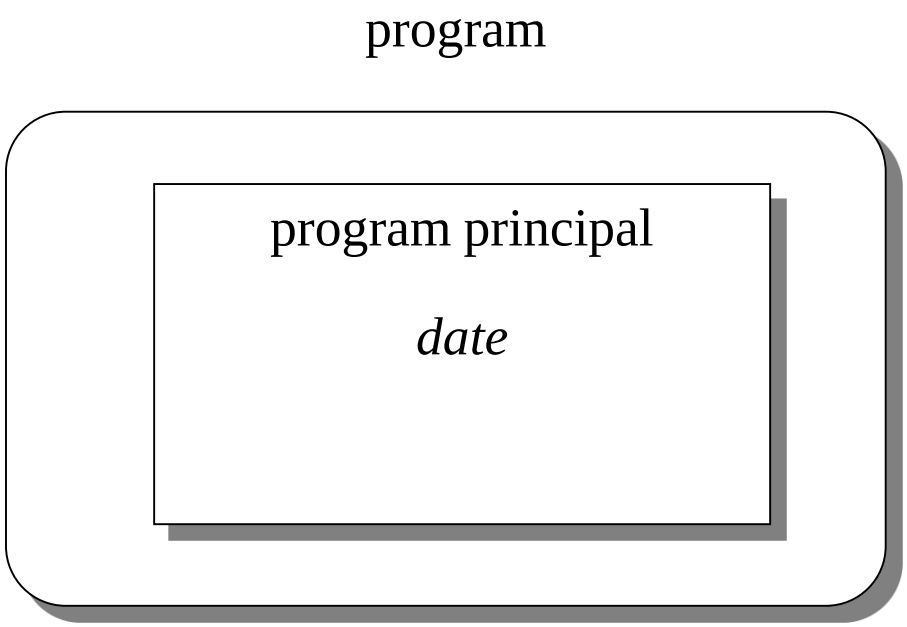
Programare orientată pe obiecte (1)

Cuprins

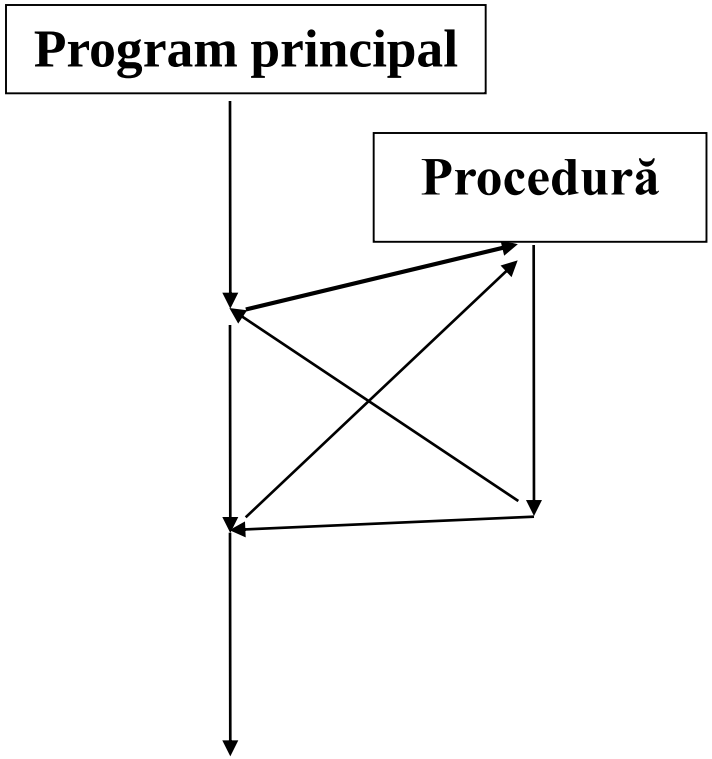
1 O trecere în revistă a tehnicilor de programare

