Zrób Sobie Firmę

Co to jest programowanie obiektowe?

Dodatek Informatyczny (II)

Co to jest programowanie obiektowe?

1.Wprowadzenie

Nie wszystkie języki programowanie mogą być "zorientowane obiektowo". Ale APL, Ada, Clu, C++ i Smaltalk są językami obiektowymi. "Obiektowy" w wielu kręgach stał się synonimem "dobry", a w prasie branżowej można było znaleźć takie sylogizmy:

-Ada jest dobra

-Obiektowość jest dobra

-Ada jest obiektowa

Po prostu musimy być bardziej ostrożni w naszych pojęciach i logice. Artykuł ten przedstawia spojrzenie na to, co powinno oznaczać "zorientowany obiektowo" w kontekście języków programowania ogólnego przeznaczenia. Przykłady będą podawane w C++. Powodem tego jest to, że język C++ jest jednym z kilku języków, które wspierają abstrakcję danych i programowanie obiektowe dodatkowo do tradycyjnych technik programowania.

2.Paradygmaty programowania

Programowanie obiektowe jest techniką programowania - paradygmatem dla pisania "dobrych" programów dla zbioru problemów. Jeśli termin "język programowania obiektowego" coś oznacza, musi oznaczać język programowania, który udostępnia mechanizmy, które wspierają obiektowy styl programowania. Tu istnieje istotna różnica. Mówi się, że język wspiera styl programowania, jeśli dostarcza udogodnień, które umożliwiają wygodne (dość proste, wydajne i bezpieczne) zastosowanie tego stylu. Język nie obsługuje techniki, jeśli wymaga dodatkowego wysiłku i umiejętności w pisaniu takich programów; jedynie pozwala stosowanie tej techniki. Na przykład, możesz napisać programy strukturyzowane w Fortranie, pisać bezpieczne programy w C i użyć abstrakcji danych w Modula-2, ale jest to trudne, ponieważ języki te nie wspierają tych technik. Wsparcie dla paradygmatu jest nie tylko oczywistą postacią narzędzi języka, które pozwalają na bezpośrednie zastosowanie tego paradygmatu, ale również w bardziej subtelnej formie czasu kompilacji i /lub w czasie wykonania, kontrolę przed przypadkowym odchyleniem od paradygmatu. Najbardziej oczywistym przykładem tego jest sprawdzanie typu. Niejednoznaczność wykrywania i sprawdzania w czasie uruchamiania może być zastosowane dla rozszerzenia wsparcia językowego dla paradygmatów. Pozajęzykowe udogodnienia takie jak biblioteki standardowe i środowiska programistyczne mogą również dostarczyć znaczącego wsparcia dla paradygmatów. Jeden język nie jest koniecznie lepszy od innego, ponieważ posiada tą funkcję a inny nie. Istnieje na to wiele przykładów. Ważną kwestią jest nie to, co język posiada, ale czy posiadane funkcję są wystarczające dla wsparcia pożądanego stylu programowania w wybranym obszarze zastosowań:

1.Wszystkie funkcje muszą być dokładnie i elegancko zintegrowane w języku

2.Musi być możliwe stosowanie funkcji w połączeniu, w celu osiągnięcia rozwiązań, które w przeciwnym razie wymagałoby dodatkowy oddzielnych funkcji

3.Powinno być jak najmniej fałszywych i "specjalnego przeznaczenia" funkcji

4.Funkcja powinna być taka, że jej realizacja nie nakłada znaczących kosztów na programy, które jej nie wymagają

5.Użytkownik musi znać jedynie podzbiór języka wyraźnie stosowany do pisania programu

Ostatnie dwie zasady mogą być podsumowane, jako "to, czego nie wiemy, nie boli". Jeśli istnieją jakiekolwiek wątpliwości, co do przydatności funkcji, lepiej ją pominąć. Dużo łatwiej jest dodawać funkcje do języka niż usuwać lub modyfikować tą, która znalazła swoją drogę do kompilatorów. Przedstawimy kilka styli programowania i kluczowych mechanizmów języka koniecznych dla ich wsparcia. Oczywiście nie wyczerpuje to tego tematu.

Programowanie proceduralne

Pierwotny (i prawdopodobnie jeszcze powszechnie używany) paradygmat programowania brzmi:

"Zdecyduj, jakiej procedury chcesz; użyj najlepszych algorytmów, jakie możesz znaleźć"

Skupiamy się na projektowaniu przetwarzania, algorytmie koniecznym dla wykonania żądanego obliczenia. Języki wspierają ten paradygmat przez możliwość przekazywania argumentów do funkcji i zwracania wartości z funkcji. Literatura powiązana z tym sposobem myślenia jest wypełniona omówieniami sposobów przekazywania argumentów, sposobów rozróżniania różnych rodzajów argumentów, różnych rodzajów funkcji (procedury, podprogramy, makra...) itp. Fortran jest oryginalnym językiem proceduralnym; Algol60,Algol68,C i Pascal są późniejszymi wynalazkami w tej samej tradycji. Typowym przykładem "dobrego stylu" jest funkcja pierwiastka kwadratowego. Przy danym argumencie, tworzy wynik. Aby to zrobić, wykonuje dobrze znane obliczenia matematyczne:

```
double sqrt(double arg)
{
        // kod dla obliczenia pierwiastka kwadratowego
}
void  jakaś_funkcja ()
{
        double root2 = sqrt(2);
        // ....
}
```

Z punktu widzenia organizacji programu, funkcje są używane do tworzenia porządku w labiryncie algorytmów.

Ukrywanie danych

Z biegiem lat, nacisk w projektowaniu programów przesunął się od projektowania procedur do organizacji danych. Między innymi odzwierciedla to wzrost rozmiaru programu. Zbiór powiązanych procedur z danymi, którymi manipulują jest często nazywany modułem. Paradygmat programowania to:

"Zdecyduj, jakich modułów chcesz; podziel program tak, aby ukryć dane w modułach"

Ten paradygmat jest również znany, jako "zasada ukrywania danych". Jeśli nie ma grupowania procedur z powiązanymi danymi, wystarczy styl programowania proceduralnego. W szczególności, techniki dla projektowania "dobrych procedur" są obecnie stosowane dla każdej procedury w module. Najbardziej typowym przykładem jest definicja modułu stosu. Główne problemy, jakie należy rozwiązać:

1.Dostarczanie interfejsu użytkownika dla stosu (na przykład funkcje push() i pop())

2.Upewnienie się, że reprezentacja stosu (np. wektor elementów) może być tylko dostępny przez ten interfejs użytkownika

3.Upewnienie się, że stos jest inicjowany przed jego pierwszym użyciem

Oto wiarygodny zewnętrzny interfejs dla modułu stosu:

```
// deklaracja interfejsu modułu stosu znaków

char pop();

void push(char);

const stack_size = 100;
```

Zakładając, że ten interfejs znajduje się w pliku nazwanym stack.h, "wewnętrznie" może być definiowany tak:

```
#include "stack.h"

static char v[stack_size];          // 'static' oznacza lokalnie do tego pliku/modułu

static char* p = v;                 // stos jest początkowo pusty


char pop()

{

        //  sprawdzenie i odłożenie

}

void push(char c)

{

        // sprawdzenie i zdjęcie

}
```

