Tehnici inovative de dezvoltare a programelor – programarea generica

Prof.dr. Dorin IRIMESCU
Universitatea "Politehnica" Bucuresti
e-mail: irimescu@cs.pub.ro

Programarea generica, aparuta ca o extensie a programarii orientate obiect, ofera o solutie simpla la cerintele de standardizare si reutilizare în domeniul tehnologiilor de realizare a programelor. Folosirea conceptelor programarii generice si a bibliotecii STL poate sa creasca semnificativ productivitatea scrierii programelor, daca utilizatorul cunoaste structura bibliotecii si foloseste un stil de programare adecvat. Cadrul de dezvoltare definit de STL, format în esenta din structuri de date fundamentale si algoritmi generici, ofera un potential enorm pentru a beneficia de standardizare si reutilizare, mai ales dupa includerea bibliotecii STL în noul standard al limbajului C++. Lucrarea prezinta o aplicatie de verifcare a corectitudinii scrierii pentru limba româna care foloseste componente software parametrizate din STL. Solutia se remarca prin concizia si simplitatea programului. Sunt evidentiate, totodata performantele bune atinse prin utilizarea algoritmilor generici STL. În viitorul apropiat se asteapta ca prin efortul reunit al programatorilor de pretutindeni, sa se extinda dezvoltarea de componente generice si accesibilitatea lor.
Cuvinte cheie: *programare generica, STL, reutilizare software.*

Evolutii în tehnologiile de dezvoltare a programelor

Progresele în domeniul tehnologiilor electronice folosite în realizarea componentelor hardware impun cerinte tot mai mari asupra programelor si conduc la cresterea complexitatii sistemelor software. Aceasta situatie a fost evidentiata înca din anii '60, când decalajul metodelor de dezvoltare a programelor a devenit atât de pregnant încât a început sa se vorbeasca despre o "criza" în domeniul software. Criza software se manifesta prin esecul a numeroase proiecte de dezvoltare a sistemelor informatice de mari dimensiuni. Insuccesele se datoreaza atât unor cauze de ordin tehnic cât si managerial. Un articol celebru din anii '60, "Mass Produced Software Components", propunea ca dezvoltarea programelor sa se faca pornind de la niste componente reutilizabile, sau blocuri constructive standardizate. Acestea ar fi similare circuitelor electronice integrate care intra în alcatuirea echipamentelor hardware ale calculatorului. La mai bine de 30 de ani de la aparitia articolului suntem astazi înca departe de a atinge viziunea prefigurata de Dong McIlroy, autorul articolului.

Introducerea în anii '80 a metodelor de proiectare si dezvoltare orientate obiect a promis sa ofere solutia realizarii rapide a unor programe mari, la un pret rezonabil. S-a sperat ca metodologia orientata obiect va revolutiona dezvoltarea sistemelor informatice complexe. Exista în prezent un numar important de medii de dezvoltare orientate obiect, sisteme vizuale, produse pentru dezvoltarea de software asistata de calculator – CASE, produse de reengineering etc. prea putin folosite în transformarea sistemelor informatice existente. Analiza si programarea orientata obiect au dobândit un rol important în scrierea programelor, fiind larg raspândite în practica si în învatamântul de specialitate. În acelasi timp, au devenit tot mai vizibile si partile lor slabe: posibilitati limitate de reutilizare si adaptabilitate scazuta. Un studiu din 1995 arata ca numarul de aplicatii majore care au fost implementate folosind instrumente orientate obiect este redus [7]. Mai mult, abordarea orintata obiect contine o serie de dezavantaje intrinseci [8]. Cunoscuta revista BYTE atragea atentia asupra deficientelor abordarii orientate obiect printr-un titlu surprinzator de pe

coperta numarului din mai 1995: "Calculul orientat obiect a esuat, reuseste dezvoltarea cu componente...".
Chiar daca afirmatia exagereaza esecul tehnologiilor orientate obiect este clar ca, în ciuda progreselor în domeniul limbajelor de programare si al instrumentelor de dezvoltare, costurile scrierii, întretinerii si modificarii unor sisteme informatice sunt foarte mari. O serie de tehnici noi, aflate în diverse stadii de elaborare si testare, urmaresc sa faca dezvoltarea programelor mai eficienta si mai flexibila. Iata câteva dintre acestea:

- cadre de dezvoltare – frameworks
- modele de proiecte – design patterns
- programarea generica
- componente
- programarea orientata aspect
- programarea orientata subiect
- programarea intentionala
- programarea generativa.

