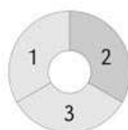
Fabryki obiektów

Techniki opisane w tym artykule pozwalają tworzyć obiekty na podstawie identyfikatorów dostarczanych w czasie działania programu, co jest wygodniejsze niż podawanie typów w formie zrozumiałej dla kompilatora.

Dowiesz się:
- Jak tworzyć obiekty w C++;
- Co to jest wzorzec fabryki.

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

Poziom trudności



Fabryka obiektów jest klasą, której obiekty pośredniczą przy tworzeniu innych obiektów. Dostarczana informacja jednoznacznie identyfikuje konkretny typ, znany w momencie kompilacji, ale nie jest to literał, więc informacja o typie jest nieodpowiednia dla kompilatora, na przykład jest to napis lub inny identyfikator. Fabryka ukrywa przed użytkownikiem mechanizm zamiany identyfikatora na literał dostarczany do operatora `new`, upraszczając tworzenie obiektów (patrz Rysunek 1).

W języku C++ podczas tworzenia obiektu należy podać konkretny typ w formie zrozumiałej dla kompilatora (patrz Listing 1). Nie możemy posłużyć się mechanizmem funkcji wirtualnych, nie możemy także przekazać identyfikatora typu w czasie działania, argumentem operatora `new` może być tylko literał oznaczający typ znany w momencie kompilacji. Po utworzeniu obiektu można się do niego odwoływać poprzez wskaźnik lub referencję na klasę bazową, ale przy tworzeniu należy podać rzeczywisty typ obiektu, nie można użyć mechanizmu późnego wiązania (funkcji wirtualnych).

Funkcje fabryczne

Przykład wykorzystania fabryki obiektów został pokazany na Listingu 2, gdzie pokazano funkcję `create` tworzącą obiekty klas na podstawie danych zapisanych w strumieniu wejściowym, na przykład w pliku. Funkcja ta dostarcza obiekt odpowiedniego typu konkretnego dla hierarchii `Figure`. Metody zapisu (`write`) oraz odczytu (`read`) są dostarczane przez każdą z klas konkretnych. Jeżeli chcemy obiekt utworzyć (odczytać), to typ obiektu jest dostarczany w czasie działania, jest on wyznaczany przez identyfikator dostarczany przez strumień. Ponieważ tak dostarczany identyfikator nie jest akceptowany jako argument dla operacji `new`, należy wykorzystać fabrykę, która pozwoli tworzyć obiekty na podstawie identyfikatora, rolę tę pełni funkcja `createObj`. Po utworzeniu obiektu możemy wczytać składowe, wykorzystując mechanizm funkcji wirtualnych, wołając metodę `read` dla utworzonego obiektu.

Przy zapisie obiektu do strumienia wyjściowego (na przykład do pliku) nie potrzebujemy dodatkowych mechanizmów, które dostarczą identyfikator dla danego typu, ponieważ możemy wykorzystać mechanizm funkcji wirtualnych. Posiadając wskaźnik lub referencję na klasę bazową, wołamy metodę `write`, która zapisuje identyfikator klasy konkretnej oraz jej składowe. Odczyt obiektu jest o wiele bardziej złożony niż zapis ze względu na to, że zapis posługuje się istniejącymi obiektami, natomiast odczyt musi je tworzyć.

Fabryka skalowalna

Funkcja `createObj` jest prostą fabryką obiektów, pozwala ona tworzyć obiekty na podstawie informacji, która jest dostarczana w czasie działania, ale ma wiele wad: mapowanie identyfikatora na typ za pomocą instrukcji

Listing 1. *Podczas tworzenia obiektu należy podać typ w odpowiedniej formie*

```
class Bazowa { /* ... */ }; //przykładowa definicja typu
class KonkretnaA : public Bazowa { /* ... */ };
Bazowa* p = new KonkretnaA; //przy tworzeniu trzeba podać konkretny typ
//nazwa typu musi być zrozumiała dla kompilatora
```

Szybki start

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

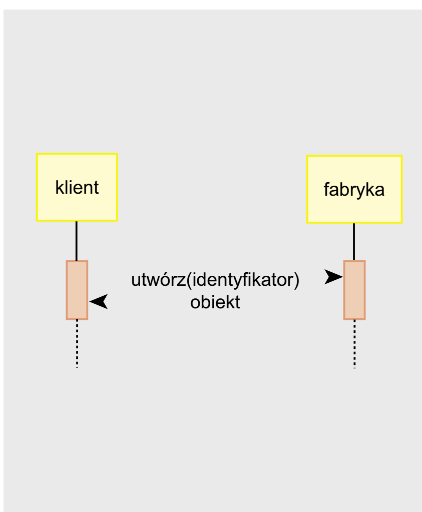
`switch` sprawia, że funkcja ta jest zależna od wszystkich klas w hierarchii, jeżeli będzie dodawana lub usuwana jakaś klasa, to modyfikacji będzie musiał podlegać także kod fabryki; brak kontroli przy wiązaniu identyfikatora z typem sprawia, że musimy zapewnić, aby przy odczycie obiektu korzystać z tego samego identyfikatora co przy zapisie. Poza tym, zestaw identyfikatorów jest zasobem globalnym, przy modyfikowaniu zbioru klas w danej hierarchii musi on podlegać modyfikacjom.