Byłoby całkiem możliwe zmienić reprezentację tego stosu na listę połączoną. Użytkownik nie ma dostępu do tej reprezentacji, (ponieważ v i p zadeklarowano statycznie, to znaczy lokalnie do pliku/modułu, w którym zostały zadeklarowane. Taki stos można użyć tak:

```
#include "stack.h"
void jakaś_funkcja ()
{
        push ('c');
        char c = pop();
        if (c != 'c') error ("niemożliwe");
}
```

Pascal (zdefiniowany pierwotnie) nie zapewnia odpowiednich możliwości do takiego grupowania: jedynie mechanizm dla ukrywania nazw z " reszty programu" jest tworzony, jako lokalny do procedury. Prowadzi to dziwnego zagnieżdżania procedury i nadmiernego polegania na danych globalnych. C wgląda nieco lepiej. Jak pokazano w powyższym przykładzie, można zdefiniować "moduł" przez zgrupowanie powiązanych funkcji i definicji danych razem w pojedynczym pliku źródłowym. Programista może potem kontrolować, jakie nazwy są widziane przez resztę programu (nazwa może być widziana przez resztę programu, chyba, że jest zadeklarowana statycznie). W konsekwencji w C możemy osiągnąć pewien stopień modularności. Jednak nie ma ogólnie przyjętego paradygmatu dla stosowania tej możliwości i techniki w oparciu o deklaracje statyczne są raczej niskopoziomowe. Jeden z następców Pascala,Modula-2, idzie nieco dalej. To formalizuje pojęcie modułu, czyniąc go fundamentalną konstrukcją języka z dobrze zdefiniowanymi deklaracjami modułu, wyraźną kontrolą zakresu nazw (import/eksport), mechanizm inicjalizacji modułu i zbiór powszechnie znanych i akceptowanych stylów zastosowania. Różnica między C a Modula-2 w tym obszarze może być podsumowana stwierdzeniem, że C tylko umożliwia rozkład programu na moduły, podczas gdy Modula-2 wspiera tą technikę

Abstrakcja danych

Programowanie z modułami prowadzi do scentralizowania wszystkich typów danych pod kontrolą typu menadżera modułu. Jeśli ktoś chce dwa stosy, definiuje menadżera modułu stosu z interfejsem jak ten:

```
class stack_id;             //stack_id jest typem
                            //brak szczegółów o stosach lub stack_id są znane tu
stack_id create_stack(int size); // tworzymy stos i zwraca jego identyfikator
destroy_stack(stack_id);            //wywołujemy kiedy stos nie jest dłużej potrzebny
void push(stack_id, char);
char pop(stack_id);
```

Jest to z pewnością o wiele lepsze od tradycyjnego niestrukturalnego bałaganu, ale "typy" implementowane w ten sposób wyraźnie różnią się od typów wbudowanych w język. Każdy typ menadżera modułu musi definiować oddzielny mechanizm tworzenia "zmiennych" tego typu, nie ma ustalonych norm dla przypisywania identyfikatorów obiektu, "zmienna" tego typu nie ma nazwy znanej kompilatorowi lub w środowisku programistycznym, ani taka zmienna nie przestrzega zwykłych zasad zakresu ani zasad przekazywania argumentu. Typy tworzone przez mechanizm modułu są w najważniejszych aspektach różne od typów wbudowanych i oferują wsparcie niższe niż w stosunku do wsparcia przewidzianego dla typów wbudowanych. Na przykład:

```
void f()
{
        stack_id s1;
        stack_id s2;

        s1 = create_stack(200);
        // Ups : zapomnieliśmy stworzyć s2

        push(s1, 'a');
        char c1 = pop(s1);
        if (c1 != 'a')  błąd ("niemożliwe");
        push(s2,'b');
        char c2 = pop(s2);
        if (c2 != 'b') błąd ("niemożliwe");

        destroy_stack(s2);
        // Ups : zapomnieliśmy zniszczyć s2
}
```

Innymi słowy, koncepcja modułu, który wspiera paradygmat ukrywania danych pozwala na ten styl programowania, ale go nie wspiera. Języki takie jak Ada, Clu i C++ atakują ten problem przez zezwolenie użytkownikowi na definiowanie typów, które zachowują się w (prawie) ten sam sposób jak typy wbudowane. Taki typ jest często nazywany abstrakcyjnym typem danych. Lepszą jednak nazwą byłaby "typ zdefiniowany przez użytkownika".

Paradygmat programowania brzmi:

"Zdecyduj, jakich typów chcesz; zapewnij pełny zbiór działań dla każdego typu"

W przypadku, gdy nie ma potrzeby więcej niż jednego obiektu tego typu, wystarcza użycie modułów, jako styl programowania z ukrywaniem danych. Typy arytmetyczne takie jak liczby wymierne i zespolone są typowym przykładem typu zdefiniowanego przez użytkownika:

```
class complex {
        double re, im;
public:
        complex(double r, double i) {re = r ; im = 1}
        complex(double r) { re = r; im = 0;}                    // konwersja float -> zespolona

        friend complex operator+(complex, complex);
        friend complex operator-(complex, complex);   // minus binarny
        firend complex operator-(complex)                       // minus jednoargumentowy
        friend complex operator*(complex, complex);
        friend complex operator/ (complex,complex);
        // ....
};
```

Deklaracja klasy (to znaczy typu zdefiniowanego przez użytkownika) complex określa reprezentację liczby zespolonej i zbiór działań na liczbie zespolonej. Reprezentacja jest private, to znaczy, re i im są dostępne tylko dla funkcji określonych w deklaracji klasy complex. Takie funkcje mogą być zdefiniowane tak:

```
complex operator+(complex a1, complex a2)
{
return complex(a1.re+a2.re, a1.im + a2.im);
}
```

i używać tak:

```
complex a = 2.3;
complex b = 1/a'
complex c = a+b*complex(1,2,3);
// ...
c= -(a/b)+2;
```

Większość, ale nie wszystkie, moduły są lepiej wyrażone, jako typy definiowane przez użytkownika. Dla koncepcji, gdzie "reprezentacja modułu" jest pożądana nawet, kiedy jest dostępna właściwa

funkcja dla zdefiniowania typów, programista może zadeklarować typ i tylko pojedynczy obiekt tego typu. Alternatywnie, język może dostarczyć koncepcji modułu dodatkowo i inaczej niż koncepcja klasy.

