Komendy

Żądania jako obiekty

Reprezentacja żądania, czyli wołania funkcji lub metody przez obiekt, pozwala na rozdzielenie momentu tworzenia żądania od momentu jego zgłoszenia, co nazywamy opóźnionym wołaniem. Za pomocą przedstawionych technik można rozdzielić moduły, które tworzą żądania, od modułów, które wołają zawarte w nich funkcje lub metody, co pozwala redukować zależności w aplikacji.

Dowiesz się:
- Co to jest wzorzec komendy;
- Co to jest funktor;
- Jak manipulować funktorami za pomocą boost::function;
- Jak tworzyć funktory za pomocą boost::bind.

Powinieneś wiedzieć:
- Jak pisać proste programy w C++;
- Co to są szablony.

Poziom trudności

Wołając funkcję lub metodę, wysyłamy żądanie wykonania pewnej akcji. Aby wołać, musimy znać interfejs modułu zawierającego daną funkcję lub interfejs klasy zawierającej daną metodę, następnie dostarczamy argumenty odpowiednich typów. Przechowując odniesienie (na przykład wskaźnik) do funkcji lub metody wewnątrz obiektu, mamy możliwość opóźnionego uruchomienia wskazywanego kodu, co jest istotne, gdy chcemy, aby inne moduły tworzyły żądania, wiązały obiekt z metodą, która będzie dla tego obiektu wołana, oraz z argumentami, a inne inicjowały odpowiednie akcje. Obiekty przechowujące odnośnik do funkcji lub metody oraz argumenty nazywamy komendami, nazwę tę stosuje się także do wzorca projektowego opisanego przez „*bandę czworga*" (Gamma, Helm, Johnson, Vlissides) w książce „*Wzorce projektowe*", (WNT, 2005). Biblioteki boost dostarczają udogodnień, które wspierają tworzenie i manipulowanie tego typu obiektami w języku C++.

Komenda, czyli żądanie jako obiekt

Żądanie wykonania pewnej akcji w języku C++ implementuje się jako wołanie funkcji lub wołanie metody. Takie rozwiązanie zakłada znajomość odpowiednich interfejsów przez kod tworzący żądanie i kod je zgłaszający, przy wołaniu funkcji lub metody te dwie czynności są tożsame, nierozdzielne. Na Listingu 1 pokazano wołanie przykładowej metody `move` dla obiektu typu `Window`, aby wywołać tę metodę, musimy znać interfejs klasy `Window`, mieć dostęp do obiektu tej klasy i dostarczyć odpowiednich argumentów. Komenda pozwala rozdzielić akt dostarczania argumentów i wiązania ich z obiektem i nazwą metody, co nazywa się tworzeniem komendy, od aktu wołania żądania. Inne moduły mogą tworzyć komendy, a inne je wołać, przy czym wołanie nie wymaga znajomości szczegółów, nie wymaga podawania argumentów, nie wymaga znajomości interfejsu klas, które będą realizować żądanie. Na Listingu 1 pokazano to na przykładzie komendy `CmdMove`, która ma zaszytą w kodzie nazwę metody oraz typ obiektu, do którego będzie kierowane żądanie. W czasie tworzenia tej komendy dostarczamy obiekt, dla którego będzie wołana metoda `move` oraz argumenty dla tej metody, nazwa wołanej metody oraz typ obiektu, który obsłuży żądanie, są zaszyte w kodzie tej komendy.

Za pomocą komend możemy zmniejszyć liczbę powiązań w systemie, na przykład obiekty interfejsu użytkownika mogą być inicjowane odpowiednimi komendami, nie muszą wtedy znać szczegółów ich tworzenia i działania, po wystąpieniu odpowiedniego zdarzenia uruchamiają dostarczone komendy, patrz Rysunek 1. Koszt tego rozwiązania to wprowadzenie nowych klas, które reprezentują komendy, przechowywanie obiektów tych klas (zazwyczaj niewielkich, ale występujących w dużych ilościach), wydłużony czas zgłaszania żądania, funkcja lub metoda jest wołana pośrednio.