Fabryka skalowalna, przedstawiona na Listingu 3, umożliwia tworzenie obiektów na podstawie identyfikatorów, wprowadza mniejszą liczbę zależności w porównaniu z poprzednio omówionym rozwiązaniem, ponieważ jest ona zależna tylko od klasy bazowej, a nie od wszystkich klas konkretnych. Mniejsza liczba zależności wynika z zastosowania wskaźnika na funkcję tworzącą obiekty danej klasy konkretnej oraz przez użycie dynamicznej struktury przechowującej mapowanie pomiędzy identyfikatorem a typem.

Klasa konkretna woła metodę `registerFig`, przekazując swój identyfikator oraz funkcję tworzącą obiekty danej klasy. Metoda ta dodaje element do kolekcji, można więc elastycznie modyfikować zestaw klas, których obiekty będą tworzone przez fabrykę. Jeżeli chcemy usuwać wpisy, to należy zaimplementować metodę `unregister`, która będzie usuwała elementy z kolekcji.

Tworzenie obiektów odbywa się w metodzie `create`, która wyszukuje funkcję tworzącą dla danego identyfikatora. Jeżeli taka funkcja zostanie znaleziona, to jest ona wołana (patrz Listing 3), a obiekt utworzonej klasy jest zwracany.

Klasa konkretna musi dostarczyć funkcję tworzącą obiekty, funkcja ta (patrz Listing 4) może być umieszczona w module zawierającym implementację klasy konkretnej, podobnie moduł ten może zawierać kod rejestrujący dany typ w fabryce, tak jak pokazano na Listingu 4.



Rysunek 1. *Tworzenie obiektów przez fabrykę*

Fabryka skalowalna jest bardziej elastyczna, ale też bardziej kosztowna niż rozwiązanie bezpośrednie (wybór typu w zależności od identyfikatora za pomocą `switch` lub łańcucha `if ... else`), ponieważ wymaga przechowywania kolekcji wiążącej identyfikator z funkcją tworzącą. Obiekt jest tworzony za po-

Listing 2. *Przykład, w którym uzasadnione jest wykorzystanie fabryki*

```
class Figure { //klasa bazowa
public:
   enum Type { SQUARE, CIRCLE, /* ... */ };//identyfikatory klas konkretnych
   virtual bool write(ostream& os) const = 0;//zapisuje obiekt
   virtual bool read(istream& is) = 0;//odczytuje obiekt
};
class Square : public Figure {//jedna z klas konkretnych
public:
   bool zapisz(ostream& os) {//zapisuje kwadraty
   os << KWADRAT;//zapisuje identyfikator typu
   //zapisuje poszczególne składowe
}
bool read(istream& is) {//odczytuje obiekt, zakładając, że jest to kwadrat
   //odczytuje poszczególne składowe
}
};
//pozostałe klasy konkretne także dostarczają metody odczytu i zapisu
//...
Figure* createObj(istream& is) {//funkcja pełni rolę fabryki
   Figure::Type type;
   is >> type; //odczytuje identyfikator typu
   Figure* obj;
   switch(type) { //zapewnia mapowanie pomiędzy identyfikatorem a typem
      case SQUARE://w formie zrozumiałej dla kompilatora
         return new Square(); //tworzy odpowiedni obiekt
      case CIRCLE: /* ... */
   }
}
//tworzy obiekt na podstawie identyfikatora i odczytuje jego składowe
Figure* create(istream& is) {
   Figure* obj = createObj(is); //tworzy obiekt odpowiedniego typu
   obj->read(is);//obiekt istnieje, może wykorzystać funkcje wirtualne
}
```

Listing 3. *Fabryka skalowalna*

```
class FigFactory {
public:
   typedef Figure* (*CreateFig)(); //wskaźnik na funkcję tworzącą obiekt
   //rejestruje nowy typ
   void registerFig(int id, CreateFig fun) {
      creators_.insert( value_type(id, fun) ); //dodaje do kolekcji
   }
   //tworzy obiekt na podstawie identyfikatora
   Figure* create(int id) { //tworzy obiekt danego typu
      Creators::const_iterator i = creators_.find(id);
      if(i ! = creators_.end() ) //jeżeli znalazł odpowiedni wpis
         return (i->second)(); //woła metodę fabryczną
      return 0L; //zwraca pusty wskaźnik, gdy nieznany identyfikator
   }
private:
   typedef std::map<int, CreateFig> Creators;
   Creators creators_;//przechowuje powiązania pomiędzy identyfikatorem a funkcją
                         tworzącą
};
```

średnictwem tej funkcji, więc tworzenie trwa dłużej (jeden skok więcej w porównaniu z metodą bezpośrednią).