Problemy z abstrakcją danych

Abstrakcyjny typ danych definiuje rodzaj czarnego pola. Kiedy został zdefiniowany, w rzeczywistości nie współdziała z resztą programu. Nie ma sposobu zaadaptowania go do nowego zastosowania wyjątkiem modyfikacji jego definicji. Może to prowadzić do poważnej nieelastyczności. Rozważmy definicję typu shape dla zastosowania w systemie graficznym. Załóżmy na chwilę, że system musi wspierać okręgi, trójkąty i kwadraty. Załóżmy również, że mamy klasy:

class point { /* … */ };

class color { /* … */ };

Możemy zdefiniować kształt tak:

enum kind {okrąg, trójkąt, kwadrat};

```
class shape{
        point center;
        color col;
        kind k;
        // przedstawiania kształtu
public:
        point where ()          {return center;}
        void move(point to)     {center = to ; draw(); }
        void draw ();
        void rotate(int);
        // więcej działań
};
```

"Pole typu" k jest konieczne dla zezwolenia działaniom takim jak draw() i rotate( ) dla określenia, na jakiego rodzaju kształcie działają. Funkcja draw () może być zdefiniowana tak:

```
void shape ::draw()
{
        switch (k) {
        case okrąg:
```

```
        // rysujemy okrąg
                break;
        case trójkąt:
                // rysujemy trójkąt


        break;
        case kwadrat:
                // rysujemy kwadrat
        }
}
```

To jest bałagan. Funkcja taka jak draw () musi "wiedzieć" o wszystkich rodzajów kształtów. Dlatego, kod dla takiej funkcji wzrasta za każdym razem kiedy dodajemy nowy kształt do systemu. Jeśli zdefiniujemy nowy kształt, każde działanie na kształcie musi być zbadane (i możliwie) zmodyfikowane. Nie będzie można dodać nowego kształtu do systemu chyba, że masz dostęp do kodu źródłowego dla każdego działania. Ponieważ dodawanie nowego kształtu obejmuje "dotknięcie" kodu przy każdy ważnym działaniu na kształtach, wymaga dużych umiejętności i potencjalnie wprowadza błędy w kodzie obsługi innych (starszych) kształtów.

Programowanie obiektowe

Problem polega na tym, że nie ma różnicy między ogólnymi właściwościami dowolnego kształtu (kształt ma kolor, może być narysowany itd.) a właściwościami określonego kształtu (okrąg jest kształtem, który ma promień, jest rysowany przez funkcję rysowania okręgu itp.). Wyrazimy to rozróżnienie i wykorzystamy definicję programowania obiektowego. Język z konstrukcją, która pozwala wyrażać i używać tego rozróżnienia wspiera programowanie obiektowe. Inne języki nie. Mechanizm dziedziczenia Simula dostarcza rozwiązania. Po pierwsze, określamy klasę, która definiuje ogólne właściwości wszystkich kształtów:

```
class shape _
        point center;
        color col;
// …
public:
point where() {return center;)
void move (point to) {center = to; draw();}
virtual void draw();
virtual void rotate(int);
```

```
// ...
};
```

Funkcje, dla których interfejs wywołania może być zdefiniowany, ale gdzie implementacja nie może być zdefiniowana z wyjątkiem określonego kształtu, oznaczymy, jako "virtual" (W Simula i C++ termin "może być redefiniowane później w klasie pochodnej z tej klasy"). Przy danej definicji, możemy ogólną funkcję manipulowania kształtami:

```
void rotate_all(shape* v, int size, int angle)
// obracamy wszystkie składowe wektora "v" rozmiaru "size" o stopień "angle"
{
        for (int i = 0; i < size ; i++) v[i].rotate(angle);
}
```

Definiując określony kształt, musimy powiedzieć, że jest to kształt i określić jego właściwości (w tym funkcje wirtualne)

```
class circle : public shape {

        int radius;
public:
        void draw () { /* ... */ };
        void rotate(int) {}          // tak funkcja null
};
```

W C++, klasa circle będzie pochodną z klasy shape, a klasa shape będzie klasą bazową dla klasy circle. W alternatywnej terminologii nazywamy circle i shape subklasą i superklasą, odpowiednio. Paradygmat programowania to:

"Zdecyduj, jakich klas chcesz; zapewnij pełny zbiór działań dla każdej klasy; utwórz wspólność wyraźnie za pomocą dziedziczenia"

Tam gdzie nie ma wspólności wystarcza abstrakcja danych. Ilość wspólności między typami, które można wykorzystać za pomocą dziedziczenia i funkcji wirtualnych jest papierkiem lakmusowym możliwości zastosowania programowania obiektowego w obszarze zastosowania. W pewnych obszarach, takich jak grafika interaktywna, istnieją ogromne możliwości w zakresie programowania obiektowego. W innych obszarach, takich jak klasyczne typy arytmetyczne i obliczeniach opartych na nich, wydaje się to być trudnym zakresem dla więcej niż abstrakcji danych i funkcji potrzebnych dla wsparcia programowania zorientowanego obiektowo. Znalezienie podobieństw między typami w systemie nie jest procesem trywialnym. Ilość wspólności będących wykorzystywanymi wpływa na sposób, w jaki system zaprojektowano. Kiedy projektujemy system, podobieństwo musi być aktywnie poszukiwane zarówno przez projektowanie klasy jak i bloki budowane dla innych typów , a poprzez analizę klas widać czy wykazują one podobieństwa, które mogą być wykorzystane we

wspólnej klasie bazowej. Po zapoznaniu się ze minimalnymi wsparciem dla programowania proceduralnego, ukrywania danych, abstrakcji danych i programowania obiektowego, przejdziemy do dość szczegółowego opisu funkcji, które - gdy nie są konieczne - mogą tworzyć bardziej efektywną abstrakcję danych i orientacji obiektowej.

3. Wsparcie dla abstrakcji danych

Podstawowe wsparcie dla programowania z abstrakcyjnymi danymi składa się z funkcji dla definiowania zbioru operacji (funkcje i operatory) dla typu i dla ograniczenia dostępu do obiektów tego typu do tego zbioru działań. Kiedy to zrobi, programista szybko odkrywa, że udoskonalenia języka są potrzebne do wygodnej definicji i korzystania z nowych typów. Przeciążenie operatorów jest tego dobrym przykładem.

Inicjowanie i czyszczenie

Kiedy reprezentacja typu jest ukryta musi być dostarczony jakiś mechanizm dla użytkownika dla zainicjowania zmiennych tego typu. Prostym rozwiązanie jest wymaganie od użytkownika wywołania pewnych funkcji dla inicjowania zmiennej przed jej użyciem. Na przykład:

```
class vector {
        int sz;
        int* v;

public:
        void init(int size);        // wywołujemy init dla inicjowania sz i v
                                    // przed pierwszym użyciem vector
        // …
};
vector v;
// nie używamy tu v
v.init(10);
// tu używamy v
```

Jest to podatne na błędy i nieeleganckie. Lepszym rozwiązaniem jest zezwolenie projektantowi typu dostarczenie rozpoznawalnej funkcji dla wykonania zainicjowania. Przy takiej funkcji, alokacja i inicjalizowanie zmiennej staje się pojedynczą operacją (często nazywana instancją lub konstrukcją) zamiast dwóch odrębnych działań. Taka funkcja inicjalizacyjna jest często nazywana konstruktorem. W przypadku gdzie konstrukcja obiektów typu jest nietrywialna, często potrzebujemy komplementarnej operacji czyszczenia obiektów po ich ostatnim użyciu. W C++ taka funkcja czyszcząca jest nazywana destruktorem. Rozważmy typ vector:

```
class vector {
```

```
        int sz;                 // liczba elementów
        int* v;                 // wskaźnik do liczb całkowitych
public:
        vector(int);            // konstruktor
        ~vector();              // destruktor
        Int& operator[ ] (int index);   // operator indeksu
};
```

Konstruktor vector może być zdefiniowany dla zaalokowania przestrzeni tak:

```
vector : : vector (int s)
{
        if (s <=0) błąd ("zły rozmiar wektora");
        sz = s;
        v = new int [s];        // alokowanie tablicy "s" liczb całkowitych
}
```

Destruktor vector zwalnia używaną pamięć:

```
vector : :  ~vector ()
{
        delete v;               //zwalnianie pamięci wskazywanej przez v
}
```

C++ nie wspiera czyszczenia pamięci. Jest to jednak kompensowane przez zezwolenie typowi na utrzymanie własnego zarządzania pamięcią bez konieczności interwencji użytkownika. Jest to powszechnie używane przez mechanizm konstruktora/destruktora, ale zastosowania tego mechanizmu nie są powiązane z zarządzaniem pamięcią.

