Wielometody

Rozszerzenie funkcji wirtualnych

Mechanizm funkcji wirtualnych pomaga wybrać metodę, biorąc pod uwagę rzeczywisty typu obiektu, dla którego daną metodę się woła. Jeżeli istnieje potrzeba wyboru funkcji w zależności od dwóch lub większej ilości typów, to odpowiedni mechanizm musimy dostarczyć sami.

Dowiesz się:

- Co to są wielometody;
- Jak implementować ten mechanizm w C++.

Powinieneś wiedzieć:

- Jak pisać proste programy w C++;
- Co to jest dziedziczenie i funkcje wirtualne;
- Co to jest wzorzec wizytatora;
- Co to jest programowanie generyczne.

Mechanizm późnego wiązania (funkcje wirtualne) pozwala współdzielić kod: posługujemy się interfejsem do klasy bazowej, natomiast specyficzne dla danego typu operacje tworzymy jako nadpisane funkcje wirtualne. Takie podejście jest bardzo wygodne, dlatego jest dostarczane przez języki wspierające obiektowe podejście do programowania (np. przez C++). Funkcje wirtualne pozwalają na wybór odpowiedniej wersji nadpisanej metody, uwzględniając rzeczywisty typ obiektu, dla którego tę metodę wołamy. Mechanizm ten ma ograniczenia – wybieramy funkcję składową w zależności od jednego typu Jeżeli istnieje potrzeba wyboru funkcji w zależności od dwóch lub większej ilości typów, to odpowiedni mechanizm musimy dostarczyć sami, jeżeli programujemy w C++. Mechanizm ten nazywany jest wielometodą (ang. *multimethods, multiple dispatch*). Pewne obiektowe języki programowania dostarczają wielometod, na przykład Common Lisp ma wbudowane wielometody, Python – mechanizm ten dostępny jako rozszerzenie. Poniższy artykuł pokazuje implementacje wielometod w C++.

Przykład, którym będziemy się posługiwali, aby pokazać zastosowania wielometod, dotyczy hierarchii figur, pokazanej na Rysunku 1. Na początek przyjrzyjmy się możliwościom, które dają funkcje wirtualne. Dla figury możemy dostarczyć metodę obliczającą jej pole. Aby wybierać odpowiedni algorytm obliczania pola, stosujemy mechanizm późnego wiązania, ponieważ funkcja ta jest zależna od jednego typu (typu figury, dla której obliczamy pole). Implementacja jest typowa: dostarczamy funkcję wirtualną w klasie bazowej, nadpisujemy ją w klasach konkretnych, dostarczając odpowiedniego algorytmu obliczania pola. Za wywołanie odpowiedniej metody jest odpowiedzialny mechanizm funkcji wirtualnych dostarczany przez język, patrz Listing 1.

Mechanizm ten nie wystarczy do implementacji funkcji `intersect`, która zwraca pole przecięcia dwóch figur. Jeżeli dysponujemy odpowiednimi funkcjami zawierającymi algorytmy obliczania pola przecięcia kół, prostokątów, prostokąta i koła itp., to problem sprowadza się do odpowiedniego wyboru tych funkcji, w zależności od rzeczywistych typów dwóch obiektów. Przykładowo, jeżeli dysponujemy funkcją `double intersect(const Circle& a, const Circle& b)`, która dostarcza pola przecięcia dwóch kół, funkcją `double intersect(const Circle& a, const Rect& b)` oraz funkcją `double intersect(const Rect& a, const Rect& b)` , to należy wywołać odpowiednią, mając dwa uchwyty do typu `Figure&` (patrz Rysunek 2). Wielometody umożliwiają takie wołanie, podob-

Szybki start

Aby uruchomić przedstawione przykłady, należy mieć dostęp do kompilatora C++ oraz edytora tekstu. Na wydrukach pominięto dołączanie odpowiednich nagłówków oraz udostępnianie przestrzeni nazw, pełne źródła dołączono jako materiały pomocnicze.

nie jak funkcje wirtualne, gdy wybór dotyczy jednego typu.Wielometody umożliwiają takie wołanie.