Aby zaimplementować w pełni funkcjonalną fabrykę obiektów, należy uwzględnić dodatkowe zagadnienia. Po pierwsze, problem dostarczania odpowiedniego obiektu fabryki wymaganego przy rejestracji klas (na wydruku 4 została użyta funkcja getFactory). Często stosowanym rozwiązaniem jest singleton. Po drugie, należy zarządzać czasem życia powołanych obiektów, fabryka tworzy obiekty na stercie, ale kto ma je zwalniać? W tym celu warto posłużyć się sprytnymi wskaźnikami (patrz SDJ 11/2009). Kolejnym zadaniem jest wiązanie identyfikatora z typem, aby wykluczyć możliwości pomyłek. Można odpowiedzialność dostarczania identyfikatorów przenieść na fabrykę, obiekt ten ma może tworzyć unikalne identyfikatory, zaś klasy, które są rejestrowane w fabryce, mogą ten identyfikator przechowywać w składowej statycznej (patrz Listing 5). Obiekty klas będą wykorzystywały ten identyfikator podczas zapisu, natomiast fabryka wykorzystuje go podczas odczytu.

Inną możliwością jest generowanie identyfikatora przez mechanizmy kompilatora, na przykład używając struktury `type_id`, wtedy nie trzeba zarządzać nim w fabryce.

Inicjacja fabryki skalowalnej (rejestracja typów) może być uproszczona, jeżeli będzie wykorzystywana kolekcja typów z biblioteki boost::mpl (patrz SDJ 12/2009) oraz algorytmy, które na kolekcji operują(patrz Listing 6). Funkcja tworząca może być metodą statyczną klasy konkretnej, wtedy nie musi zawierać nazwy tworzonej klasy. Takie rozwiązanie prezentuje Listing 6.

Listing 4. *Przykład funkcji tworzącej i rejestracji typu w fabryce*

```
Figure* CreateSquare() { //funkcja tworząca dla typu konkretnego
   return new Square();
};
FigFactory& factory = getFactory();//pobiera obiekt fabryki
factory.registerFig(SQUARE, CreateSquare); //rejestruje się w fabryce
```

Listing 5. *Fabryka skalowalna zarządzająca identyfikatorami*

```
typedef shared_ptr<Figure> PFigure; //sprytny wskaźnik
class FigFactory {
public:

   typedef PFigure (*CreateFig)(); //wskaźnik na funkcję tworzącą obiekt
   int registerFig(CreateFig fun) {//zwraca id zarejestrowanego typu
      creators_.insert( value_type( currentId_, fun) );
      return currentId_++; //zwraca identyfikator
   }
   PFigure create(int id); //tworzy obiekt danego typu (patrz Listing 3)

private:
   int currentId_; //kolejny wolny identyfikator
   //składowe związane z funkcjami tworzącymi
};
FigFactory& factory = FigFactory::getInstance();//singleton
Square::id_ = factory.registerFig(CreateSquare); //ustawia identyfikator
```

Listing 6. *Rejestracja klas w fabryce skalowalnej przez algorytm biblioteki boost::mpl*

```
class Square : public Figure {
public:

   //każda klasa konkretna dostarcza metodę statyczną create
   static Figure* create() { return new Square; }
};

struct RegisterFigure { //szablon użyty do rejestracji
   template<typename T> void operator()(T) {
      T::id_ = Factory::getInstance().registerFig( T::create );
   }
};
typedef mpl::vector<Square, Circle> Types; //kolekcja typów dla klas konkretnych
mpl::for_each< Types > ( RegisterFigure() ); //woła w czasie wykonania operację dla
                              każdego typu
```

Podsumowanie

Opisany sposób tworzenia obiektów na podstawie identyfikatora dostarczanego podczas działania programu jest jednym z wzorców kreacyjnych, termin został zaproponowany przez *bandę czworga* (Gamma, Helm, Johnson, Vlissides) w książce *Wzorce projektowe*. Inne udogodnienia dotyczące tworzenia obiektów, takie jak fabryki prototypów i fabryki abstrakcyjne, są tematem jednego z kolejnych artykułów.

W Sieci

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

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.

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