Przypisanie i inicjalizacja

Kontrola tworzenia i niszczenia obiektów jest wystarczającą dla wielu typów, ale nie dla wszystkich, Może również być konieczna kontrola wszystkich operacji kopiowania. Rozważmy klasę vector:

```
vector v1(100);
vector v2 = v1;         //tworzymy nowy wektor v2 inicjowany przez v1
v1 = v2;                // przypisujemy v2 do v1
```

Musi być możliwe zdefiniowanie znaczenie inicjalizacji v2 i przypisania do v1. Alternatywnie powinno być możliwe zakazanie takiej operacji kopiowania; najlepiej, aby obie alternatywy powinny być dostępne. Na przykład:

```
class vector {
        int*  v;
        int  sz;
public:
        // …
        void operator= (const vector&);          // przypisanie
        vector(const vector&);                   // inicjalizacja
};
```

określa, że operacje zdefiniowane przez użytkownika powinny być używane do interpretacji przypisania i inicjalizacji vector. Przypisanie może być zdefiniowane tak:

```
vector : : operator=(const vector& a)                    //sprawdza rozmiar i kopiuje elementy
{
        if (sz != a.sz) błąd ("zły rozmiar wektora dla = ");
        for (int i = 0); i <sz ; i++) v[i] = a.v[i];
}
```

Ponieważ operacja przypisania opiera się na "starej wartości" wektora będącego przypisanym, operacja inicjalizacji musi być inna. Na przykład:

```
vector : : vector (const vector& a)                      // inicjalizacja wektora z innego wektora
{
        sz = a.sz;                                       //ten sam rozmiar
        v = new int [sz];                        // alokacja elementu tablicy
        for (int i = 0; i < sz, i++) v[i] = a.v[i]       // kopiowanie elementów
}
```

W C++, konstruktor kopiowania, np. X (const X&) definiuje wszystkie inicjowane obiekty typu X innym obiektem typu X. Dodatkowo jawnie inicjowane konstruktory kopiowania są używane do obsługi argumentów przekazywanych "przez wartość" a funkcja zwraca wartości. W C++ przypisanie obiektu klasy X może być zakazane przez zadeklarowanie przypisania private:

```
class x {
```

```
        void operator = (const X&);          // tylko składowe X może
        X(const X&);                         // kopiować X
        // …
public:
        // …
};
```

Ada nie wspiera konstruktorów, destruktorów, przeładowania przypisania lub kontroli definiowanej przez użytkownika przekazywania argumentu i zwracania z funkcji. To ogranicza typ klas, jakie można definiować i wymusza na programiście "techniki ukrywania danych"; to znaczy, użytkownik musi stworzyć i używać menadżerów modułu typu niż właściwego typu.

Typy parametryzowane

Dlaczego chcesz zdefiniować wektor liczb całkowitych? Użytkownik zwykle potrzebuje elementy wektora pewnego nieznanego typu do napisania typu wektora. W konsekwencji typ wektora powinien być wyrażony w taki sposób, że pobiera typ elementu, jako argument:

```
template<class T> class vector {          //elementy wektora typu T
        T* v;
        Int sz;
public:
        vector (int s)

        {
                if (s <= 0) błąd    ("zły rozmiar wektora");
                v - new T [sz = s];          // alokujemy tablice "s", "T"
        }
        T& operator [ ] (int i);
        int size () {return sz;}
        // …
};
```

template określa rodzinę typów wygenerowanych przez specjalny argument(-y). Wektory określonych typów mogą być teraz definiowane i używane:

```
vector<int> v1(100);                 // v1 jest wektorem 100 liczb całkowitych
```

```
vector<complex> v2(200);                    // v2 jest wektor 200 liczb zespolonych

v2[i] = complex(v1[x],v1[y]);
```

Ada Clu , ML i C++ wspierają typy parametryzowane.. Tam nie trzeba kosztów czasu wykonania w porównaniu z klasą gdzie wszystkie typy związane są bezpośrednio. Problem z typami parametryzowanymi jest taka, że każda instancja tworzy typ niezależny. Na przykład, typ vector<char>jest niepowiązany z typem vector<complex>. Idealnie chciałoby się wyrazić i wykorzystać wspólność typów generowanych z tego samego typu parametryzowanego. Na przykład, zarówno vector<char> i vector<complex> ma funkcje size (), która jest niezależna od parametru typu. Możliwe jest, ale nie trywialne, wydedukowanie tego z definicji klasy wektor a potem pozwala na zastosowanie size () na zastosowanie do dowolnego wektora. Interpretowany lub dynamicznie kompilowany język (taki jak Smalltalk) lub język, który wspiera zarówno typy parametryzowane i dziedziczenie (taki jak C++) ma tutaj przewagę.

Obsługa błędów

Przy rozwoju programów, zwłaszcza, gdy szeroko stosujemy biblioteki, ważne stają się standardy obsługi błędów (lub ogólniej: "wyjątkowe okoliczności"). Ada, Algol68, Clu i C++ każdy wspiera standardowy sposób obsługi wyjątków. Rozważmy ponownie przykład vector:

```
class vector {
        // …
        class range { };            //typ będący używany dla wyjątków
};
        if (i < 0 || sz <= 1) throw range ();
        return v[i];
}
```

Zamiast wywołania funkcji błędu, vector : : operator [ ] () może wywołać kod obsługi błędów. Powoduje to, że stos wywołań będzie rozwiązany dopóki obsługa błędów dla vector : : range nie zostanie znaleziona. Obsługa wyjątków może być zdefiniowana dla określonego bloku:

```
void f(int i) {
        try {                       // wyjątki w tym bloku try są obsługiwane przez
                                    // obsługę błędów zdefiniowane poniżej
                vector v(i) ;
                // …
                v[i] = 7; // powoduje wyjątek vector : : range
```

```
        // …
                int i = g ()        // może powodować wyjątek vector : : range
        }
        catch (vector : : range) {
                error("f() : błąd zakresu wektora");
        }
}
```