Komenda może dostarczać semantyki wartości, a wtedy możemy ją kopiować, zwracać jako wynik działania, przechowywać w kontenerach, dostarczać jako argument. Przykładem wykorzystania tych cech jest utworzenie kolekcji, która przechowuje kolejno wykonywane komendy, rejestrując żądania użytkownika (tworząc historię). Kolekcja taka pozwala na ponawianie żądań (`Redo`), wystarczy kolejny raz uruchomić komendę. Jeżeli dla komendy zdefiniuje się metodę, która po wywołaniu wykona akcję odwrotną (akcję, która sprawi, że stan aplikacji będzie taki, jak przed wykonaniem komendy, co możemy jednoznacznie wykonać gdy mamy metodę i jej argumenty, na przykład dla dodawania będzie to odejmowanie), to dysponując kolekcją kolejno wykonanych komend, można dostarczyć funkcje wycofywania żądań (`Undo`), wołamy dla kolejnych komend akcje odwrotne.

Funktory, boost::function

Komendy można podzielić na takie, które tylko przekierowują sterowanie, to znaczy wołają istniejące funkcje i metody, oraz na takie,

Szybki start

Aby uruchomić przedstawione przykłady, należy mieć dostęp do kompilatora C++ oraz edytora tekstu. Niektóre przykłady korzystają z udogodnień dostarczanych przez bibliotekę `boost::function` oraz `boost::bind`, 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.

które zawierają implementację pewnego algorytmu. Tworzenie tych pierwszych można automatyzować za pomocą szablonów, które przechowują wskaźnik na funkcję lub metodę oraz argumenty, natomiast te drugie zazwyczaj tworzy się jako nowe typy, choć tutaj także możliwe są udogodnienia (na przykład biblioteka `boost::lambda`).

W języku C++ przyjęło się implementować komendę jako klasę, która dostarcza przeciążonego operatora wołania funkcyjnego, nazywaną funktorem lub obiektem funkcyjnym, patrz Listing 2. Obiekty takie mogą przechowywać stan, na przykład zapamiętywać argumenty przekazane w konstruktorze, zaś składnia wołania funktora jest taka sama jak wołanie funkcji.

Biblioteka `boost::function` pozwala tworzyć (generować) funktory, dostarcza ona szablonów, które przechowują wskaźnik na funkcję lub wskaźnik na metodę albo inny funktor, dając możliwość posługiwania się utworzonymi obiektami jak wartościami, możemy je kopiować, umieszczać w kolekcjach, zwracać jako wynik czy dostarczać jako argument.

Tworząc obiekt funkcyjny, dostarczamy informację o typie zwracanym oraz typach argumentów, patrz Listing 3, na którym pokazano tworzenie takich obiektów, używając szablonu `function`. Pierwszy parametr tego szablonu to zwracany typ, natomiast drugi, ujęty w nawiasy zwykłe, to lista typów argumentów, parametry szablonu przypominają deklarację funkcji. Ze względu na to, że niektóre kompilatory nie potrafią poprawnie interpretować takiej składni, można wykorzystać szablony `functionN`, gdzie `N` jest cyfrą oznaczającą liczbę argumentów (od 0 do 10), szablony te zawierają od 1 do 11 parametrów, typ zwracany i typy argumentów.

Każdy funktor utworzony przez tę bibliotekę dziedziczy po klasie `function_base`, która dostarcza konstruktora, konstruktora kopiującego, operatora przypisania, destruktora, a także operatora wołania funkcyjnego, patrz Listing 4. Klasy `function0`, `function1`, ... , `function10` reprezentują funktory, klasa `function` jest klasą pomocniczą, pozwalającą wykorzystywać bardziej przejrzystą składnię. Klasa bazowa `function_base` zawiera bufor, w którym przechowuje wskaźnik do funkcji, wskaźnik do metody albo wskaźnik do innego obiektu funkcyjnego. Bufor jest unią (w stylu C), której największym składnikiem jest struktura zawierająca wskaźnik do składowej oraz wskaźnik do obiektu. Obiekt klasy bazowej zawiera także wskaźnik na zarządcę bufora, więc obiekt funkcyjny utworzony za pomocą przedstawionego narzędzia ma wielkość od 16 do 32 bajtów (dwa wskaźniki plus wskaźnik na metodę). Zarządzanie buforem polega między innymi na dostarczeniu odpowiedniego kodu wołającego przechowywaną akcję, przykładowo do wołania funkcji stosu-

