fasada dla przeróbek

Pora na kolejne zastosowanie wzorca Façade, pozornie dość podobne do fasady dla systemów w budowie. Otóż kiedy znajdziemy się w potrzebie „refaktoryzacji" całego podsystemu, w naszym projekcie powinniśmy rozważyć zastosowanie **fasady refaktoryzacji** jako środka oddzielenia „dobrego" kodu, który ma pozostać niezmieniony, od „złego" kodu, który ma być przerobiony. Typowo w czasie przerabiania systemu dochodzi do wprowadzania licznych zmian w interfejsach różnych klas, co potencjalnie wymusza znaczące zmiany w interfejsie systemu jako całości.

Wyobraźmy sobie, że dostaliśmy w spadku po poprzednikach dość przestarzały system efektów cząsteczkowych i stoimy przed zadaniem przerobienia go i uruchomienia tak, aby współdziałał z licznymi, znacznie nowszymi systemami gry. Zastany system może nie posiadać elementów i funkcji oczekiwanych w nowym środowisku albo po prostu nie spełniać wszystkich wymagań narzucanych przez projekt gry. Trzeba więc przepisać („refaktoryzować", kiedy rozmawia się z szefem) liczne klasy w zastanym systemie, dodać do niego nowe klasy i pozbyć się niektórych niepotrzebnych. Oczywiście całe otoczenie podsystemu natychmiast przestanie działać, przez co koledzy rozwijający inne podsystemy, uzależnione od systemu efektów cząsteczkowych, zostaną zatrzymani w swoich pracach. Mamy wtedy dwie opcje.

Pierwsza to dłubanina i orka w zastanym systemie, która czasowo całkowicie blokuje jakiekolwiek korzystanie z efektów cząsteczkowych w grze. Wynikły z tego przestój w rozwoju projektu może dotyczyć niemal każdego programisty w zespole, od programistów podsystemu graficznego, którzy muszą mieć na oku inwentarz cząsteczkowy, przez artystów, którzy chcieliby móc testować różne nowe tekstury, po projektantów poziomów, którzy chcieliby, żeby ich dzieła działały. Nawet jeśli z początku wydaje się, że uda się przerobić system dość szybko, zawsze jest ryzyko nieprzewidzianych trudności i opóźnień w przywróceniu kodu do stanu używalności. Czasem najgorsze, co się może przytrafić projektowi, to właśnie wzięcie działającego podsystemu, który kierownictwo i wydawca uznali za działający, i zablokowanie dostępu do niego. Z punktu widzenia osób niezaangażowanych w projekt konieczność przerobienia podsystemu oznacza nie postęp, ale regres całego projektu. Dla kontynuacji projektu najlepiej jest, jeśli w każdym momencie można „czynnikom decyzyjnym" pokazać możliwie dużo działającego kodu i możliwie wiele chociaż jako tako sprawnych podsystemów.

Weźmy więc pod rozwagę drugą możliwość: ustanowienie wzorca Façade jako tymczasowego interfejsu, który udostępniałby zarówno nowe funkcje i elementy podsystemu, jak i te, które zostaną pozostawione ze starego. Zazwyczaj wiadomo już, jakie cechy ma reprezentować nowy system (od czego są regularne konsultacje z projektantami?), więc ów tymczasowy interfejs, fasada dla przeróbek napisze się właściwie sama. Następny krok będzie polegał na zastosowaniu fasady jako otoczki dla zachowywanych funkcji przerabianego podsystemu. Pozwoli to podtrzymać dostępność zachowywanych elementów na czas, kiedy system będzie przerabiany. A wszystkie nowe funkcje i elementy również powinny być udostępniane poprzez tymczasową fasadę.

Po wdrożeniu fasady dla przeróbek (co nie powinno zająć dużo czasu) można już swobodnie kontynuować prace nad „wnętrznościami" nowego systemu, a wszyscy inni będą mogli cieszyć się niezakłóconą dostępnością tak im potrzebnych cząsteczek.

Do czasu dokończenia systemu okaże się zapewne, że fasada zamieniła się w solidnego menedżera systemu efektów cząsteczkowych (to prawdopodobne zwłaszcza wtedy, kiedy już na początku miało się takie plany). Ale nawet jeśli nie, można szybko zwalić fasadę i odsłonić światu kompletny, nowy interfejs systemu; wymuszony tym przestój nie powinien być dłuższy niż kilka godzin.

