Wizytator

upraszczanie zależności przy modyfikacji interfejsu klas

Operacje dla obiektów w hierarchii klas często implementujemy, wykorzystując funkcje wirtualne. Gdy liczba takich metod rośnie, klasy mają trudną do określenia odpowiedzialność, kod staje się mało przejrzysty. Przedstawiona technika rozwiązuje ten problem.

Dowiesz się:
- Co to jest wzorzec wizytatora;
- Jak używać biblioteki boost::variant.

Powinieneś wiedzieć:
- Jak pisać proste programy w C++;
- Co to jest dziedziczenie i funkcje wirtualne.

Poziom trudności

Dodawanie nowej klasy do hierarchii polega na utworzeniu tej klasy oraz nadpisaniu metod. Modyfikujemy jedynie fragment kodu źródłowego zawierający implementację nowej klasy, najczęściej tworzymy nowy plik. Dodawanie metody, która będzie nadpisywana, jest bardziej skomplikowane: modyfikujemy klasę bazową oraz wszystkie klasy pochodne, dostarczając odpowiednią funkcjonalność. W tym przypadku modyfikacja obejmuje wiele różnych fragmentów kodu. Taka asymetria pomiędzy modyfikacją hierarchii klas a modyfikacją interfejsu w tej hierarchii jest niewygodna, zwłaszcza gdy częściej będziemy modyfikować funkcjonalność niż strukturę. Artykuł przedstawia wzorzec projektowy wizytatora (inna nazwa to odwiedzający), który pozwala uprościć zależności przy modyfikacji funkcji operujących na hierarchii klas. W dalszej części artykułu przedstawiona zostanie zasada działania tego wzorca, przykład użycia oraz wykorzystanie wizytatora w bibliotece boost::variant.

Niedogodności nadpisywania metod

Wzorzec projektowy wizytatora (inna nazwa to odwiedzający), opisany między innymi w książce Gamma, Helm, Johnson, Vlissides „Wzorce projektowe", pozwala sprawić, że modyfikacja funkcjonalności będzie prosta, natomiast złożona będzie modyfikacja struktury. Przykład ilustrujący omawianą technikę wykorzystuje hierarchię klas `Unit`, przedstawioną na Rysunku 1, reprezentującą różne oddziały wykorzystywane w grze strategicznej: jednostki piechoty, czołgi, wyrzutnie rakiet itd. Podczas tworzenia kolejnych wersji gry hierarchii tej prawie nie będziemy zmieniać, typy jednostek będą ustalone we wczesnych etapach implementacji. Spodziewamy się, że będą zmieniane funkcje operujące na jednostkach lub grupach jednostek. Funkcje te będą obliczały wartość bojową jednostek, ich szybkość przemieszczania, będą automatycznie rozmieszczać jednostki na danym obszarze, obrazować te jednostki, generować statystyki itp. Zbiór tych funkcji będziemy rozszerzali podczas tworzenia kolejnych wersji aplikacji.

Gdyby funkcje operujące na jednostkach implementować jako metody klas, tak jak pokazano na Listingu 1, to wystąpi niedogodność, o której była mowa na początku, dodawanie lub usuwanie metody wymaga modyfikacji wszystkich klas w hierarchii. Przy takim podejściu kod dotyczący tej samej funkcji będzie rozproszony, każda klasa będzie miała fragment funkcjonalności, zmniejsza to czytelność kodu i utrudnia jego modyfikacje. Przykładowo, obliczanie statystyk (liczby żołnierzy, liczby czołgów itd.) wymaga, oprócz klasy przechowującej liczniki, dostarczenia metody w każdej klasie konkretnej, która aktualizuje te liczniki (Listing 1). Dodatkowo klasy reprezentujące jednostki będą miały wiele różnych metod, trudno będzie określić ich odpowiedzialność.

Kod źródłowy będzie bardziej przejrzysty, jeżeli daną funkcję zrealizujemy w spójnym fragmencie kodu. Dla rozpatrywanego przykładu możemy aktualizację liczników przenieść do klasy, która je zawiera (patrz Listing 2). Klasy reprezentujące jednostki będą prostsze, nie muszą zawierać metod związanych z obliczaniem statystyk. Kod realizujący daną funkcję jest umieszczony w jednym miejscu, np. kod obliczania statystyk zawiera klasa `Statistics` (Listing 2). Niestety przedstawiona technika wymaga jawnego badania typu obiektu za pomocą mechanizmów refleksji, co jest dosyć czasochłonne. Łańcuch instrukcji warunkowych, który porównuje typ obiektu ze wszystkimi typami w hierarchii, nie jest eleganckim rozwiązaniem.