Wielometody wykorzystujące rzutowanie dynamiczne

Celem naszego działania będzie dostarczenie funkcji `double intersect(const Figure&, const Figure&)`. Funkcja ta otrzymuje dwa argumenty, ich typy nie są znane, mamy uchwyty do klasy bazowej. Funkcja `intersect` będzie wielometodą, będzie wołać odpowiednią wersję przeciążonej funkcji, pokazanej na Listingu 2, określając konkretne typy argumentów.

Listing 1. *Klasy reprezentujące figury. Późne wiązanie pozwala tworzyć metody zależne od jednego typu (typu figury)*

```
class Figure { //klasa bazowa
public:
   virtual double getArea() const = 0; //funkcja wirtualna, musi być nadpisana w klasie konkretnej
   virtual ~Figure() {}
};
class Rect : public Figure { //klasa konkretna
public:
   virtual double getArea() const { //nadpisuje metodę
      //oblicza pole prostokąta
   }
   //pozostałe metody i składowe
};
class Circle : public Figure { //inna klasa konkretna
public:
   virtual double getArea() const { //nadpisuje metodę
      //oblicza pole koła
   }
};
Figure* f = new Rect(); //posługujemy się wskaźnikiem do klasy bazowej
double area = f->getArea();//woła metodę uwzględniając rzeczywisty typ obiektu (tutaj typem tym jest Rect)
```

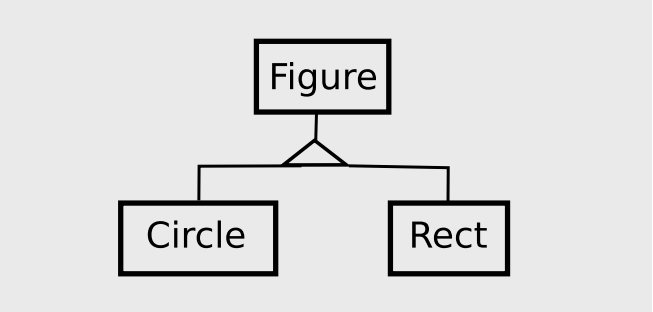
Listing 2. *Funkcje obliczające odpowiednie pola przecięcia. Argumentami są typy konkretne*

```
double intersect(const Circle& a, const Circle& b);//oblicza pole przecięcia dwóch kół
double intersect(const Circle& a, const Rect& b); //oblicza pole przecięcia prostokąta i koła
double intersect(const Rect& a, const Rect& b);//oblicza pole przecięcia dwóch prostokątów
```

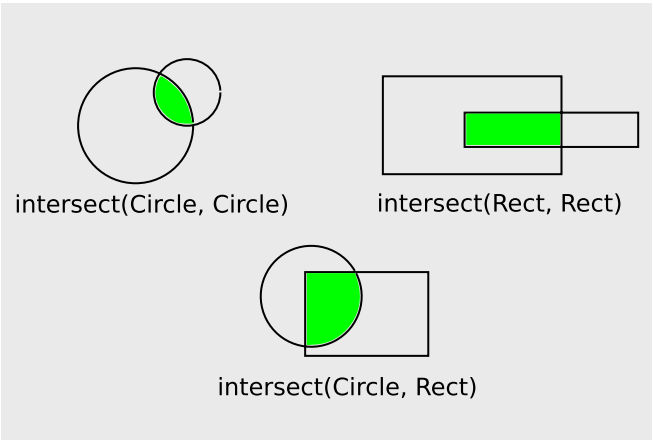
Listing 3. *Wielometody wykorzystujące rzutowanie dynamiczne. Implementacja bardzo niewydajna i trudna w pielęgnacji*

```
double intersect(const Figure&a, const Figure& b) {//dostarczamy typów bazowych
   if(Circle* pa = dynamic_cast<Circle*>(&a)) {//bada typ pierwszego argumentu
      if(Circle* pb = dynamic_cast<Circle*>(&b)){
         return intersect(*pa, *pb);//ma konkretne typy, woła dla dwóch kół
      } else if(Rect* pb = dynamic_cast<Rect*>(&b){
         return intersect(*pa, *pb);//woła przecięcie dla koła i prostokąta
      }
   } else if(Rect* pa = dynamic_cast<Rect*>(&a)){//bada typ pierwszego argumentu
      if(Circle* pb = dynamic_cast<Circle*>(&b)){
         return intersect(*pb, *pa);//zamienia kolejność argumentów, aby wołać odp. funkcję
      } else if(Rect* pb = dynamic_cast<Rect*>(&b){
         return intersect(*pa, *pb);//woła przecięcie dla dwóch prostokątów
      }
   }
   return 0.0;//nieznane typy
}
```