Jest wiele sposobów definiowana wyjątków i zachowania obsługi błędów. Słabe wykorzystanie obsługi błędów może być poważnym drenażem wydajności czasu uruchamiania i przenośności implementacji języka. Obsługa wyjątków C++ może być implementowana tak, że kod nie jest uruchamianym chyba, że jest wywołany wyjątek lub przenośność między implementacjami C przez (domyślne) użycie funkcji biblioteki standardowej C stejmp() i longimp ()

Konwersja typów

Konwersje typu definiowanych przez użytkownika, takie jak z liczb zmiennoprzecinkowych do liczb zespolonych implikowaną przez konstruktor complex(double), okazały się niezwykle przydatne w C++. Takie konwersje mogą być stosowane wyraźnie lub programista może polegać na kompilatorze i dodać je pośrednio w razie potrzeby i jednoznacznie:

```
complex a = complex(1);
complex b = 1;                  // 1 : -> complex (1)
a = b+complex(2);
a = b+2;                        // 2  : -> complex(2)
```

Konwersja typów zdefiniowanych przez użytkownika została wprowadzona w C++, ponieważ mieszane tryby arytmetyczne jest normą w językach dla pracy numerycznej a ponieważ większość typów zdefiniowanych przez użytkownika używana dla "obliczeń" (na przykład macierze, łańcuchu

znaków i adresowanie maszynowe) mają naturalne odwzorowanie do i/lub innych typów. Koercja okazała się szczególnie przydatna z punktu widzenia organizacji programu.

```
complex a = 2;
complex b= a+2;                 // interpretowane jako operator+ (a, complex(2))
b= 2+a;                 // interpretowane jako operator+(complex(2),a)
```

Tylko jedna funkcja jest konieczna dla interpretacji działania "+" a te dwa operandy są obsługiwane identycznie przez typ systemu. Co więcej, klasa complex jest zapisana bez konieczności modyfikacji pojęcia liczb całkowitych pozwalając na gładką i naturalną integrację tych dwóch koncepcji. Jest to kontrast do "czystego systemu obiektowego" gdzie działania będą interpretowane jak to:

```
a+2;  // a.operator+(2)
2+a; // 2.operator+ (a)
```

Modyfikują klasę integer, aby uczynić 2+a poprawnym. Modyfikacja istniejącego kodu powinna być unikana w miarę możliwości, kiedy dodajemy nowe obiekty do systemu. Zazwyczaj programowanie obiektowe oferuje wysokiej, jakości funkcje dla dodawania do systemu bez modyfikacji istniejącego kodu. W tym przypadku jednak, lepszego rozwiązania dostarcza abstrakcja danych

Iteratory

Twierdzi się, że język wspierający abstrakcję danych musi zapewnić sposób definiowania struktur sterujących. W szczególności, mechanizm, który pozwala użytkownikowi na określenie pętli na elementach pewnego typu zawierających elementy są często potrzebne. Musi to być osiągnięte bez wymuszenia na użytkowniku zależności od szczegółów implementacjo typu definiowanego przez użytkownika. Biorąc pod uwagę wystarczająco silny mechanizm dla definiowania nowych typów oraz możliwości przeciążania operatorów, może być obsługiwany bez oddzielnego mechanizmu definiowania struktur sterujących. Dla wektora, definiowanie iteratorów nie jest konieczne, ponieważ porządek jest dostępny dla użytkownika poprzez wskaźniki. Jest kilka możliwych styli iteratorów.

```
class vector_iterator{
vector& v'
public:
vector_iterator(vector& r) { i = 0; v = r;}
int operator () () { returm i < v/size() ? v.elem(i++) : 0;}
};
```

vector_iterator może teraz być zadeklarowany i używany dla vector tak:

```
void f(vector& v)
{
        vector_iterator next(v);
        int i;
        while i=next () ) print (i);
}
```

Więcej niż jeden iterator może być aktywny dla pojedynczego obiektu w danym czasie, a typ może mieć kilka różnych typów iteratorów zdefiniowanych dla niego, więc mogą być wykonywane różne rodzaje iteracji. Iterator jest raczej prostą strukturą sterującą. Bardziej ogólny mechanizm moż e być również zdefiniowany. Na przykład, standardowa biblioteka C++ dostarcza klasy współprogramu. Dla wielu typów "kontenera", takich jak vector, można uniknąć wprowadzania oddzielny typ iteratora przez zdefiniowanie mechanizmu iteracji, jako część samego typu. Vector może być zdefiniowany, aby miał "bieżący element":

```
class vector {
        int* v;
        int sz;
        int current;
public:
        // ...
        int next () {return (current < sz) ? v[current++] : 0; }
        int prev () {return (0 <= -- current) ? v[current] : 0' }
};
```

Wtedy iteracja może być wykonywana tak:

```
vector v(sz);
int i;
while (i = v.next () 0 print (i);
```

To rozwiązanie nie jest tak ogólne jak rozwiązanie iteratora, ale unika kosztów w ważnym specjalnym przypadku gdzie tylko jeden rodzaj iteracji jest konieczny i gdzie tylko jedna iteracja w danym czasie jest konieczna dla vector. Jeśli jest to konieczne, można zastosować bardziej ogólne rozwiązanie. Zwróć uwagę, że "proste" rozwiązania wymagają większej ostrożności od projektanta klasy kontenera niż rozwiązanie iteratora. Technika typu iteratora może być również używana do definiowania iteratorów, które mogą być ograniczone do kilku różnych typów kontenerów, zatem dostarcza mechanizmu dla iteracji przez różne typy kontenerów z pojedynczym typem operatora

Wiele implementacji

Podstawowym mechanizmem dla wsparcia programowania obiektowe, klasy pochodne i funkcje wirtualne mogą być używane dla wsparcia abstrakcji danych przez zezwolenie na kilka różnych implementacji dla danego typu. Rozważmy ponownie przykład stosu:

```
template<classT>
        class stack {
        public:
                virtual void push(T) = 0;          //czysta funkcja wirtualne
                virtual T pop() = 0;                        // czysta funkcja wirtualna
        };
```

Notacja =0 określa, że żadna definicja nie jest wymagana dla funkcji wirtualne i, że klasa jest abstrakcyjna, to znaczy, że klasa może być używana jedynie, jako klasa bazowa. To pozwala na użycie stosu a nie jego tworzenie:

```
stack<cat> s;                    //błąd : stack jest abstrakcyjny
void jakaś_funkcja(stack<cat>& s , cat kitty)     //ok
{
        s.push(kitty);
        cat c2 = s.pop();
        // …
}
```

Ponieważ żadna reprezentacja jest określona w interfejsie stosu, jego użytkownicy są izolowani od szczegółów implementacyjnych. Możemy teraz dostarczyć kilka różnych implementacji stosu. Na przykład, możemy dostarczyć zaimplementowanego stosu w tablicy

```
template<class T>
    class astack : public stack<T> {
        // rzeczywista reprezentacja obiektu stack
        // w tym przypadku tablica
        // …
    public:
        astack(int size);
        ~astack();
        void push(T);
        T pop ();
    };
```

A gdzie indziej zaimplementowany stos używa połączonej listy:

```
template<class T>
    class lstack : public stack<T> {
        // …
    };
```

Teraz możemy stworzyć i używać stosów:

```
void g()
{
    lstack<cat> s1(100);
    astack<cat> s2(100);
    cat ginger;
    cat snowball;
    some_function(s1, ginger);
    some_function(s2, snowball);
}
```

Tylko kreator stacks, g (), musi martwić się o różne rodzaje stacks, użytkownik jakaś_funkcja () jest całkowicie izolowany od takich szczegółów. Ceną takiej elastyczności jest taki, że każde działanie na takim typie musi być funkcją wirtualną.