Wzorzec Observer

Jednym z najczęstszych, a przy tym najtrudniejszych do wyśledzenia problemów towarzyszących programowaniu gry jest sytuacja, kiedy jakiś obiekt (`ObjectA`), do którego odnosi się inny obiekt (`ObjectB`), wychodzi poza zasięg i kończy żywot bez jakiegokolwiek powiadamiania o tym fakcie drugiego obiektu. Jeśli potem `ObjectB` będzie próbował skorzystać ze wskaźnika do `ObjectA`, sprowokuje w najlepszym wypadku zrzucenie wyjątku — adres będzie niepoprawny. Problem ilustruje kod z listingu 8.1.

Listing 8.1. *Dwie klasy powiązane zależnością posiadania*

```
// Prościutka klasa
class ObjectA
{
public:
    // Tu konstruktor i reszta
    void DoSomethingCool();
};

// Inna prościutka klasa odwołująca się do
// egzemplarza klasy ObjectA
class ObjectB
{
public:
    ObjectB(ObjectA* pA) : m_pObjectA(pA) {}
    void Update()
    {
        if (m_pObjectA != NULL)
            m_pObjectA->DoSomethingCool();
        }
    }

protected:
    ObjectA* m_pObjectA;      // nasz własny obiekt ObjectA
};

// Utworzenie obiektu klasy ObjectA
ObjectA* pA = new ObjectA();

// Utworzenie obiektu ObjectB, z adresem obiektu ObjectA
ObjectB* pB = new ObjectB(pA);

// Aktualizacja obiektu ObjectB
pB->Update();      // bez problemu!

// Usunięcie egzemplarza ObjectA
delete pA;

// Ponowna aktualizacja obiektu ObjectB
// wyłoży program
pB->Update();      // niedobrze...
```

Drugie wywołanie `p->Update()` zawodzi, ponieważ pamięć, w której poprzednio znajdował się obiekt wskazywany składową `m_pObjectA`, już go nie zawiera; obiekt został z niej usunięty.

Jedno z rozwiązań tego problemu wymaga, aby obiekt klasy `ObjectA` wiedział o tym, że odwołuje się do niego obiekt klasy `ObjectB`. Wtedy przy destrukcji obiektu mógłby on powiadomić obiekt `ObjectB` o fakcie usunięcia, pozwalając uwzględnić ten fakt w logice działania obiektu `ObjectB` (listing 8.2).

Listing 8.2. *Uzupełnienie procedury destrukcji o powiadomienie; likwidacja problemu wiszącego wskaźnika*

```
// Obiekty klasy ObjectB mogą być teraz powiadamiane
// o usunięciu obiektów wskazywanych klasy ObjectA
class ObjectB
{
public:
    // Tu wszystko inne...
    void NotifyObjectADestruction()
    {
        // Skoro wiemy, że m_pObjectA właśnie jest usuwany,
        // wypadałoby przynajmniej ustawić jego wskaźnik na NULL
        m_pObjectA = NULL;
    }
};

// Obiekty klasy ObjectA wiedzą teraz,
// że są posiadane przez obiekty ObjectB
class ObjectA
{
public:
    // Tu wszystko jak było...
    ~ObjectA()
    {
        m_pOwner->NotifyObjectADestruction();
    }
    void SetOwner(ObjectB* pOwner);

protected:
    ObjectB* m_pOwner;
};
```

Zwróćmy uwagę, że destruktor klasy `ObjectA` powiadamia teraz obiekt właściciela, że dany egzemplarz jest właśnie usuwany. W egzemplarzu `ObjectB` ustawia się wtedy wskaźnik obiektu posiadanego na `NULL`, co ma zapobiec przyszłym próbom wywoływania `DoSomethingCool()` (po uzupełnieniu funkcji `Update()` o odpowiedni sprawdzian wartości wskaźnika). Niby wszystko działa, ale można to zrobić odrobinę lepiej.

Wyobraźmy sobie, że klasa `ObjectB` ma robić coś ciekawszego, co również zależy od stanu posiadanego obiektu klasy `ObjectA`. Jeśli stan obiektu posiadanego ulegnie zmianie, „właściciel" tego obiektu powinien mieć to na uwadze i odpowiednio reagować. Skomplikujmy przykład jeszcze bardziej, wprowadzając do układu dwa nowe obiekty dwóch zupełnie różnych klas: `ObjectC` i `ObjectD`, które również powinny być w swoich działaniach świadome bieżącego stanu obiektu klasy `ObjectA`. Nasz prosty schemat posiadacz-posiadany przestanie się sprawdzać, bo rozbudowanie relacji pomiędzy klasami oznacza pojawienie się więcej posiadaczy. A naszym celem jest, aby obiekty klas `ObjectB`, `ObjectC` i `ObjectD` mogły być świadkami zmian zachodzących w obiektach i mogły wykorzystywać swoje obserwacje do własnych celów.