1.1 Programare nestructurată



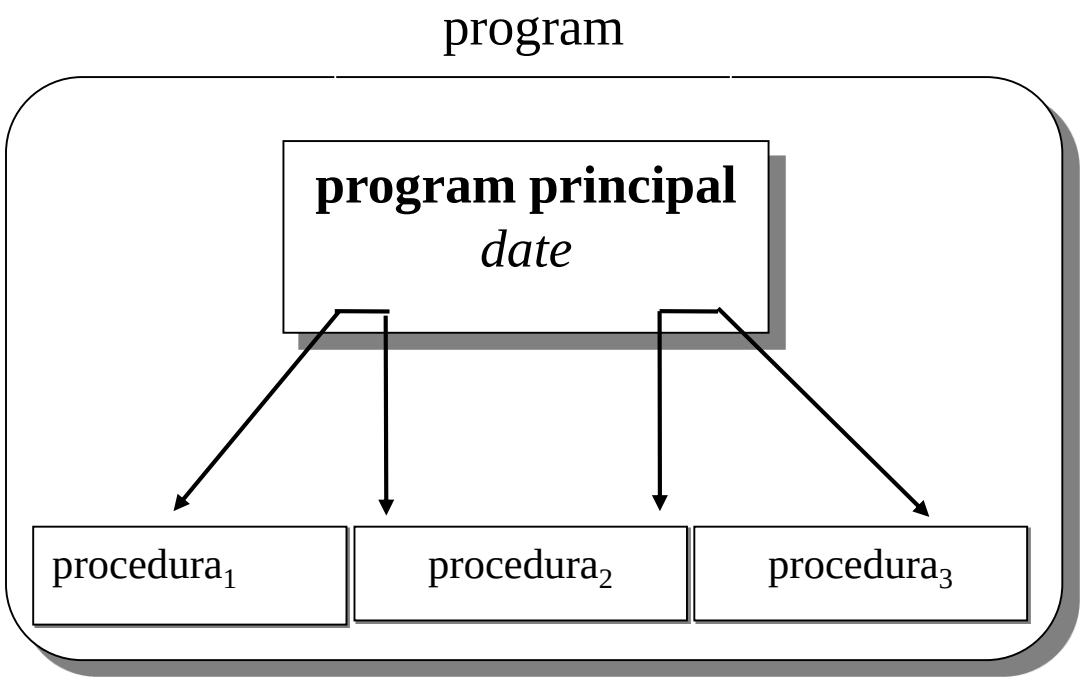
1.2 Programare procedurală

•Se grupează secvenţele de instrucţiuni folosite de mai multe ori într-un singur loc (procedură).
•O *apelare de procedură* este utilizată pentru a se invoca procedura.
•După ce secvenţa este prelucrată, execuţia programului continuă după poziţia în care a fost făcută apelarea.



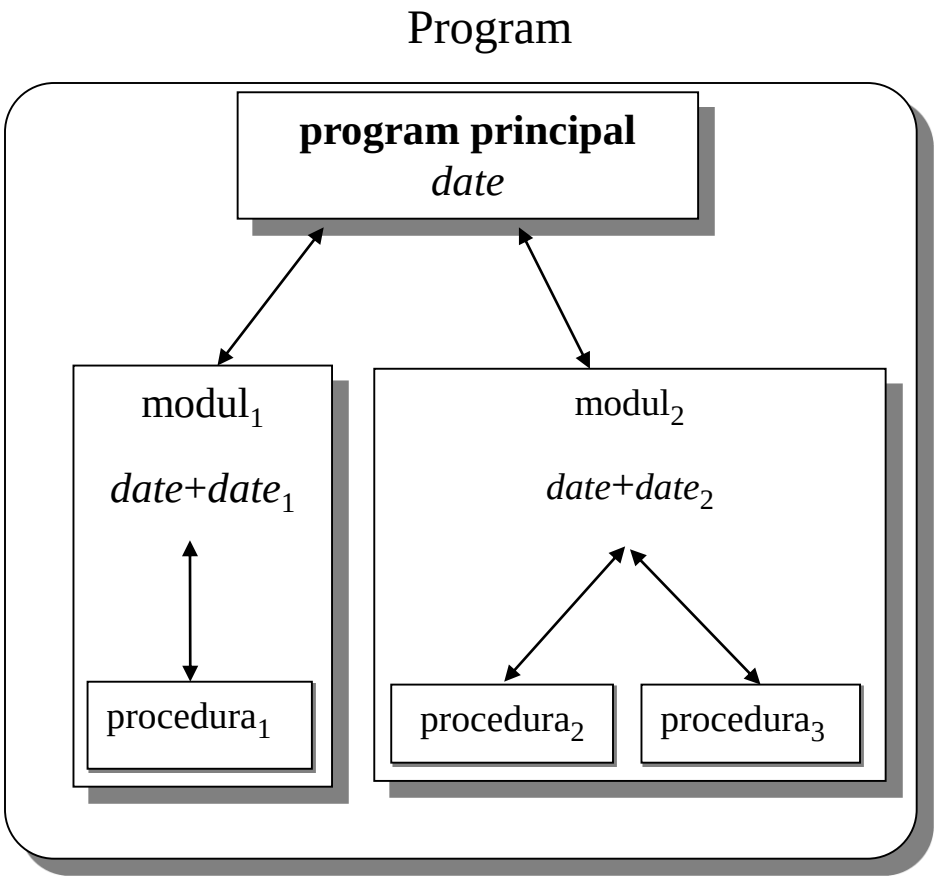
1.2 Programare procedurală (continuare)