Problemy z implementacją

Wsparcie potrzebne dla abstrakcji danych jest przede wszystkim dostarczane w postaci funkcji języka zaimplementowanej przez kompilator. Jednak, typy sparametryzowane są najlepiej implementowane ze wsparciem linkera z pewną wiedzą od semantyce języka, a obsługa wyjątków wymaga wsparcia ze

środowiska czasu uruchomienia. Obie mogą by implementowane w celu spełnienia surowych kryteriów dla szybkości czasu kompilacji i wydajności bez kompromisów ogólności czy wygody programisty. Zdolne do definiowania typów, programy w większym stopniu zależą od typów z bibliotek (a nie tylko tych opisanych w instrukcji języka). To oczywiście stawia większe wymagania obiektom dla wyrażania tego, co jest wstawione lub pobierane z biblioteki, funkcji dla znajdowania tego, co zawiera biblioteka, funkcji dla określania, jaka część biblioteki jest wykorzystywana przez program itd. Dla języków kompilowanych, funkcje dla obliczania konieczna minimalna kompilacja po zmianie staje się ważna. Istotne jest to, że linker/loader – z odpowiednią pomocą kompilatora - jest zdolny doprowadzić program do pamięci dla wykonania bez przynoszenia dużej ilości powiązanego, ale nieużywanego kodu.

4.Wsparcie dla programowania obiektowego

Podstawowe wsparcie dla programisty piszącego programy obiektowe składa się z mechanizmu klasy z dziedziczeniem i mechanizmu, który pozwala wywołać funkcje składowe zależne od rzeczywistego typu obiektu (w przypadku, gdy rzeczywisty typ jest nieznany w czasie kompilacji). Przy projektowaniu funkcji składowej, krytyczny jest mechanizm zgłoszeniowy. Dodatkowo, ważne są funkcje wspierające techniki abstrakcji danych, ponieważ argumenty dla danych abstrakcyjnych i dla ich udoskonalenia wspierające eleganckie zastosowanie typów są równie poprawne, kiedy dostępne jest wsparcie dla programowania obiektowego. Sukces obu technik zależy od konstrukcji typów oraz łatwości, elastyczności i wydajności takich typów. Programowanie obiektowe pozwala typom zdefiniowanym przez użytkownika być bardziej elastycznymi i ogólniejszymi niż te zaprojektowane do zastosowania tylko z technikami abstrakcji danych.

Mechanizmy zgłoszeniowe

Kluczową funkcją języka wspierającą programowanie obiektowe jest mechanizm, przez który funkcja jest wywoływana do danego obiektu. Na przykłady przy danym wskaźniku p, jak wywołać obsługę p -> f(arg)? Istnieje wiele opcji wyboru. W językach takich jak C++ czy Simula, gdzie używana jest statyczna kontrola typu, typ systemu może być wykorzystywany dla wyboru między różnymi mechanizmami zgłoszeniowymi. W C++ są dostępne dwie alternatywy:

1.Normalne wywołanie funkcji: funkcja składowa będąca wywoływaną jest określana w czasie kompilacji (przez wgląd w tablicę symboli kompilatora) i wywołana przy zastosowaniu standardowej funkcji mechanizmu wywołania z argumentem dodanym dla identyfikacji obiektu, dla którego funkcja jest wywoływana. Gdzie "standardowe wywołanie funkcji" nie jest uważane za skuteczne na tyle, aby programista mógł zadeklarować funkcję inline a kompilator będzie próbował rozszerzyć inline jej ciało. W ten sposób, można osiągnąć sprawność rozwinięcia makra bez pogorszenia semantyki funkcji standardowej. Optymalizacja ta jest równie cenna jak wsparcie abstrakcji danych.

2.Wywołanie funkcji wirtualnej: funkcja może być wywołania zależnie od typu obiektu, dla którego jest wywoływana. Ten typ nie może generalnie być określony w czasie uruchamiania. Zazwyczaj wskaźnik p będzie pewną klasą bazową B a obiekt będzie obiektem pewnej klasy pochodnej D. Mechanizm wywołania musi spojrzeć na obiekt i znaleźć pewne informacje tam umieszczone przez kompilator dla określenia, jaka funkcja f będzie wywołana. Kiedy taka funkcja jest znaleziona, powiedzmy D : : f, może być wywołana przy zastosowaniu mechanizmu opisanego powyżej. Nazwa f jest w czasie kompilacji konwertowana do indeksu do tablicy wskaźników do funkcji. Ten wirtualny mechanizm wywołania może być postrzegany, jako wydajny mechanizm "normalnego wywołanie

funkcji". W standardowej implementacji C++, jest używanych tylko pięć dodatkowych odniesień do pamięci. W językach ze słabym statycznym sprawdzaniem typów musimy zastosować inny mechanizm. To, co jest robione w języku takim jak Smaltalk to przechowywanie listy nazw wszystkich

funkcji składowych (metod) klasy, więc mogą być znajdowane w czasie wykonywania

3.Wywołanie metody: Najpierw jest znajdowana właściwa tablica nazw metod przez zbadanie obiektu wskazywanego przez p. W tej tabeli (lub zbiorze tabel) ciąg znakowy "f" jest wyszukiwany, aby zobaczyć czy ten obiekt ma f (). Jeśli f () jest znaleziona, jest wywoływana; w przeciwnym razie ma miejsce obsługa błędów. To wyszukiwanie różni się od wyszukiwania wykonywanego w czasie

kompilacji w statycznie sprawdzanych językach, w którym wywołanie metody używa metody tabeli dla aktualnego obiektu.

Wywołanie metody jest niewydajne w porównaniu z wywołaniem funkcji wirtualnej, ale bardziej elastyczna. Ponieważ statyczna kontrola typu argumentów zazwyczaj nie może być wykonane dla wywołania metody, użyte metody muszą być wspierane przez dynamiczną kontrolę typu.