Rozwiązaniem jest następny wzorzec — **Observer**, czyli obserwator. Obserwator to obiekt, którego istotą jest możliwość obserwowania zmian zachodzących w jakimś innym obiekcie, określanym mianem **przedmiotu obserwacji** (ang. *subject*). Ten ostatni może być obserwowany przez dowolną liczbę obserwatorów i wszystkich ich powiadamia o ewentualnych zmianach. Do wdrożenia wzorca Observer przygotujemy się poprzez implementacje obu klas wzorca (listing 8.3).

Listing 8.3. *Proste klasy obserwatora i przedmiotu obserwacji*

```
// Bazowa klasa obserwatora
class Observer
{
public:
    virtual ~Observer();
    virtual void Update() = 0;    // klasa abstrakcyjna...
    void SetSubject(Subject* pSub);

protected:
    Subject* m_pSubject;
};

// Bazowa klasa przedmiotu obserwacji
class Subject
{
public:
    virtual ~Subject()
    {
        std::list<Observer*>::iterator iter;
        for (iter = m_observers.begin();
             iter != m_observers.end();
             iter++)
        {
            // Powiadomienie obserwatorów o destrukcji
            (*iter)->SetSubject(NULL);
        }
    }

    virtual void AddObserver(Observer* pObserver)
    {
        m_observers.push_back(pObserver);
    }

    virtual void UpdateObservers()
    {
        std::list<Observer*>::iterator iter;
        for (iter = m_observers.begin();
             iter != m_observers.end();
             iter++)
        {
            (*iter)->Update();
        }
    }

protected:
    std::list<Observer*> m_observers;
};
```

Klasa bazowa `Observer` jest całkiem prosta: jedyne, co zawiera, to wskaźnik przedmiotu obserwacji. Nieco bardziej rozbudowana jest klasa bazowa samego obiektu obserwacji. Posiada ona listę obserwatorów (wskaźników obiektów obserwatorów) i środek do powiadamiania tych obserwatorów o zmianach swojego stanu w postaci metody `UpdateObservers()`. Dodatkowo obiekt przedmiotu obserwacji „powiadamia” swoich

obserwatorów o fakcie własnej destrukcji — powiadomienie odbywa się przez ustawienie wskaźnika przedmiotu obserwacji u obserwatora na NULL (ale to tylko przykładowy sposób powiadomienia, można je zrealizować zupełnie dowolnie).

Weźmy teraz na warsztat konkretny przykład wdrożenia wzorca Observer. Załóżmy, że mamy w naszej grze zaimplementować jednostkę wyrzutni rakiet. Po odpaleniu wyrzutnia ma wyrzucić rakietę, która będzie za sobą ciągnęła smugę wizualizowaną za pomocą efektów cząsteczkowych; odpalenie ma też być wizualizowane za pomocą dynamicznego oświetlenia otoczenia rakiety (odbicia blasku dyszy); oczywiście odpaleniu musi też towarzyszyć odpowiednio przerażający dźwięk. Wszystkie te trzy efekty muszą być aktualizowane w miarę zmiany stanu rakiety, do tego odpowiednią reakcję powinno wywołać również uderzenie rakiety w jakiś obiekt w świecie gry. Z klas bazowych z listingu 8.3 możemy wyprowadzić klasy pochodne i skorzystać w nich z dobrodziejstw wzorca Observer w implementacji naszej wyrzutni (listing 8.4).