Wzorzec wizytatora pozwala zachować zalety wynikające z umieszczenia kodu realizującego daną funkcję w jednym miejscu (poza klasami w hierarchii), usuwając konieczność badania typu obiektu za pomocą łańcucha instrukcji dynamic_cast.

Szybki start

Aby uruchomić przedstawione przykłady, należy mieć dostęp do kompilatora C++ oraz edytora tekstu. Niektóre przykłady korzystają z udogodnień dostarczanych przez bibliotekę boost::variant, warunkiem ich uruchomienia jest instalacja bibliotek boost (w wersji 1.36 lub nowszej) Na wydrukach pominięto dołączanie odpowiednich nagłówków oraz udostępnianie przestrzeni nazw, pełne źródła dołączono jako materiały pomocnicze.

Wzorzec wizytatora

Wzorzec wizytatora wykorzystuje pomocniczą hierarchię klas, zwaną hierarchią wizytującą lub odwiedzającą, która jest związana z hierarchią klas, dla której chcemy odwiedzać (wizytować) obiekty. Klasa bazowa tej nowej hierarchii, abstrakcyjny wizytator, dostarcza metod, które będą wołane dla poszczególnych obiektów klas hierarchii odwiedzanej. Dla omawianego przykładu abstrakcyjny wizytator został przedstawiony na Listingu 3.

Dla każdej klasy odwiedzanej (wizytowanej) dostarczamy metodę `accept`, która woła odpowiednią metodę wizytatora. Metoda `accept` jest wykorzystywana przez wizytator do uzyskania informacji o typie obiektu. Modyfikacja hierarchii odwiedzanej została przedstawiona na Listingu 4. Dla wizytatora, który jest argumentem, wołana jest jedna z przeciążonych metod `visit`, w zależności od typu obiektu, który nadpisuje metodę `accept`.

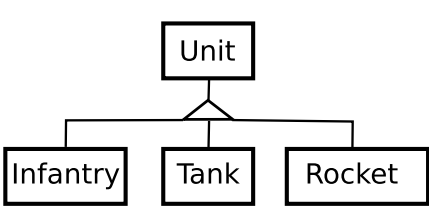
Operacje na hierarchii odwiedzanej, na przykład zbieranie statystyk, implementujemy, wykorzystując klasę pochodną po abstrakcyjnym wizytatorze (Listing 5). Technika ta pozwala na uzyskanie typu obiektu odwiedzanego bez rzutowania dynamicznego, wykorzystujemy dwukrotnie funkcje wirtualne. Innymi słowy, metody `visit` dostają jako argument obiekt odpowiedniego typu, wybór tego typu odbywa się poprzez mechanizm późnego wiązania, użyty przy wyborze odpowiedniej metody `accept oraz przy` wołaniu metody `visit`.

Przykład użycia wizytatora zawiera Listing 6. Metoda `accept` jest wołana dla obiektu `u`, który jest typu Tank (ale wskaźnik jest typu `Unit*`). Metoda ta będzie wołała metodę `visit(Tank&)` dla przekazanego argumentu, czyli dla obiektu `s`.

Wzorzec wizytatora stosuje się po to, aby ułatwić dodawanie nowych operacji dla hierarchii klas, oraz po to, aby metody wykonujące daną funkcję umieścić w tym samym miejscu. Dodanie nowego wizytatora nie wymaga wiele zmian, należy wyprowadzić nową klasę z klasy `Visitor`, a następnie nadpisać w niej odpowiednie metody. Inne klasy nie są zmieniane, w szczególności nie są zmieniane klasy w hierarchii wizytującej.

boost::variant

Wzorzec wizytatora jest wykorzystywany w szablonie klasy `boost::variant` do implementacji operacji na przechowywanym obiekcie. Szablon `variant` tworzy typ złożony, który jest równoważny unii z C (definiowanej przez `union`) lub rekordowi z wariantami z Pascala. Obiekt typu `variant` może przechowywać jeden z obiektów składowych, wielkość (zajętość pamięci) jest zależna od wielkości największego obiektu składowego. Typ ten posiada semantykę wartości, to znaczy konstruktor kopiujący, operator przypisania, można go stosować jako argument funkcji, zwracać jako wartość, prze-



Rysunek 1.