Kontrola typu

Przykład kształtu pokazał potęgę funkcji wirtualnych. Co mechanizm wywołania metody robi dla ciebie? Możesz próbować wywołać dowolną metodę dla dowolnego obiektu. Możliwość wywołania dowolnej metody dla dowolnego obiektu pozwala projektantowi bibliotek ogólnego przeznaczenia na przerzucenie odpowiedzialności za kontrole typu na użytkownika.. Upraszcza to projektowanie bibliotek. Jednak wtedy użytkownik odpowiada za błędne typy, jak te:

```
// zakładamy dynamiczną kontrolę typu
// *** NIE C++ ***
Stack s;            // Stos może przechowywać wskaźniki di obiektów dowolnego typu
cs.push (new Saab900);
cs.push(new Saab37B);

cs.pop () -> takeoff();          //dobrze: Saab37B jest samolotem
cs.pop() ->takeogg();            //Ups! Błąd czasu uruchamiania: Saab900 to samochód
                                 // samochód nie ma metody takeoff
```

Próba użycia car, jako plane będzie wykryte przez obsługę komunikatu i właściwa obsługa błędów będzie wywołana. Jednak jest to pocieszające, kiedy użytkownik jest również programistom. Nieobecność statycznej kontroli typu nie zagwarantuje, że błąd tej klasy nie jest obecny w systemie dostarczanym końcowemu użytkownikowi. Połączenie klas sparametryzowanych i użycie funkcji wirtualnych może dostarczyć elastyczności, łatwość projektowania i łatwe użycie bibliotek zaprojektowanych z metodą wyszukiwania bez osłabiania statycznej kontroli typów lub ponoszenie znacznych kosztów w czasie uruchamiania (w czasie lub przestrzeni). Na przykład:

```
stack<plane*>cs;
```

```
cs.push(new Saab900);          // błąd czasu kompilacji

cs.push(new Saab37B);

cs.pop() ->takeoff();                    // dobrze : Saab 37B jest samolotem

cs.pop() ->takeoff();
```

Użycie statycznej kontroli typu i wywołania funkcji wirtualnej prowadzi do całkiem innego stylu programowania, który wykonuje dynamiczną kontrolę typu i wywołanie metody. Na przykład, klasa Simula i C++ określa stały interfejs do zbioru obiektów (dowolna klasa pochodna), podczas gdy klasa

Smalltalk określa inicjujący zbiór działań dla obiektów (dowolnej podklasy). Innymi słowy, klasa Smalltalk jest minimalną specyfikacją a użytkownik jest zwolniony z próby działania nieokreślonego, podczas gdy klasa C++ jest specyfikacją dokładną a użytkownik gwarantuje, że jedynie działania określone w deklaracji klasy będzie akceptowana przez kompilator.

Dziedziczenie

Rozważmy język mający pewną postać metody wyszukiwania bez posiadania mechanizmu dziedziczenia. Czy można mówić, że język wspiera programowanie obiektowe? Myślę, że nie. Można robić ciekawe rzeczy z metodą tabeli dla dostosowania zachowań obiektów. Jedna, aby uniknąć chaosu, musi istnieć jakiś systematyczny sposób kojarzenia metod i struktur danych zakładających ich obiektową reprezentację. Aby umożliwić użytkownikowi obiektu zrozumieć, jakiego rodzaju zachowania się spodziewać, także musi być jakiś standardowy sposób wyrażenia tego, co jest wspólne dla różnych zachowań obiektu, jaki może przyjąć. Ten "systematyczny i standardowy sposób" będzie mechanizmem dziedziczenia. Rozważmy język mający mechanizm dziedziczenia bez funkcji wirtualnych lub metod. Czy język będzie wspierał programowanie obiektowe? Myślę, że nie : przykład kształtu nie ma dobrego rozwiązania w takim języku. Jednak taki język byłby bardziej wydajny niż "zwykły" język z abstrakcją danych. To twierdzenie jest potwierdzone obserwacją, że wiele programów Simula i C++ jest strukturyzowanych przy użyciu hierarchii klas bez funkcji wirtualnych. Możliwość wyrażenia wspólności jest wyjątkowo silnym narzędziem. Na przykład, problem powiązany z koniecznością posiadania wspólnej reprezentacji wszystkich kształtów można rozwiązać. Żadna unia nie będzie konieczna. Jednak, w przypadku braku funkcji wirtualnych, programista będzie się musiał uciec do użycia "pól typów" dla określenia rzeczywistego typu obiektów, więc problemy z brakiem modularności pozostanie. Oznacza to, że klasa pochodna (subklasa) jest ważnym narzędziem programistycznym samym w sobie. Może być użyte dla wsparcia programowania obiektowego, ale ma szersze zastosowanie. Jest to szczególnie prawdziwe, jeśli określa użycie dziedziczenia w programowaniu obiektowym z myślą, że klasa bazowa wyraża ogólne pojęcie, z którego uszczegóławiane są klasy pochodne. Ta idea oddaje tylko część ekspresji mocy dziedziczenia, ale jest silnie wspierane przez języki gdzie każda funkcja składowa jest wirtualna (lub metodą). Biorąc pod uwagę kontrolę tego co jest dziedziczone, klasa pochodna może być silnym narzędziem dla tworzenia nowych typów. Dana klasa , dziedziczona może być używana dla dodania i/lub odjęcia funkcji. Relacja klasy wynikowej do bazowej nie zawsze może być całkowicie opisana z punktu widzenia szczegółowości; faktoring jest lepszym terminem. Derywacja jest innym narzędziem w rękach programisty i nie ma niezawodnego sposobu przewidzenia jak ma być używane.

Dziedziczenie wielokrotne

Kiedy klasa A i bazą dla klasy B, B dziedziczy atrybuty z A; to znaczy B jest A oprócz tego, co ma swojego. Biorąc pod uwagę to wyjaśnienie wydaje się oczywiste, że może być przydatne mieć klasę B dziedziczoną z dwóch klas bazowych A1 i A2. Nazywa się wielokrotnym dziedziczeniem. Dość standardowym przykładem użycia wielokrotnego będzie dostarczenie dwóch klas bibliotecznych i zadania dla przedstawienia obiektów pod kontrolą menadżera wyświetlania i współprogramów pod kontrolą harmonogramu, odpowiednio. Programista może potem stworzyć klasy takie jak:

```
class my_displayed_task : public displayed, public task {
        // …
};
class my_task    : public task {                // nie wyświetlane
        // …
};
class my_displayed : public displayed {  // żadnego zadania
        // …
};
```

Użycie (tylko) pojedynczego dziedziczenia tylko dwa z tych trzech wyborów będzie otwarty dla programisty. Prowadzi to albo do replikacji kodu albo utraty elastyczności - a zazwyczaj obu. W C++ ten przykład może być obsługiwany jak pokazano powyżej bez znaczących kosztów (w czasie lub przestrzeni) w porównaniu do pojedynczego dziedziczenia i bez poświęcenia statycznej kontroli typu. Niejasności są obsługiwane w czasie kompilacji

```
class A { public : void f (); /* … */ };
class B { public : void f (); /* … */ };
class C{ public A , public B /* … */ };
vodi g(C* P)
{
        p -> f ();// błąd : niejasność
}
```

W tym C++ różni się od obiektowego dialektu Lisp , który wspiera wielokrotne dziedziczenie. W tym dialekcie Lisp niejasności są rozwiązywane prze rozważenie porządku znaczących deklaracji, przez rozważenie obiektów o tej samej nazwie w różnych klasach bazowych, lub przez połączenie metod o tej same nazwie w klasach bazowych w bardziej złożone metody wyższych klas. W C++. Zazwyczaj rozwiązujemy niejasności przez dodanie funkcji:

```
class C : public A, public B {
        // …
```

```
public:
        void f ();
        // …
};
void f ()
{
        // ….
        A : : f ();

        B : : f () ;
}
```

Oprócz tej prostej koncepcji niezależnego wielokrotnego dziedziczenia wydaje się, że potrzeba bardziej ogólnego mechanizmu wyrażenia zależności między klasami w wielokrotnym dziedziczeniu. W C++, wymóg, że subobiekt, powinien być współużytkowany przez wszystkie inne subobiekty w obiekcie klasy jest wyrażony przez mechanizm wirtualnej klasy bazowej:

```
class W { /* … */ } ;        // okno
class Bwindow : public virtual W { // okno z obramowaniem
        // …
};
class  Mwindow : public virtula W { // okno z menu
        // …
};
class BMW  : public Bwindow, public Mwindow {
        // okno z obramowaniem i menu
        // …
};
```

Tu (pojedynczy) subobiekt window jest współużytkowany przez subobiekty Bwindow i Bwindow z BMW. Dialekt Lisp dostarcza koncepcji połączenia metod dla łatwego programowania używając takiej skomplikowanej hierarchii klas. C++ nie.