Listing 8.4. *Wzorzec Observer w akcji*

```
// Nasza prosta rakieta
class Rocket
{
public:
    Rocket();
    float GetSpeed();
    float GetFuel();
    void Update(float fDeltaT)
    {
        // Konieczne aktualizacje rakiety;
        // Powiadomienie obserwatorów, żeby też mogli
        // przeprowadzić aktualizacje
        UpdateObservers();
    }
};

// Klasa efektów cząsteczkowych dla smugi za rakietą
class RocketFlames : public PrticleSystem, public Observer
{
public:
    RocketFlames(Rocket* pRocket)
    {
        m_pRocket = pRocket;
        pRocket->AddObserver(this);
    }
    virtual void Update(
    {
        // Tu jakieś fajne efekty zmiany smugi, np.
        // zależnie od ilości pozostałego paliwa
        // rakietowego
        float fFuelRemaining = m_pRocket->GetFuel();
        // ...
    }
protected:
    Rocket* m_pRocket;
};

// Klasa oświetlenia dynamicznego od żaru silnika rakiety
class RocketLight : public DynamicLight, public Observer {
```

```
public:
    RocketLight (Rocket* pRocket)
    {
        m_pRocket = pRocket;
        pRocket->AddObserver(this);
    }
    virtual void Update()
    {
        // Dostosowanie jasności i koloru oświetlenia do
        // ilości pozostałego paliwa rakietowego
        float fFuelRemaining = m_pRocket->GetFuel();
        // ...
    }
protected:
    Rocket* m_pRocket;
};

// Klasa efektu dźwiękowego huku odpalenia i świstu lotu rakiety
class RocketWhistle : public Sound3D, public Observer {
public:
    RocketWhistle(Rocket* pRocket)
    {
        m_pRocket = pRocket;
        pRocket->AddObserver(this);
    }
    virtual void Update()
    {
        // Dostosowanie głośności i brzmienia świstu do prędkości rakiety
        float fSpeed = m_pRocket->GetSpeed();
        // ...
    }
protected:
    Rocket* m_pRocket;
};
```

Za pomocą klas z listingu 8.4 (oczywiście odpowiednio uzupełnionych o niezbędne szczegóły implementacji) można tworzyć obiekty rakiet i kojarzyć z nimi wielorakie efekty, a także automatyzować aktualizację tych efektów przy aktualizacji stanu samej rakiety, aby w końcu również automatycznie powiadomić klasy efektów o usunięciu obiektu rakiety. W naszej obecnej implementacji obserwatorzy (obiekty efektów) mogą elegancko obsłużyć u siebie zdarzenie usunięcia rakiety, choć sami będą trwać dalej. Moglibyśmy pewnie wymusić autodestrukcję każdego z efektów towarzyszących, albo wymusić taką destrukcję na jakimś menedżerze efektów. Tak czy inaczej byłoby to efektywne i proste do zaprogramowania, a o to głównie chodziło.

Wzorzec Visitor

Mało jest zadań, które bardziej irytowałyby programistę gier niż implementowanie czegoś tylko po to, żeby potem powielać tę samą funkcjonalność w wielu miejscach programu. A tak bywa dość często, zwłaszcza kiedy klasy są kompletnie niepowiązane ze sobą, ale mają podobne i w podobny sposób wykorzystywane cechy i funkcje.

Klasycznym przykładem takiego czegoś są zapytania pozycyjne wykonywane celem odnalezienia rozmieszczonych w świecie gry obiektów różnych typów. Oto kilka przykładów takich zapytań:

- Wyszukanie wszystkich przeciwników znajdujących się w określonej odległości od gracza.
- Wyszukanie wszystkich apteczek rozmieszczonych powyżej pewnej odległości od gracza.
- Zliczenie liczby elementów w stożku pola widzenia AI.
- Zlokalizowanie wszystkich punktów teleportacyjnych poniżej poziomu gracza.
- Wyliczenie zbioru wszystkich punktów nawigacyjnych, w pobliżu których znajduje się choćby jeden przeciwnik.
- Wyszukanie wszystkich dynamicznych świateł w polu widzenia gracza.
- Wyszukanie pięciu źródeł dźwięku znajdujących się najbliżej gracza.

Listę można by kontynuować dość długo i powstałaby lista typowych czynności, realizowanych w praktycznie każdej grze, w jej rozmaitych podsystemach i systemach: w AI, w grafice, w dźwięku, w wyzwalaczach akcji i tak dalej. A co gorsza, niekiedy zapytania pozycyjne bywają wielce specyficzne. Weźmy choćby dość skomplikowany wyzwalacz, który korzysta z testu zbliżeniowego, połączonego z testem znajdowania się w polu widzenia i do tego z pewnym progiem minimalnego dystansu aktywacji. Kto nie musiał czegoś takiego implementować, jest szczęśliwym człowiekiem.