Listing 1. *Przykład funkcji, która wymaga modyfikacji wszystkich elementów w hierarchii*

```
class Statistics { //Klasa reprezentuje statystyki
public:
   void addSoldiers(int n){ soldiers_ += n; }; //aktualizuje licznik
   void addTanks(int n){ tanks_ += p; }
   void addRockets(int n) { rockets_ += n; }
private:
   int soldiers_; //licznik żołnierzy
   int tanks_;
   int rockets_;
};
class Unit { //klasa bazowa
public: //zawiera interfejs do metody aktualizującej statystyki
   virtual void update(Statistics& s) = 0;
};
void Infantry::update(Statistics& s)//aktualizuje statystyki
   s.addSoldiers( countSoldiers() );
}
void Tank::update(Statistics& s) {
   s.addTanks(1);
}
```

Listing 2. *Odwrócenie zależności pozwala zgrupować kod aktualizujący w jednym miejscu, jednak wymaga jawnego badania typu obiektów*

```
void Statistics::update(const Unit& unit) {
   if(const Infantry* p = dynamic_cast<const Infantry*>(&unit) ) { //jawne badanie
                        typu
      soldiers_ += p->countSoldiers();
   } else if (dynamic_cast<const Tank*>(&unit) {
      ++tanks_;
   } // łańcuch warunków dla wszystkich typów w hierarchii
}
```

Listing 3. *Abstrakcyjny wizytator dla jednostek z omawianej gry strategicznej*

```
class Visitor { //abstrakcyjny wizytator
public:
   virtual void visit(Infantry&) = 0;
   virtual void visit(Tank&) = 0;
   virtual void visit(Rocket&) = 0;
};
```

Listing 4. *Metoda w hierarchii odwiedzanej wykorzystywana przez wizytatory*

```
class Unit { //Klasa bazowa
public:
   virtual void accept(Visitor& v) = 0;
};
class Infantry : public Unit {
public:
   virtual void accept(Visitor& v) { v.visit(*this); } //woła v.visit(Infantry&)
};
class Tank : public Unit {
public:
   virtual void accept(Visitor& v) { v.visit(*this); } //woła v.visit(Tank&)
};
```

chowywać w kontenerach standardowych. Typami składowymi mogą być klasy dostarczające konstruktorów, co jest zabronione przy używaniu unii w C++. Przykłady wykorzystania wariantów pokazano na wydruku 7.

Dostęp do przechowywanych wartości wariantu jest możliwy poprzez wizytator. Musimy dostarczyć konkretnego wizytatora dziedziczącego po szablonie `static_visitor`(parametrem tego szablonu jest typ zwracany z metod wizytujących, możemy zwracać wartość w metodzie wizytującej). Metody wizytujące implementuje się jako przeciążone operatory wołania funkcyjnego, zamiast metod `visit`, co jest częstą praktyką przy implementacji tego wzorca w C++. Wizytację uruchamia funkcja `apply_visitor`, do której przekazujemy obiekt wizytatora i wariant, patrz Listing 8.

Listing 5. *Klasa obliczająca statystyki jako wizytator*

```
class Statistics : public Visitor { //wizytator konkretny
public:
   virtual void visit(Infantry& u) {//wołane dla jednostek piechoty
      soldiers_ += u.countSoldiers();
   }
   virtual void visit(Tank& t) {//wołane dla czołgów
      ++tanks_;
   }
   /* ... */
    };
```

Listing 6. *Przykład użycia wizytatora*

```
Unit* u = new Tank(); //dostarcza jednostkę
Statistics s; //tworzy obiekt statystyk
s.accept(u); //dwukrotnie wykorzystuje funkcje wirtualne
```