Wielometodę możemy realizować bezpośrednio, wykorzystując rzutowanie dynamiczne. Przykład implementacji pokazano na Listingu 3. Jest ona bardzo niewydajna i trudna w pielęgnacji, ponieważ posługuje się łańcuchem operacji rzutowania typu `dynamic_cast`. W ciągu instrukcji warunkowych badamy rzeczywisty typ pierwszego argumentu, a następnie rzeczywisty typ drugiego argumentu. Wykonujemy przy okazji wiele operacji rzutowania dynamicznego, a każda z nich jest uznawana za kosztowną, ponieważ przegląda listę klas w hierarchii (a jeżeli stosujemy dziedziczenie wielobazowe, to przegląda drzewo). Kod jest rozwlekły i zawiera powtórzenia (taki sam kod dla badania typu drugiego argumentu, gdy typ pierwszego jest ustalony). Trudność w pielęgnacji wynika także z konieczności modyfikacji łańcucha instrukcji warunkowych po zmianie hierarchii klas, gdyż w łańcuchu tym badamy wszystkie typy. Dla dużej liczby typów konkretnych kod znacznie się rozrasta, ponieważ musimy uwzględnić wszystkie pary typów. Takie rozwiązanie ma jedną zaletę – jest bardzo proste. Opisane poniżej sposoby tworzenia wielometod będą starały się usunąć wady przedstawionego rozwiązania.



Rysunek 1. *Klasy reprezentujące figury, wykorzystane do demonstracji działania wielometod*



Rysunek 2. *Funkcje obliczające pole przecięcia figur gdy znany jest ich konkretny typ*

Wykorzystanie wizytatora

Wybór funkcji w zależności od wielu typów można oprzeć o wzorzec wizytatora. Wzorzec ten oraz jego realizacja w C++ był tematem artykułu w SDJ 4/2010.

Listing 4. *Wizytator dla hierarchii figur*

```
class Visitor {//wizytator bazowy
public:
   virtual void visit(const Rect&) = 0;
   virtual void visit(const Circle&) = 0;
};
class Figure {//klasa bazowa dostarcza metody accept
public:
   virtual void accept(Visitor& v) const = 0;
   //pozostałe metody
}
class Rect : public Figure {//klasa konkretna
public:
   virtual void accept(Visitor& v) const {
      return v.visit(*this);//woła metodę visit(const Rect&) dla wizytatora
   }
   //pozostałe metody i składowe
};
class Circle : public Figure {//klasa konkretna
public:
   virtual void accept(Visitor& v) const {
      return v.visit(*this);//woła metodę visit(const Circle&) dla wizytatora
   }
   //pozostałe metody i składowe
};
```

Wizytator pozwala na wybór typu bez rzutowania dynamicznego. Klasy, dla których chcemy stosować to rozwiązanie muszą zostać zmodyfikowane. Powinny one dostarczać metodę `accept`. Należy także utworzyć klasę wizytatora bazowego, patrz Listing 4.

Tworząc wielometodę, będziemy stosowali wizytator wielokrotnie. Jeżeli rozważamy wybór funkcji w zależności od dwóch typów (najprostsza wersja wielometody), to wizytator stosujemy dwa razy, do wyboru pierwszego i drugiego argumentu. Wizytator nie wykorzystuje rzutowania dynamicznego, dlatego sposób ten jest bardzo wydajny. Wadą rozwiązania jest tworzenie klas pomocniczych (wizytatorów) oraz niebanalne przepływy sterowania. Przykład dla przecięcia figur pokazano na Listingu 5.

Wielometoda używa klasy pomocniczej,wizytatora `IntersectVisitor`, który pozwala rozstrzygnąć, jaki jest typ jednego (pierwszego) obiektu. Na wydruku pokazano, że wizytator jest przekazany jako argument metody `accept` dla obiektu `a`, więc w odpowiedniej metodzie `visit` dostaniemy obiekt `a` przekazany jako typ konkretny. Mamy więc rzeczywisty typ obiektu `a`.