A kto **musiał**, możliwe że nieszczęśliwie dopracował się takiej implementacji:

```
// Punkt nawigacyjny
class Waypoint
{
public:
    // Tu rzeczy typowe dla p. nawigacyjnego...
    // Podaj położenie
    const Vector3D& GetLocation();
};

// Dźwięk 3d, z pewnymi elementami upodabniającymi do punktu nawigacyjnego
// ale poza tym zupełnie z nim niepowiązany
class Sound3D
{
public:
    // Tu typowe elementy klasy dźwiękowej...
    // Podaj położenie
    const SoundPosition& GetPos();
};

// I dziupla przeciwników...
class SpawnPoint
{
public:
    // Tu typowe elementy...
    // Podaj pozycję — tym razem na płaszczyźnie!
   const Vector2D& GetSpawnLocation();
};
```

Każda z tych klas udostępnia dość podobną publiczną funkcję akcesora do danych o położeniu obiektu, ale każda z tych funkcji zwraca wartość innego typu (`Vector2D` jest inny od `Vector3D` i tak dalej), różne są też nazwy akcesorów. Z powodu tych różnic w kodzie gry można się spodziewać czegoś takiego:

```
// Menedżer punktów nawigacyjnych, ze
// stosownymi zapytaniami pozycyjnymi
class WaypointManager
{
public:
    Waypoint* FindNearestWaypoint(const Vector3D& pos);
    Waypoint* FindNearestWaypointInRange(const Vector3D& pos,
                  float fInnerRange);
    Waypoint* FindNearestWaypointOutsideRange(float fOuterRange);
    Waypoint* FindWaypointsInCone(const Vector3D& pos,
        const Vector3D& dir);

    // itd., itd.
};
```

Jeśli poszukamy w kodzie dłużej, znajdziemy całkiem niezależny podprojekt z klasą `LightManager`, która robi praktycznie to samo, ale operuje nie na punktach nawigacyjnych, tylko na światłach; podobne implementacje menedżerów znajdziemy też dla agentów AI (`AIManager`), dla dźwięków (`SoundManager`) i tak dalej.

Jeśli to dopiero początek projektu, można uniknąć powielania kodu przez wyprowadzenie punktów nawigacyjnych, świateł, dźwięków i czego tam jeszcze ze wspólnej klasy bazowej, reprezentującej pozycję i orientację. Ale zazwyczaj smutna prawda jest taka, że powielanie zauważa się dopiero wtedy, kiedy na tak dramatyczną zmianę interfejsu jest za późno, a sama zmiana okazuje się zbyt ryzykowna.

Wtedy rozwiązaniem jest **wzorzec Visitor** — wizytator. Wizytator pozwala na efektywne rozszerzanie kolekcji zazwyczaj niepowiązanych ze sobą klas bez ingerowania w interfejsy tych klas. Zazwyczaj odbywa się to przez utworzenie nowego zestawu funkcji operujących na interfejsach różnych klas w celu pozyskania z tych klas podobnych informacji. W tym przypadku każda z naszych hipotetycznych klas (punkt nawigacyjny, światło, dźwięk itd.) posiada pozycję (ewentualnie również orientację) i możemy napisać zestaw zapytań pozycyjnych, które będą operowały na obiektach dowolnej z tych klas.

Implementacja wzorca Visitor składa się z dwóch zasadniczych komponentów:

1. Klasy wizytatora, równoległej do każdej z klas, którą chcemy objąć wizytacją, a udostępniającej mechanizm wizytacji. Alternatywnie można ustanowić jednego wizytatora dla każdej kategorii wspólnych funkcji czy elementów, operującego na całym zestawie różnych klas.
2. Pojedynczej metody w każdej wizytowanej klasie; niech będzie to `AcceptVisitor()` i niech pozwala korzystać wizytatorowi z publicznego interfejsu klasy.

Wróćmy do naszego przykładu i przyjrzyjmy się prostej klasie wykorzystywanej do wizytowania obiektów różnych typów, z których każdy w jakiś sposób reprezentuje pozycję:

```
class PositionedPbjectVisitor
{
public:
    PositionedObjectVisitor();
    virtual ~PositionedObjectVisitor();

    // Jawne metody wizytacji dla każdego wizytowanego
    // typu obiektów
    virtual void VisitWaypoint(Waypoint* pWaypoint);
    virtual void VisitSound3D(Sound3D* pSound3D);
    virtual void VisitSpawnPoint(SpawnPoint* pSpawnPoint);
};
```

Zgodnie z wymogiem z drugiego punktu każda z wizytowanych klas musi posiadać metodę `AcceptVisitor()`:

```
void Waypoint::AcceptVisitor(PositionedObjectVisitor & viz)
{
    viz.VisitWaypoint(this);
}

void Sound3D::AcceptVisitor(PositionedObjectVisitor & viz)
{
    viz.VisitSound3D(this);
}

void SpawnPoint::AcceptVisitor(PositionedObjectVisitor & viz)
{
    viz.VisitSpawnPoint(this);
}
```