Listing 1. *Wzorzec komendy umożliwia opóźnione wołanie metod*

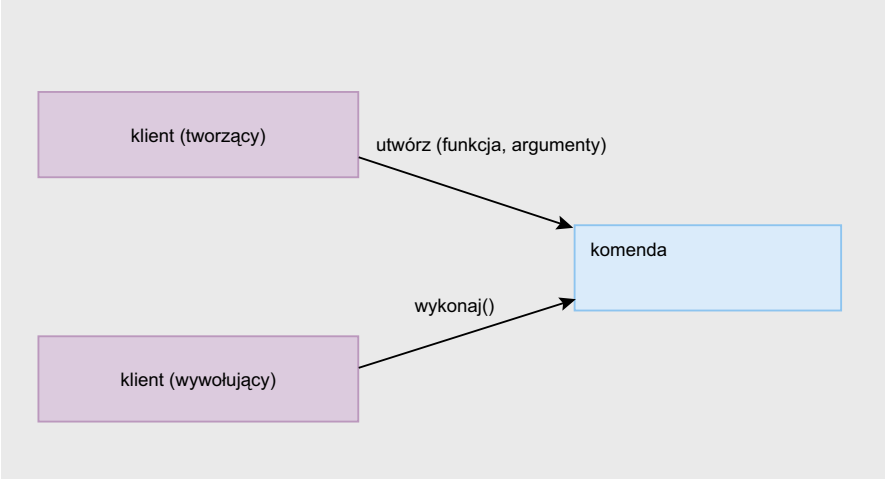
```
Window w;//utworzenie i inicjacja obiektu
w.move(20, 30); //przykładowa metoda, dostarczane są argumenty i wykonywana jest
                    akcja
class Cmd { //klasa bazowa komendy
public:
   virtual void execute() = 0;
};
class CmdMove : public Cmd { //komenda konkretna
public:
   CmdMove(Window* w, int x, int y) : w_(w), x_(x), y_(y) {}
   virtual void execute() { w_->move(x_, y_); }
private:
   Window* w_; //obiekt, do którego będzie wysyłane żądanie
   int x_, y_;//argumenty są przechowywane w składowych
};
Cmd* cmd = new CmdMove(w, 20, 30); //dostarczane argumenty, akcja nie jest
                    wykonywana
//...
cmd->execute(); //wykonanie komendy, nie wymaga dostarczania argumentów
```

Listing 2. *Funktor, czyli klasa, która dostarcza operatora wołania funkcyjnego*

```
class Funktor { //przykładowy funktor
public:
   void operator()(int i); //operator wołania funkcyjnego
};
Funktor f;
f(5); //obiekt tej klasy można wołać podobnie jak funkcję
```

Listing 3. *Funktory tworzone przez boost::function*

```
function<void (int, int)> pf; //funktor dwuargumentowy zwraca void
//boost::function2<void, int, int> pf; - to samo, inna składnia
pf(1,2); //wyjątek boost::bad_function_call, obiekt pf jest pusty
void fun(int x, int y) { cout << x + y << endl; }//przykładowa funkcja
pf = fun; //obiekt pf wskazuje na funkcję fun
pf(2,3);//opóźnione wywołanie fun dla x = 2, y = 3
struct Foo { //przykładowy funktor
   void operator()(int x, int y);
};
pf = Foo();
pf(2,3);//wywołanie Foo.operator() dla x = 2, y = 3
```



Rysunek 1. *Wzorzec komendy, upraszcza zależności w aplikacji. Kod uruchamiający komendę nie jest zależny od szczegółów dotyczących żądania*