Aby wyznaczyć rzeczywisty typ obiektu `b`, stosujemy wizytator po raz drugi. W zależności od typu pierwszego obiektu wizytatorem tym będzie `CircleVisitor`, jeżeli obiekt `a` jest kołem, albo `RectVisitor`, jeżeli obiekt `a` jest prostokątem. Różne klasy wizytatorów pomocniczych użytych do wyznaczania typu drugiego obiektu są potrzebne, ponieważ przechowują one rzeczywisty typ pierwszego argumentu.

Metody `visit` dla wizytatora wołanego na rzecz obiektu `b` dostarczają typu tego obiektu. Wewnątrz nich możemy wołać odpowiednią funkcję `intersect`,

Listing 5. *Wykorzystanie wizytatora do wyboru funkcji w zależności od dwu typów*

```
struct CircleVisitor : public Visitor { //wizytator, gdy pierwszym typem jest koło
   CircleVisitor(const Circle& c) : c_(c), value_(0.0) {} //przekazuje jeden argument
   virtual void visit(const Circle& c) { value_ = intersect( c_, c); }//przecięcie koła z kołem
   virtual void visit(const Rect& r) { value_ = intersect(c_, r); }//przecięcie koła z prostokątem
   const Circle& c_; //pierwszy obiekt przechowywany jako typ konkretny
   double value_;//wynik
};
struct RectVisitor : public Visitor {//wizytator, gdy pierwszym typem jest prostokąt
   RectangleVisitor(const Rect& r) : r_(r), value_(0.0) {}
   virtual void visit(Rect& r) { value_ = intersect(r, r_); } //przecięcie prostokąta z prostokątem
   virtual void visit(Circle& c) { value_ = intersect( c, r_); } //przecięcie koła z prostokątem
   Rect& r_; //pierwszy obiekt przechowywany jako typ konkretny
   double value_;//wynik
};
struct IntersectVisitor : public Visitor { //rozstrzyga dwa typy, wykorzystuje wizytatory pomocnicze
   IntersectVisitor(const Figure& fig) : fig_(fig), value_(0.0) {}//przechowuje drugi obiekt
   virtual void visit(const Circle& c) {//pierwszym typem jest koło
      CircleVisitor circVisitor(c);//tworzy odpowiedni wizytator pomocniczych
      fig_.accept( circVisitor ); //wybiera metodę w zależności od typu drugiego argumentu
      value_ =  circVisitor.value_; //przekazuje wynik obliczeń
   }
   virtual void visit(const Rect& r) { //pierwszy typ to prostokąt
      RectVisitor rectVisitor(r); //tworzy odpowiedni wizytator
      fig_.accept( rectVisitor );
      value_ = rectVisitor.value_;
   }
   const Figure& fig_;//przechowuje jeden z obiektów
   double value_; wartość zwracana
};
double intersect(const Figure& a, const Figure& b) {/multimetoda wykorzystująca wizytator
   IntersectVisitor visitor(b);
   a.accept(visitor);
   return visitor.value_;
}
```

ponieważ mamy konkretne typy obu argumentów. Wyniki obliczeń są przechowywane w składowej `value_` wizytatora, a następnie zwracane użytkownikowi.

Przedstawione rozwiązanie jest wydajne, ale wymaga ono żmudnego tworzenia klas pomocniczych (wizytatorów). Dla wielometody pozwalającej na wybór w zależności od dwóch typów, gdy różnych typów jest N, należy stworzyć N+1 wizytatorów. Tworzenie tych klas można automatyzować, wykorzystując metaprogramowanie i bibliotekę *boost::mpl* (*patrz SDJ 12/2009*). Przykład szablonów dla wielometod zawiera biblioteka *faif* (patrz ramka), ich opis zostanie zawarty w książce *Średnioza-awansowane programowanie w C++*.

Bezpośrednia implementacja późnego wiązania

Późne wiązanie możemy zaimplementować bezpośrednio, wykorzystując wielowymiarową tablicę wskaźników. Prosty przykład pokazano na Listingu 6. Implementacja bezpośrednia wymaga dostarczenia szeregu funkcji o tym samym interfejsie, więc musimy posłużyć się operatorami rzutowania dynamicznego.

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 „Średniozaawansowane programowanie w C++", która ukaże się niebawem.