Generalitate si eficienta prin metode generice

Programarea generica reprezinta o forma de reutilizare care foloseste tipuri generice si algoritmi generici pentru a fi adaptati unor cerinte concrete. Prin "genericitate" se pun în evidenta similitudinile structurale si de comportament ale obiectelor si se evita scrierea unor clase si metode al caror cod se repeta aproape identic. Programarea generica este rezultatul cercetarilor efectuate timp de peste 15 ani de catre Alexander Stepanov, cu diferiti colaboratori si folosind diverse limbaje de programare. Obiectivul central al cercetarii a fost scrierea unor algoritmi cât mai generali, fara a influenta negativ performantele programului. Se urmareste ca un algoritm dezvoltat sa lucreze asupra tuturor tipurilor de date si structurilor pentru care are sens operatia respectiva [1],[6].
Înca din anii '70, Stepanov a observat ca o serie de algoritmi ramân nemodificati atunci când se schimba structura de date folosita pentru reprezentarea datelor. De exemplu, un algoritm de cautare functioneaza la fel atunci când elementele pe care le analizam sunt memorate sub forma unui tablou, a unei liste înlantuite s.a.m.d. Structura de date trebuie sa îndeplineasca doar câteva proprietati semantice de baza, cum ar fi: posibilitatea de acces la un element al structurii, posibilitatea de a trece de la un element al structurii la urmatorul, indicarea sfârsitului domeniului de valori. Acest lucru ne permite sa separam un algoritm de modul particular de implementare a datelor, fara a afecta performantele algoritmilor. O descoperire practica evidentiata de catre Stepanov arata ca exista sute de algoritmi ce pot fi descrisi în termenii unor structuri de date si algoritmi generici.
În ceea ce priveste performantele algoritmilor generici, un algoritm este considerat "relativ eficient" daca este la fel de performant ca un algoritm non-generic scris în acelasi limbaj de programare si "absolut eficient" daca performantele sale egaleaza pe cele ale algoritmului scris în limbaj de asamblare. Limbajul de programare care s-a dovedit cel mai adecvat pentru atingerea dezideratelor de performanta este C++. Încercarile facute de catre Stepanov în Ada si Scheme nu au permis egalarea performantelor similare primitivelor scrise în limbajul de baza, adica nu s-a atins nici macar eficienta relativa. În schimb, limbajul C++ este capabil sa depaseasca eficienta relativa, tinzând catre eficienta absoluta, demonstrând ca eficienta si generalitatea nu se exclud reciproc [1],[6].

Bibliteca STL – parte a standardului ANSI / ISO C++

Observatiile lui Stepanov au condus la aparitia unui nou concept în programare. În 1985 Stepanov a dezvoltat o biblioteca generica în limbaj Ada. La acea data limbajul C++ nu dispunea de sabloane pentru implementarea noului stil de programare. Abia în 1992 a fost realizata prima versiune a bibliotecii STL – Standard Template Library pentru C++, în cadrul unui proiect condus de Stepanov la firma Hewlet Packard [7],[5],[6].
Importanta STL si a programarii generice a crescut considerabil în urma includerii

STL în noul standard al limbajului C++. Comitetul pentru standardizare a recunoscut potentialul STL, care devine parte obligatorie a compilatoarelor C++ elaborate de catre orice producator. S-a deschis în acest mod calea prin care programarea generica sa ajunga la nivelul programatorului de rând [2], [3].

STL este o biblioteca de uz general care contine structuri de date fundamentale si algoritmi generici. Se estimeaza ca STL va creste productivitatea activitatii de programare prin reutilizarea unor componente standard într-un domeniu ce ofera un potential urias de a beneficia de standardizare, cel al structurilor de date si algoritmilor asociati [4], [5], [6].

STL este alcatuita din sase componente:

- structuri de date sau "containere" - obiecte ce pot sa contina si sa gestioneze colectii de alte obiecte
- algoritmi – functii generice care folosesc iteratori ca sa poata sa lucreze asupra mai multor tipuri de structuri de date
- iteratori – asigura modalitatea de acces a unui algoritm la elmentele unei structuri de date pentru examinarea si traversarea structurii; sunt similari pointerilor din C
- obiecte functii – sunt generalizari ale conceptului de functie; ele iau locul pointerilor la functii din programarea C traditionala
- adaptori – modifica interfata unei componente
- alocatorii – inglobeaza datele privind modelul de memorie corespunzator masinii.