Listing 7. *Wariant, przykłady użycia*

```
//obiekt, który może przechowywać jeden z trzech typów
variant<int, double, std::string> var;//obiekt przechowujący int
var = "Hej";//teraz przechowuje napis
var = 2.7; //teraz przechowuje wartość typu double
//przykładowa deklaracja funkcji, która ma argument typu variant
void function(const variant<int, double, std::string>& v);
function(var); //przekazuje jako argument do funkcji
```

Listing 8. *Dostęp do wartości przechowywanych w wariancie*

```
typedef variant<int, double, string> Var;
class MyVisitor //dostęp za pomocą wizytatora
   : public boost::static_visitor<void> { //metody visit zwracają void
public:
   void operator()(int& i) { /* metoda dla obiektu typu int */ }
   void operator()(double& d) { /* ... */ }
   void operator()(string& s) { /* ... */ }
};
apply_visitor(MyVisitor, var); //wizytacja obiektu var
```

Więcej w książce

Zagadnienia dotyczące współcześnie stosowanych technik w języku C++, wzorce projektowe, programowanie generyczne, prawidłowe zarządzanie zasobami przy stosowaniu wyjątków, programowanie wielowątkowe, ilustrowane przykładami stosowanymi w bibliotece standardowej i bibliotekach boost, zostały opisane w książce „Średnio zaawansowane programowanie w C++", która ukaże się niebawem.

W Sieci

- *http://www.boost.org* – dokumentacja bibliotek boost;
- *http://www.open-std.org* – dokumenty opisujące nowy standard C++.

Wariant, czyli unia z kontrolą typów i wyróżnikiem bieżącego typu, korzysta z programowania generycznego do bezpiecznego przechowywania obiektów w tym samym miejscu pamięci. Narzuty pamięciowe są małe, związane jedynie z wyróżnikiem aktualnego typu (obecnie 4 bajty na platformach wspieranych przez boost). Wewnętrzny bufor ma wielkość maksymalnego obiektu, który może być przechowywany w wariancie (uwzględniając wyrównanie).

Podsumowanie

Wizytator pozwala implementować funkcje operujące na hierarchii klas jako oddzielne typy, co upraszcza zależności w aplikacji i pozwala na tworzenie przejrzystego kodu. Oprócz wielu zalet wizytator ma kilka wad.

Operacje na hierarchii odwiedzanej, implementowane w wizytatorze, wykorzystują interfejs klas odwiedzanych, więc muszą istnieć odpowiednie metody, które daną operację pozwolą wykonać. Może to prowadzić do złamania enkapsulacji, na przykład jeżeli do realizacji funkcji implementowanych w wizytatorze wymagana jest znajomość wewnętrznego stanu obiektów odwiedzanych.

Wizytator wprowadza cykliczne zależności pomiędzy hierarchią odwiedzaną i odwiedzającą, które sprawiają, że implementacja tego wzorca wymaga użycia deklaracji klas. Klasa bazowa hierarchii odwiedzanej(klasa `Unit`) jest zależna od deklaracji klasy bazowej hierarchii klas wizytujących (klasa `Visitor`), ponieważ zawiera deklarację metody `accept`. Abstrakcyjny wizytator jest zależny od deklaracji wszystkich klas konkretnych w hierarchii odwiedzanej, a więc od klas `Infantry`, `Tank`, `Rocket`. Klasy konkretne w hierarchii odwiedzanej zależne są od klasy bazowej, dziedziczą po niej. Wprowadzając dodatkowe klasy i wykorzystując dziedziczenie wielobazowe, możemy pozbyć się zależności cyklicznych w tym wzorcu. Taką implementację opisano w książce „Średnio zaawansowane programowanie w C++" (patrz ramka).

Wizytator dostarcza mechanizmu równoważnego dodawaniu metod do klas. Dostarcza on mechanizmu wyboru odpowiedniej metody w zależności od dwóch typów, jednym jest typ obiektu odwiedzanego, drugim typ wizytatora. Rozszerzeniem tej techniki są wielometody, które będą omówione w jednym z kolejnych artykułów.

ROBERT NOWAK

Adiunkt w Zakładzie Sztucznej Inteligencji Instytutu Systemów Elektronicznych Politechniki Warszawskiej, zainteresowany tworzeniem aplikacji bioinformatycznych oraz zarządzania ryzykiem. Programuje w C++ od ponad 10 lat.
Kontakt z autorem:rno@o2.pl