Listing 4. *Szczegóły implementacji szablonów boost::function*

```
//11 szablonów function0, ..., function10 reprezentuje obiekt funkcyjny
//przykład dla function2
template <typename R, typename T1, typename T2>
class function2 : public function_base {
public:
   static const int arity = 2;//liczba argumentów
   function2();
   function2(const function2&);
   function2& operator=(const function2&);
   template <typename Functor> Functor* target();//dostęp do bufora
   template <typename Functor> const Functor* target() const;//dostęp do bufora
   R operator()(T1, T2, ..., TN);//operator wołania funkcyjnego
};
template <typename R, typename T1, typename T2>
R function2<R, T1, T2>::operator()(T1 arg1, T2 arg2) {
   return bufor.func_ptr(arg1, arg2); //implementacja ogólna
}
template <typename R, typename T1, typename T2>
void function2<void, T1, T2>::operator()(T1 arg1, T2 arg2) {
   bufor.func_ptr(arg1, arg2); //stosowana gdy funkcja zwraca void
}
```

Listing 5. *Wykorzystanie obiektów funkcyjnych tworzonych przez boost::function*

```
void fun() { } //przykładowa funkcja
typedef function<void ()> Action; //inna składnia: function0<void>
std::vector<Action> actions; //kolekcja komend
actions.push_back( fun ); //dodaje element do kolekcji
actions.front()(); //wykonanie pierwszej komendy, tutaj funkcji fun
```

Listing 6. *Przykłady wykorzystania boost::bind*

```
void fun(int, int); //deklaracja przykładowej funkcji, która będzie używana w
                    przykładach
function<void ()> f0 = bind(fun, 1, 2); //tworzy obiekt funkcyjny, przechowuje
                    dostarczone wartości
f0(); //wołanie tego obiektu oznacza wołanie fun(1, 2)
typedef function<void (int)> Fun1; //typ funkcji z jednym parametrem, zwracającej
                    void
Fun1 f1 = bind(fun, 1, _1); //pierwszy argument jest przechowywany w obiekcie,
                    drugi będzie pobrany przy wołaniu
f1(4); //równoważne wołaniu fun(1, 4)
void fun2(int i1, int i2, int i3, int i4); //przykładowa funkcja z 4 argumentami
Fun1 f2 = bind(fun2, _1, _1, _1, _1 ); //utworzenie obiektu funkcyjnego z jednym
                    argumentem
f2(3); //równoważne wołaniu fun2(3,3,3,3)
(bind(fun2, _4, _2, _3, _1))(1, 2, 3, 4);//równoważne wołaniu fun2(4,2,3,1)
class Foo() { //przykładowa klasa
   int f(); //przykładowa metoda (bez argumentów)
   void g(int i); //przykładowa metoda (z jednym argumentem)
};
Foo foo;//obiekt klasy używany w przykładach
f0 = bind(&Foo::f, &foo); //obiekt funkcyjny bez argumentów,  obiekt przekazany
                    przez wskaźnik
f0 = bind(&Foo::f, foo); //obiekt reprezentuje wołanie metoda f dla kopii obiektu
                    foo
f0 = bind(&Foo::g, &foo, 1); //wołanie jest równoważne wołaniu (&foo)->g(1)
f1 = bind(&Foo::g, &foo, _1); //utworzenie obiektu funkcyjnego
f1(5); //woła metodę g dla obiektu foo z argumentem równym 5
function<void (Foo*)> f4 = bind(&Foo::f, _1); //obiekt funkcyjny z jednym
                    argumentem typu Foo
```

je się szablony pokazane na Listingu 4, wybierane w zależności od tego, czy funkcja zwraca wartość, czy zwraca `void`.

Przykład przechowywania funktorów tworzonych za pomocą tej biblioteki w kontenerach standardowych został pokazany na Listingu 5. Pokazano tam możliwość umieszczania takich obiektów w wektorze oraz możliwość ich wołania.

Przedstawione szablony nie przechowują argumentów, więc nie są to typowe komendy, natomiast pozwalają one traktować wskaźnik do funkcji, wskaźnik do metody i funktor w taki sam sposób, nadając mu semantykę wartości. W obecnej realizacji obiekty te mogą mieć od 0 do 10 argumentów, ale w standardzie C++0x będą mogły mieć dowolną ich liczbę, ponieważ będą dostępne szablony ze zmienną liczbą parametrów.