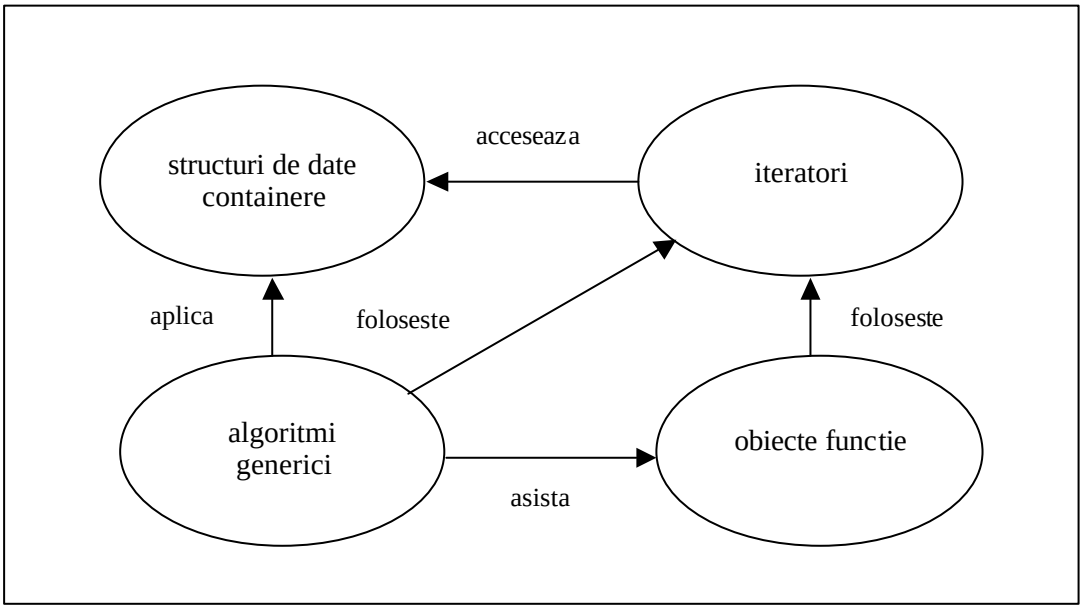

Fig. 1. Componente centrale ale bibliotecii STL

STL are la baza o descompunere ortogonala a spatiului componentelor prin decuplarea algoritmilor de structurile de date pe care le prelucreaza. Spre deosebire de clasele container traditionale, structurile de date STL contin un numar redus de metode, care asigura operatii pentru crearea, copierea, stergerea structurii, împreuna cu operatii pentru introducerea si înlaturarea de elemente. Operatiile de cautare, ordonare etc. sunt realizate prin intermediul algoritmilor generici. Aceasta organizare face biblioteca STL mai flexibila.

Elementul esential în proiectarea STL îl reprezinta iteratorii care fac legatura dintre algoritmi si structurile de date. Iteratorii asigura flexibilitatea bibliotecii STL. În loc de a scrie un algoritm pentru un anumit tip de container (structura de date), algoritmii sunt dezvoltati pentru o categorie de iteratori. Aceasta strategie face posibila folosirea aceluiasi algoritm pentru mai

multe tipuri de containere. Iteratorii dispun de operatii de incrementare si dereferentiere si sunt similari pointerilor inteligenti. STL garanteaza ca algoritmii scrisi în conformitate cu aceasta interfata abstracta vor functiona corect [3], [4], [5].

O alta noutate în stilul de lucru STL îl reprezinta obiectele functie. Ele sunt folosite în mod frecvent ca parametru generic al unui algoritm pentru a indica o operatie ce se executa pentru anumite elemente din structura de date (sau pentru toate). Obiectele functie pot fi considerate ca o generalizare a functiilor obisnuite din C/C++. Un obiect functie este o clasa pentru care s-a supraîncarcat operatorul de apel al unei functii *operator( )*. Adeseori, clasele functie nu contin date componente si nici costructori/destructori, în afara celor furnizati în mod implicit de catre compilator. Un obiect functie este transmis catre un algoritm tot asa cum un pointer la o functie este al unei functii C. Faptul ca functia poate fi expandata *inline* face ca mecanismul sa fie foarte eficient.

Aplicatie – verificarea corectitudinii scrierii pentru limba româna

Continutul unui document text urmeaza sa fie verificat din punct de vedere al corectitudinii scrierii prin compararea fiecarui cuvânt cu termenii existenti într-un dictionar. Atât textul verificat cât si dictionarul sunt pastrate initial în fisiere pe disc. Programul trebuie sa afiseze fiecare cuvânt din document care nu este gasit în dictionar sau este scris gresit.