- Un program poate fi considerat ca o secvenţă de apelări de procedură.
- *Fluxul de date* poate să fie ilustrat ca un graf ierarhic (arbore)



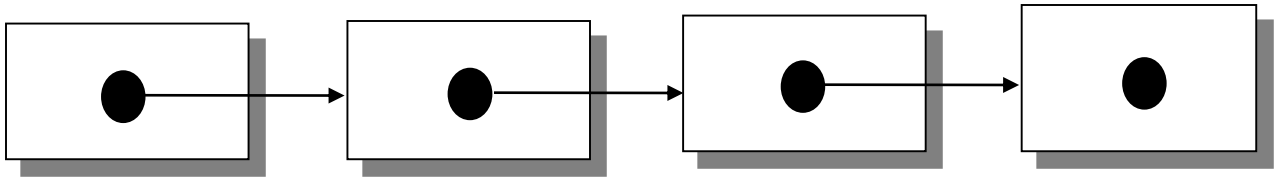
1.3 Programare modulară

•Proceduri care realizează împreună o anumită funcţionalitate sunt grupate în module *separate.*
•Un program este constituit din mai multe părţi mai mici care interacţionează prin intermediul apelărilor de procedură.



1.4 Un exemplu cu structuri de date

1.4.1 Lucrul cu liste simplu înlănţuite



- Listele simplu înlănţuite furnizează metode de acces pentru a *adăuga* un nou element şi pentru a *şterge* elementul din faţă.
- O listă se poate implementa într-un *modul* separat.
- Se scriu două fişiere: *definiţia interfeţei* şi *fişierul de implementare.*

1.4.1 Lucrul cu liste simplu înlănţuite (continuare)

```
/*
 * Definitia interfetei pentru un
   modul care implementeaza
   [PSEUDOCOD]
 * o lista simplu inlantuita pentru
   a memora date de orice tip
 */
 MODULE Lista-Simplu_Inlantuita-1
 BOOL list_initialize();
 BOOL list_append(ANY data);
 BOOL list_delete();
 list_end();
 ANY list_getFirst();
 ANY list_getNext();
 BOOL list_isEmpty();
 END Lista-Simplu_Inlantuita-1
```

Declarările din interfaţă descriu *ce* parte a informaţiei este disponibilă şi nu *cum* această parte devine disponibilă. Se *ascunde* informaţia privind implementarea în fişierul de implementare (principiu fundamental din ingineria programării). Aceasta face posibilă schimbarea implementării, de ex. utilizarea unui alt algoritm, fără să fie necesar să se schimbe alte module ale programului. Este posibil ca declarările din interfaţă să fie făcute într-un limbaj de programare, iar implementarea să fie făcută în alt limbaj de programare, dacă există o compatibilitate între cele două limbaje.