Listing 6. *Wykorzystanie tablicy wskaźników do funkcji do implementacji wielometod*

```
//funkcja pomocnicza, dostarcza odpowiedni interfejs
double intersectCircleCircle(const Figure& a, const Figure& b) {
   return intersect(dynamic_cast<const Circle&>(a), dynamic_cast<const Circle&>(b) );
}
double intersectCircleRect(const Figure& a, const Figure& b) {
   return intersect(dynamic_cast<const Circle&>(a), dynamic_cast<const Rect&>(b) );
}
double intersectRectCircle(const Figure& a, const Figure& b) {
   return intersect(dynamic_cast<const Circle&>(b), dynamic_cast<const Rect&>(a) );
}
double intersectRectRect(const Figure& a, const Figure& b) {
   return intersect(dynamic_cast<const Rect&>(a), dynamic_cast<const Rect&>(b) );
}

double intersect(const Figure& a, const Figure& b) {
   enum { CIRCLE_INDEX, RECT_INDEX, N };//indeksy dla typów oraz rozmiar tablicy
   typedef double (*PF)(const Figure&, const Figure&);//typ wskaźnika na funkcję
   static const PF CALL_TAB[N][N] = {//dwuwymiarowa tablica wskaźników na funkcje
      { &calculateCircleCircle, &calculateCircleRect },
      { &calculateRectCircle, &calculateRectRect }
   };
   struct IndexVisitor : public Visitor {//pomocniczy wizytator - zwraca indeks dla typu
      IndexVisitor() : idx_(0) {}
      virtual void visit(const Circle& ) { idx_ = CIRCLE_INDEX; }
      virtual void visit(const Rectangle& ) { idx_ = RECT_INDEX; }
      int idx_;
   }
   visitorA, visitorB;
   a.accept(visitorA);
   b.accept(visitorB);
   PF fun = CALL_TAB[visitorA.idx_][visitorB.idx_];//pobiera wskaźnik z tablicy
   return (*fun)(a,b);//woła odpowiednią funkcję
}
```

W przedstawionym rozwiązaniu koszt takiego rzutowania można zaakceptować, ponieważ jest wykonywane tylko raz dla każdego argumentu. Jest to inny przypadek, niże łańcuch warunków, które wykonują rzutowanie dla wszystkich typów w hierarchii.

Implementacja bezpośrednia wykorzystuje jawnie tablicę wskaźników. Musimy dostarczyć operacji konwersji typu obiektu na indeks w tej tablicy. Działanie wielometody to odczytanie indeksów odpowiadających typom argumentów, a następnie wykorzystanie tych indeksów do wyznaczenia elementu tablicy, która zawiera odpowiedni wskaźnik na funkcję. Na koniec funkcja ta jest wołana z danymi argumentami. W przedstawionym rozwiązaniu należy dostarczyć odpowiednich wskaźników na funkcje, co w najprostszym rozwiązaniu wymaga rezygnacji z przeciążania nazw. Na Listingu 6 pokazano funkcje, których nazwa sugeruje typy argumentów.

Rozwiązanie bezpośrednie zmusza nas do niewygodnych (niskopoziomowych) konwersji. Kompilator nie wspiera programisty przy sprawdzaniu poprawności generowania indeksów itd.

Podsumowanie

Wielometody pozwalają wybrać funkcję w zależności od kilku typów. W artykule pokazano przykład wielometod dla dwóch typów, ponieważ z taką sytuacją mamy do czynienia najczęściej, ale ilość tych typów można zwiększyć.

Każde z rozwiązań ma swoje mocne i słabe strony. Najbardziej wydajne jest rozwiązanie wykorzystujące wizytatory, ale wymaga ono implementacji tego wzorca dla typów, które będą badane. Rozwiązania pozostałe są prostsze i nie narzucają dodatkowych wymagań na wybierane typy, ale są mniej wydajne, wykorzystują stosunkowo wolne operatory rzutowania dynamicznego. W praktyce warto stosować szablony, które dostarczają odpowiednie klasy pomocnicze, ukrywając przed użytkownikiem niepotrzebne szczegóły.

W Sieci

- *http://www.boost.org* – dokumentacja bibliotek boost;
- *http://faif.sourceforge.net* – biblioteka zawierająca wielometody dla C++ (w wersji 0.29);
- *http://www.codeproject.com/KB/recipes/mmcppfcs.aspx* – artykuł opisujący wielometody.

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