Aplicatia poate sa fie scrisa concis si eficient folosind algoritmi si structuri de date STL, asa cum este schitat în continuare. În pseudocod, prelucrarile pentru verificarea corectitudinii scrierii se exprima astfel:

copiaza fiecare cuvânt din text la iesirea standard

care nu se gaseste în dictionar

Termenii din dictionar pot fi adusi într-o structura de date de tip container secvential; ei trebuie sa fie ordonati pentru a permite efectuarea unor cautari de tip binar. În acest scop, programul foloseste pentru dictionar un vector de tip *string* si copiaza termenii dintr-un flux de intrare de tip fisier:

```
typedef vector<string> dictionar;
ifstream f_dictionar("dictionar.txt");
copy ( istream_iterator<string>(f_dictionar),
       istream_iterator<string>( ),
       back_inserter(dictionar) );
```

Algoritmul STL *remove_copy_if* este folosit pentru a copia la iesirea standard *cout* toate elementele din fisierul document cuprinse între pozitiile [*first*, *last*) ale iteratorului, care nu îndeplinesc conditia *pred(*iterator)==true*. În cazul de fata, functia predicat verifica daca un cuvânt se gaseste sau nu în dictionar. Codul necesar este încorporat într-un obiect functie:

```
class cauta {
public:
    cauta ( bidirectional_iterator begin,
    bidirectional_iterator end ) : _begin(begin),
              _end(end) { }
    bool operator ( ) (const T&cuvant) {
    return binary_search ( _begin, _end, cuvant) ;
        }
private:
    bidirectional_iterator _begin;
    bidirectional_iterator _end;
};
```

Înainte de a compara cuvintele din document cu termenii existenti în dictionar, cele mai multe cuvinte sufera o prelucrare prealabila. Substantivele, verbele s.a.m.d. pot sa-si modifice terminatia conform particularitatilor limbii. Programatorul urmeaza sa-si concentreze eforturile asupra acestor aspecte, structurile de date si algoritmii fiind disponibili din STL.

Bibliografie

1. Stepanov, A.: "Part of the draft C++ standard, STL provides the framework for building generic, highly reusable algorithms and data structures", BYTE, October, 1995.

2. Stroustrup, B.: "A perspective on ISO C++", C++ Report 7,8, October, 1995, pp. 23-28.

3. STL site at Silicon Graphics Inc.: http://www.sgi.com/Technology/STL/

4. Mumit's STL Newbie guide: http://www.xraylith.wisc.edu/
5. HP Reference implementation of STL: ftp://butler.hlp.hp.com/stl,
sau ftp:// ftp.cs.rpi.edu/pub/stl
6. Musser, D.R., Saini, A.: "STL Tutorial and Reference Guide", Addison-Wesley, 1996.
7. Pfister, C.: "Bridging the gap between power and simplicity", Oberon Tribune, Vol.1, No.1, July 1995, http:// www.oberon.com
8. Krajnc, M.: "Why component-oriented programming ?", Oberon Microsystems, Zurich, Switzerland, 1997, http://www. oberon.com

Anexa: Program sursa

```
//programul a fost compilat folosind Microsoft Visual C++
#pragma warning(disable:4786) // suprima mesaje avertisment
//Aceasta este un bug al compilator. confirmat de Microsoft
#include <iostream>
#include <fstream>
#include <vector>
#include <iterator>
#include <algorithm>
#include <string>
using namespace std;
// obiect functie care implementeaza functia predicat
template <class bidirectional_iterator, class T>
class cauta {
        public:
                cauta(bidirectional_iterator begin,
bidirectional_iterator end):
          _begin(begin),_end(end){}
                bool operator() (const T &word) {
                        return binary_search(_begin, _end, word);
                }
        private:
                bidirectional_iterator _begin;
                bidirectional_iterator _end;
};
main() {
        ifstream ifile("f_dictionar.txt"); // fisier dictionar
        ifstream tfile("f_testat.txt");          // fisier testat
        typedef vector<string> d_type;       //defineste vector
        d_type dictionar;
        typedef istream_iterator<string> istr; // defin. iterator
        typedef ostream_iterator<string> ostr;
        copy(istr(ifile), istr(),                    // completeaza
                back_inserter(dictionar) );     // vectorul cu termenii din diction.
        typedef vector<string>::iterator d_iter;
        d_iter first=dictionar.begin();
        d_iter last=dictionar.end();
        remove_copy_if( istr(tfile), istr(), ostr(cout," "),
                        cauta< d_iter, d_type::value_type>
                        (first, last) );
        return 0;
}
```