Wiązanie argumentów, boost::bind

Biblioteka `boost::bind`, która będzie włączona do standardu C++0x, pozwala tworzyć obiekty funkcyjne (funktory), które przechowują wartości argumentów. Za jej pomocą możemy utworzyć obiekt funkcyjny na podstawie funkcji lub metody, korzystając z podobnych mechanizmów jak szablon `function`, dodatkowo możemy dostarczyć argumenty podczas tworzenia funktora, które są przechowywane w utworzonym obiekcie i używane przy wołaniu akcji. Argumenty mogą być dostarczane podczas wołania, a nawet mogą być wynikiem działania innych funktorów. Informacja o tym, czy argument ma być przechowywany w funktorze, czy pobrany podczas wołania, jest przechowywana dla każdego argumentu niezależnie, możemy przechowywać niektóre argumenty w obiekcie, a inne dostarczać podczas wołania. Przykład pokazany na Listingu 6 zawiera tworzenie obiektów funkcyjnych na podstawie funkcji, na podstawie metody, możliwość zmiany liczby argumentów i wiązania ich z obiektami, przechowywanymi w funktorze. Dla metod można tworzyć obiekt funkcyjny, dostarczając wskaźnik na obiekt, dla którego metoda będzie wołana lub dostarczając kopię tego obiektu.

Szablony dostarczane przez `boost::bind` są rozszerzeniem narzędzi standardowych `std::bind1st` oraz `std::bind2nd`, oferując wygodną składnię i większe możliwości. Szablony tworzą obiekt funkcyjny typu `bind_t`, który przechowuje wskaźnik do funkcji lub metody (podobnie jak `boost::functionN`), biblioteka zawiera 10 szablonów, z różną liczbą argumentów (od 0 do 9). W bibliotece zdefiniowane jest 9 obiektów globalnych o nazwach `_1`, ..., `_9`, są one generowane na podstawie szablonu `arg` (patrz Listing 7), obiekty te nie posiadają składowych i są używane do sygnalizacji, że odpowiedni argument będzie dostarczony podczas wołania. Obiekt `bind_t` przecho-

wuje poszczególne argumenty w składowej typu `list0`, `list1`, ..., `list9`, argument przechowywany w tej składowej jest instancją szablonu `value<T>` lub `arg<I>`, patrz Listing 7.

Jeżeli wołamy operator funkcyjny dla obiektu utworzonego przez `boost::bind`, to tworzona jest lista parametrów aktualnych (obiekt typu `list0` , ..., `list9`) a następnie stosuje się odpowiednią strategię wołania w zależności od tego, czy jest to funkcja, czy metoda klasy czy inny obiekt funkcyjny. Przy wołaniu rozstrzyga się, czy argument ma być pobrany z listy aktualnych parametrów (wtedy gdy użyto `_1`, ..., `_9`), czy argument został dostarczony wcześniej, na przykład podczas tworzenia obiektu funkcyjnego, wtedy oblicza się jego wartość. Obliczanie tej wartości wykonuje się rekurencyjnie za pomocą odpowiednich szablonów, dlatego możemy wykorzystać szablony `bind` do tworzenia obiektów funkcyjnych, które są kompozycją innych funktorów, w ten sposób bind rozszerza i zastępuje szablony `std::compose1` oraz `std::compose2`, patrz Listing 8. Argumentem, który będzie wiązany z odpowiednimi argumentami funkcji lub metody, może być obiekt funkcyjny generowany przez `bind`, na Listingu 8 pokazano różne kompozycje dla funkcji `f i g`.

Dla obiektów `bind_t` przeciążono operatory porównania (`==, !=, >, <, <=, >=`), dlatego zapis `(bind(less<T>(), bind(X), bind(Y) )` jest równoważny `bind(X) < bind(Y)`, co pokazano na Listingu 9. Operatory te pozwalają skrócić zapis i zrezygnować z części adapterów z biblioteki standardowej.