Odpowiednie metody wizytatora powinny wyglądać tak:

```
void PositionedObjectVisitor::VisitWaypoint(Waypoint* pWaypoint)
{
    const Vector3D& waypointPos = pWaypoint->GetLocation();
    // Operacje na odczytanym położeniu punktu nawigacyjnego
}

void PositionedObjectVisitor::VisitWaypoint(Sound3D* pSound3D)
{
    const SoundPosition& soundPos = pSound3D->GetPos();
    // Operacje na odczytanym położeniu źródła dźwięku
}

void PositionedObjectVisitor::VisitSpawnPoint(SpawnPoint* pSpawnPoint)
{
    const Vector2D& spawnPos = pSpawnPoint->GetSpawnLocation();
    // Operacje na odczytanym położeniu dziupli
}
```

Teraz mamy wizytatora dla każdego obiektu pozycjonowanego; klasy takich obiektów uzupełniliśmy o metodę `AcceptVisitor()` i od tego momentu możemy uzupełniać klasę wizytatora o wszystkie funkcje związane z określaniem pozycji różnych obiektów. Gdybyśmy mieli publiczny dostęp do jakiegoś kontenera takich obiektów zdatnych do wizytacji, moglibyśmy za pomocą wizytatora przeglądać całe kontenery i wybierać z nich informacje będące celem wizytacji z możliwością bieżącego przetwarzania i klasyfikowania tych informacji, na przykład zapamiętywania najbliższego znalezionego do tej pory punktu nawigacyjnego. Każde z zapytań pozycyjnych może działać w ten sposób i możemy nawet stosować je z kombinacjami obiektów różnych typów, choćby w celu wyszukania najbliższej dziupli (ang. *spawn point*).

Wnioski

Wzorce omawiane w tym rozdziale nijak nie wyczerpują zestawu wzorców projektowych, które opisano w literaturze, ale zostały wybrane jako dobrze reprezentujące wzorce pojawiające się typowo w programowaniu gier komputerowych. Chyba najbardziej powszechny i znajomy większości programistów jest Singleton, często nawet intuicyjnie wykorzystywany do implementowania klas przewidzianych do tworzenia pojedynczych, ale globalnie dostępnych egzemplarzy zarządzających zasobami. Z kolei wzorzec Façade świetnie nadaje się do eksponowania funkcjonalności systemu jego użytkownikom (systemom zależnym), zanim jeszcze system zacznie w pełni działać i bez wymuszania przestojów w tworzeniu tychże podsystemów zależnych. Wcielenie wzorca Observer udostępnia mechanizm umożliwiający obiektom automatyczne aktualizowanie swojego stanu w reakcji na zmiany stanu jakiegoś innego obiektu, od którego jakoś zależą. Wreszcie wzorzec Visitor powala na rozszerzenie funkcjonalności klasy albo zestawu klas przy możliwie małej ingerencji w ich interfejsy.

W rozdziale celowo — dla utrzymania zwartości omówienia — pominąłem szereg szczegółowych aspektów poszczególnych wzorców. Szczegóły te mogą okazać się istotne w niektórych przypadkach. Dlatego zachęcam gorąco do samodzielnych studiów nad wzorcami projektowymi, począwszy od zapoznania się z pozycjami wymienionymi w podrozdziale „Zalecana lektura” — można tam znaleźć nie tylko doprecyzowanie omawianych wzorców i ich zalety oraz ewentualne wady, ale też mnóstwo wzorców innych, tutaj przemilczanych.

Zalecana lektura

Poniższe pozycje zawierają szerokie i szczegółowe zarazem omówienie okazałego zestawu wzorców projektowych, które mogą okazać się przydatne również w projektach gier komputerowych. W rozdziale starałem się przedstawić krótko cztery najpopularniejsze wzorce, pomijając jednak co subtelniejsze kwestie dotyczące ich implementacji

i stojącej za nimi teorii. Dlatego warto uzupełnić wiedzę w drodze samodzielnych studiów choćby po to, żeby odkryć tam kwestie wypływające przy implementacjach wzorców niewłaściwych dla niuansów danego problemu.

Erich Gamma et al., *Design Patterns*, wydawnictwo Addison-Wesley 1995.

Meyers Scott, *More Effective C++. 35 New Ways to Improve Your Programs and Designs*, wydawnictwo Addison-Wesley Professional 1995.