1.4.1 Lucrul cu liste simplu înlănţuite (continuare)

- *list_initialize() trebuie să se apeleze înainte de utilizarea listei* pentru a se iniţializa variabilele locale din modul.
- *append* (pentru adăugare) şi *delete* (pentru ştergere) implementează operaţiile de accesare şi modificare a structurii.
- *Observaţie:* toate procedurile din interfaţă întorc valori (al căror tip este specificat înaintea antetului declarării de procedură) deci pot fi utilizate ca *funcţii*. Tipul BOOL întors de unele dintre dintre ele are două valori şi este introdus pentru a modela valorile de adevăr TRUE şi FALSE, iar întoarcerea unei valori de acest tip de o funcţie are semnificaţia terminării cu succes sau cu eşec a execuţiei funcţiei.
- *list_append()* are un argument *date* de un tip arbitrar.Utilizăm un tip special ANY care ne permite să îi ataşăm orice tip de date (nu orice limbaj de programare are un astfel de tip; în C acesta poate fi introdus cu *pointeri de tip void*).
- *list_end()* trebuie să fie apelată când programul încetează să permită modulului să şteargă conţinutul variabilelor sale interne. De exemplu putem să eliberăm memoria alocată.

1.4.1 Lucrul cu liste simplu înlănţuite (continuare)

list_getFfirst() şi *list_getNext()* oferă un mecanism simplu de traversare prin listă, arătat mai jos:

```
ANY data;
data <- list_getFirst();
WHILE data IS VALID DO
faceCeva(data);
data <- list_getNext();
END
```

1.4.2 Lucrul cu mai multe liste

```
/*
 * Un modul de liste pentru mai mult de o lista.
 */

MODULE Lista-Simplu-Inlantuita-2

DECLARE TYPE reprez_lista_t;

reprez_lista_t list_create();

 list_destroy(reprez_lista_t this);

BOOL list_append(reprez_lista_t this, ANY data);

ANY list_getFirst(reprez_lista_t this);

ANY list_getNext(reprez_lista_t this);

BOOL list_isEmpty(reprez_lista_t this);

 END Lista-Simplu-Inlantuita-2;
```

Pentru a putea să gestioneze mai mult decât o listă modulul pentru liste trebuie reproiectat.

Se crează o nouă descriere a interfeţei care include acum o definiţie pentru un tip numit *reprezentant de listă*.

Reprezentantul este utilizat în orice procedură dată pentru a identifica în mod unic lista cu care se lucrează.

1.4.2 Lucrul cu mai multe liste (continuare)

• Utilizăm *DECLARE TYPE* pentru a introduce noul tip pe care îl numim *reprez_lista_t.* Nu specificăm cum este reprezentat şi implementat în mod efectiv acest tip.

• Şi de această dată *ascundem* detaliile de implementare în fişierul de implementare. Faţă de versiunea precedentă, în care ascundeam numai funcţii, respectiv proceduri, acum ascundem şi informaţiile pentru un tip de date definit de utilizator, numit *reprez_lista_t*.

• *list_create() se utilizează* pentru a se obţine reprezentantul unei noi liste (vidă).

• Fiecare procedură conţine acum argumentul special *this* care identifică chiar lista respectivă. Toate procedurile operează acum asupra acestui reprezentant în loc să opereze asupra unei liste globale a modulului.

• Acum se pot crea *obiecte listă.* Fiecare astfel de obiect poate fi identificat în mod unic prin reprezentantul său şi sunt aplicabile numai acele *metode* care sunt definite pentru a opera asupra acestui reprezentant.