Listing 7. *Szablony wykorzystywane do przechowywania argumentów funktora*

```
template< int I > struct arg { //szablon arg nie posiada składowych
   arg() {}
};
boost::arg<1> _1; //obiekt globalny, podobnie arg<2>, … arg<9>
template<class T> class value { //szablon przechowujący wartości
public:
   value(T const & t): t_(t) {}
   T& get() { return t_; }
   T const & get() const { return t_; }
private:
   T t_;
};
```

Listing 8: *Kompozycja za pomocą bind*

```
int f(int x, int y); //przykładowa funkcja
int g(int z); //przykładowa funkcja
bind(g, bind(f, _1, _2) ); //dwuargumentowy obiekt funkcyjny: g( f(x,y) )
bind(f, bind(f, 1, 1 ), bind(g, _1)); //jednoargumentowy obiekt funkcyjny, f(
                      f(1,1), g(x))
```

Listing 9: *Wykorzystanie bind oraz przeciążonych operatorów i wiązania składowej*

```
struct Prac { //przykładowa struktura
   string nazw;
   int wiek;
   int pensja;
   bool operator<(const Prac& p) const { return nazw < p.nazw; }
};
vector<Prac> p; //kolekcja obiektów
sort( p.begin(), p.end() ); //sortuje po nazwisku
sort( p.begin(), p.end(), bind( less<int>(), //Sortuje po wieku
         bind(&Prac::wiek,_1),
         bind(&Prac::wiek,_2)) );
sort( p.begin(), p.end(), //sortuje po pensji
         bind(&Prac::pensja,_1) < bind(&Prac::pensja,_2) );
```

W Sieci

- *http://www.boost.org* – dokumentacja bibliotek boost;
- *http://www.codeproject.com/KB/library/BoostBindFunction.aspx* – artykuł dotyczący boost::bind;
- *http://www.open-std.org* – dokumenty opisujące nowy standard C++.

Nie zaleca się przechowywania obiektów typu `bind_t` w kolekcjach, ponieważ obiekty te nie w każdym przypadku zachowują się jak wartości. Aby mieć semantykę wartości obiekty te umieszczamy w obiektach `boost::function`.

Nienazwane obiekty funkcyjne

Dostosowywanie algorytmów standardowych do potrzeb danej aplikacji wymaga dostarczenia do algorytmu obiektu funkcyjnego, który wykonuje specyficzne przetwarzanie. Tworzenie takich obiektów wymaga utworzenia nowych typów, nawet dla wołania istniejących funkcji lub metod, ponieważ algorytmy wymagają specyficznego interfejsu. Implementacja tych typów jest nadmiarowa, gdy wołamy istniejącą metodę lub funkcję, więc zmniejsza ona czytelność kodu, klasy te są dostarczane w innym fragmencie pliku źródłowego, niż miejsce ich użycia (algorytm standardowy), co także zmniejsza czytelność. Opisane niedogodności możemy weliminować, stosując obiekty funkcyjne tworzone przez szablon `boost::bind`. Odpowiednie klasy są tworzone w czasie kompilacji, zaś ich kod jest umieszczony w algorytmie, więc jest czytelny. Wydruk 9 zawiera przykład tworzenia obiektów funkcyjnych, które porównują dwie klasy, wykorzystywany w algorytmie sortującym. Typy tworzone na potrzeby algorytmów często nie mają nazw (identyfikatorów), są wykorzystywane lokalnie, przez dany algorytm. Automatyczna generacja takich typów umożliwia pewne optymalizacje kodu wynikowego, więc zwiększamy nie tylko czytelność kodu, ale także wydajność aplikacji.

Podsumowanie

Typy generowane przez `boost::function` i `boost::bind` pozwalają tworzyć i przechowywać komendy wołające istniejące funkcje lub metody z możliwością zamiany parametrów, stosowania stałych oraz używania kompozycji. Biblioteka `boost::lambda`, która nie została tutaj opisana, pozwala na tworzenie bardziej zaawansowanych funktorów.

Komendy, oprócz możliwości separowania modułów, opóźnionego wołania akcji czy prostego sposobu implementacji wycofywania i ponawiania żądań, pozwalają na wykorzystanie przetwarzania równoległego w sposób prawie niewidoczny dla użytkownika, ponieważ komendy mogą być wykonywane przez niezależne wątki. Takie rozwiązanie stosuje się we wzorcu aktywnego obiektu, który jest tematem jednego z kolejnych artykułów.

ROBERT NOWAK

Adiunkt w Zakładzie Sztucznej Inteligencji Instytutu Systemów Elektronicznych Politechniki Warszawskiej, zainteresowany tworzeniem aplikacji dla biologii i medycyny, programuje w C++ od ponad 10 lat. Kontakt z autorem: rno@o2.pl