Hermetyzacja

Rozważmy składową klasy (albo składową danych albo funkcję składową), która musi być chroniona przed "nieautoryzowanym dostępem". Jaki wybór może być rozsądny dla ograniczenia zbioru funkcji,

które mogą być dostępne dla tej składowej? "Oczywista" odpowiedź dla języka wspierającego programowanie obiektowe to "wszystkie działania zdefiniowane dla obiektu", tzn. wszystkie funkcje składowe. Nie oczywista implikacja tej odpowiedzi jest taka, że nie może być całkowitej i skończonej listy wszystkich funkcji, które mogą mieć dostęp do składowej chronionej, ponieważ można zawsze dodać inną przez wyprowadzenie nowej klasy z chronionej klasy składowej i zdefiniowanie funkcji składowej klasy pochodnej. To pojęcie łączy duży stopień ochrony przed wypadkiem (ponieważ nie jest łatwo zdefiniować nową klasę pochodną „przez przypadek") z elastycznością potrzebną "narzędziom budowlanym" używającym hierarchii klas (ponieważ może przyznać sobie dostęp do chronionych składowych przez dziedziczenie klas). Na przykład:

```
class Window {
        // …
protected:
        Rectangle inside;
        // …
public:
        // …
};
class Dumb_terminal : Window {
        // …
public:
        void prompt();
        // …
};
```

Tu Windows określa inside, jako protected tak, więc klasy takie jak Dumb_terminal mogą odczytać ją i odkryć, jaką częścią obszaru Windows może manipulować. Niestety, "oczywista" odpowiedź dla języka zorientowanego na abstrakcję danych jest inna: „lista funkcji, które potrzebują dostępu w deklaracji klasy" Nie ma nic specjalnego w tych funkcjach. W szczególności, nie mogą być funkcjami składowymi. Funkcje nie składowe z dostępem do prywatnych klas składowych jest nazywana friend w C++. Powyższa klasa complex została zdefiniowana przy użyciu friend. Czasami ważne jest, że funkcja może być określona, jako friend w więcej niż jednej klasy. Mając pełną listę składowych i friends dostępnych jest największą zaletą, kiedy próbujemy zrozumieć zachowanie typu i szczególnie, kiedy chcemy je zmodyfikować. Hermetyzacja gwałtownie wzrasta na znaczeniu wraz z wielkością programu i z liczbą i geograficznym rozproszeniem jego użytkowników.

Problem implementacji

Wsparcie potrzebne dla programowania obiektowego jest przede wszystkim dostarczane przez system czasu uruchamiania i środowisko programistyczne. Jednym z powodów jest to, że programowanie obiektowe bazuje na poprawkach języka już zepchnięte do ich granic dla wspierania abstrakcji danych tak, że relatywnie kilka dodatków jest konieczne. Zastosowanie programowania obiektowego zaciera różnicę między językiem programowania i jego dalszego otoczenia.

5.Ograniczenie do doskonałości

Głównym problemem z językiem zdefiniowanym dla wykorzystania technik ukrywania danych, abstrakcji danych i programowania obiektowego jest to, że język programowania ogólnego przeznaczenia musi:

1.Uruchamiać się na tradycyjnych maszynach

2.Współistnieć z tradycyjnymi systemami operacyjnymi

3.Rywalizować z tradycyjnymi językami programowania pod względem wydajności w czasie wykonywania

4.Radzić sobie z każdym głównym obszarem aplikacji.

Oznacza to, że obiekty muszą być dostępne dla efektywnej pracy numerycznej (arytmetyka zmiennoprzecinkowej bez kosztów, które sprawiają, że pojawia się atrakcyjny Fortran), i że muszą być dostępne dla dostępu do pamięci w sposób, który pozwala na napisanie sterowników. Musi również istnieć możliwość zapisu wywołania, które są zgodne z często dość dziwnymi standardami wymaganymi dla tradycyjnych interfejsów systemów operacyjnych. Dodatkowo, powinna być możliwość wywołania funkcji napisanych w innych językach obiektowych i dla funkcji napisanych w języku obiektowym wywoływanych a programu napisanego w innym języku. Inną implikacją jest to, że język obiektowy nie może całkowicie polegać na mechanizmach, które nie mogą być skutecznie realizowane w tradycyjnych architektach i nadal oczekują zastosowania w językach ogólnego przeznaczenia. Bardzo ogólna implementacja wywołania metody może być obowiązkowa chyba, że są alternatywne sposoby ubiegania się o usługę. Podobnie, odzyskiwanie pamięci może stać się wąskim

Gardłem wydajności i przenośności. Większość języków obiektowych stosuje odzyskiwanie pamięci dla uproszczenia zadania programiście i zmniejszyć złożoność języka i jego kompilatora. Jednakże, powinno być możliwe zastosowanie odzyskiwanie pamięci w niekrytycznych obszarach zachowując kontrolę nad wykorzystanie pamięci w miejscach gdzie ma to znaczenie. Jako alternatywę, można mieć język bez odzyskiwania pamięci, a potem zapewnić wystarczająca siłę wyrazu umożliwiającą

projektowanie typów, które zarządzają swoją własną pamięcią. C jest przykładem tego. Obsługa wyjątków i współbieżność są innymi potencjalnymi obszarami problemowymi. Każda funkcja która jest najlepiej implementowana za pomocą linkera może stać się problemem przenośności. Alternatywą posiadania funkcji "niskopoziomowych" w języku jest obsługa głównych obszarów aplikacji przy zastosowaniu języków "niskopoziomowych".

6.Podsumowanie

Programowanie obiektowe jest programowaniem przy wykorzystaniu dziedziczenia. Abstrakcja danych jest programowanie przy zastosowaniu typów definiowanych przez użytkownika. Z kilkoma wyjątkami, programowanie obiektowe może i powinno być podzbiorem abstrakcji danych. Te techniki potrzebują właściwego wsparcia dla bycia efektywnymi. Abstrakcja danych przede wszystkim potrzebuje wsparcia w postaci funkcji języka a programowanie obiektowe wymaga dalszego wsparcia ze strony środowiska programistycznego. Aby być ogólnego przeznaczenia, język wspierający abstrakcję danych lub programowanie obiektowe, musi umożliwiać wykorzystanie tradycyjnego sprzętu.