1.5 Probleme specifice programării modulare

1.5.1 Creare / distrugere explicite

De câte se utilizează o listă trebuie să se declare explicit un reprezentant şi să se execute o apelare a *list_create()* pentru a obţine unul valid. După utilizarea listei trebuie să se apeleze în mod explicit *list_destroy()* pentru reprezentantul listei de care nu mai este nevoie. Utilizarea unei liste în interiorul unei proceduri, numită, de exemplu, foo(), trebuie făcută în următorul cadru:

```
PROCEDURE foo()
BEGIN
 reprez_lista_t lista_mea;
 lista_mea <- list_create();
 /* Se face ceva cu lista_mea */
...
 list_destroy(lista_mea);
END
```

Problemă

Un întreg este creat automat, iar, în unele limbaje de programare (de exemplu C) este şi distrus în mod automat. Unele compilatoare chiar iniţializează întregii nou creaţi cu o anumită valoare, de obicei 0 (zero).
“Obiectele" listă ar putea avea un tratament similar. Timpul de viaţă al unei liste este şi el definit de domeniul său, deci ea poate fi creată de îndată ce se intră în domeniu şi poate fi distrusă de îndată ce el este părăsit. Este util ca la momentul creării o listă să fie iniţializată ca vidă. Este deci de dorit să se poată defini o listă ca şi un întreg. Un cadru pentru aceasta ar putea arăta aşa:

```
PROCEDURE foo() BEGIN
reprez_lista_t lista_mea; /* Lista este
creata si
 initializata */
 /* Se face ceva cu lista_mea */
...
END /* lista_mea este distrusa */
```

1.5.2 Date şi operaţii decuplate

Decuplarea datelor şi operaţiilor duce la o structură bazată pe operaţii mai degrabă decât pe date: modulele grupează împreună operaţiile des folosite (cum sunt operaţiile *list_...()*) . Utilizăm aceste operaţii furnizându-le explicit datele asupra cărora ele trebuie să opereze. Structura rezultată pentru module este deci orientată spre operaţii, mai degrabă decât spre datele efective. Se poate spune că operaţiile definite specifică datele care să fie utilizate.

1.5.3 Lipsa precizării tipurilor

Utilizarea tipului special *ANY* (de exemplu, pentru a permite unei liste să conţină orice fel de date) implică faptul că tipul utilizat efectiv nu este stabilit de compilator, deci corectitudinea utilzării tipului nu poate fi verificată în cursul procesului de compilare. Iată un exemplu în care compilatorul nu poate să verifice corectitudinea:

```
PROCEDURE foo() BEGIN
 UnTipdeDate data1;
 UnAltTipdeDate data2;
 reprez_lista_t lista_mea;

 lista_mea <- list_create();
 list_append(lista_mea, data1);
 list_append(lista_mea, data2); /* Surpriza! */

...

 list_destroy(lista_mea);
 END
```

1.5.3 Lipsa precizării tipurilor (continuare)

Este de dorit să avem un mecanism care să ne permită să specificăm cu ce date va fi definită lista. Funcţia principală a listei este mereu aceeaşi, indiferent dacă ea conţine date privind persoane, numere, maşini, sau chiar liste. Deci este convenabil să declarăm o nouă listă cam aşa:

```
reprez_lista_t<Persoana> lista1; /* o lista de persoane */
reprez_lista_t<Masina> lista2; /* o lista de masini */
```

Rutinele corespunzătoare pentru liste ar întoarce automat tipurile corecte de date. Compilatorul ar putea să verifice automat corectitudinea.

1.5.4 Strategii şi reprezentare

Lucrul cu liste necesită operaţii pentru traversare. Pentru acest scop este definită o variabilă numită *cursor* care indică *elementul curent.* Aceasta implică o *strategie de traversare* care defineşte ordinea în care elementele structurii de date urmează să fie vizitate.

Pentru o structură de date simplă cum este cea de listă simplu înlănţuită se poate concepe doar o singură strategie de **traversare**: plecând de la cel mai din stânga element se vizitează succesiv vecinii din dreapta până când se ajunge la ultimul element. Structuri de date mai complexe, cum sunt de exemplu arborii, pot fi traversate utilizând mai multe strategii. Lucrurile pot fi chiar mai complicate, uneori strategiile de traversare depinzând de contextul particular în care o structură de date este utilizată. În consecinţă, are sens să se separe reprezentarea efectivă sau forma structurii de date de strategia sa de traversare.

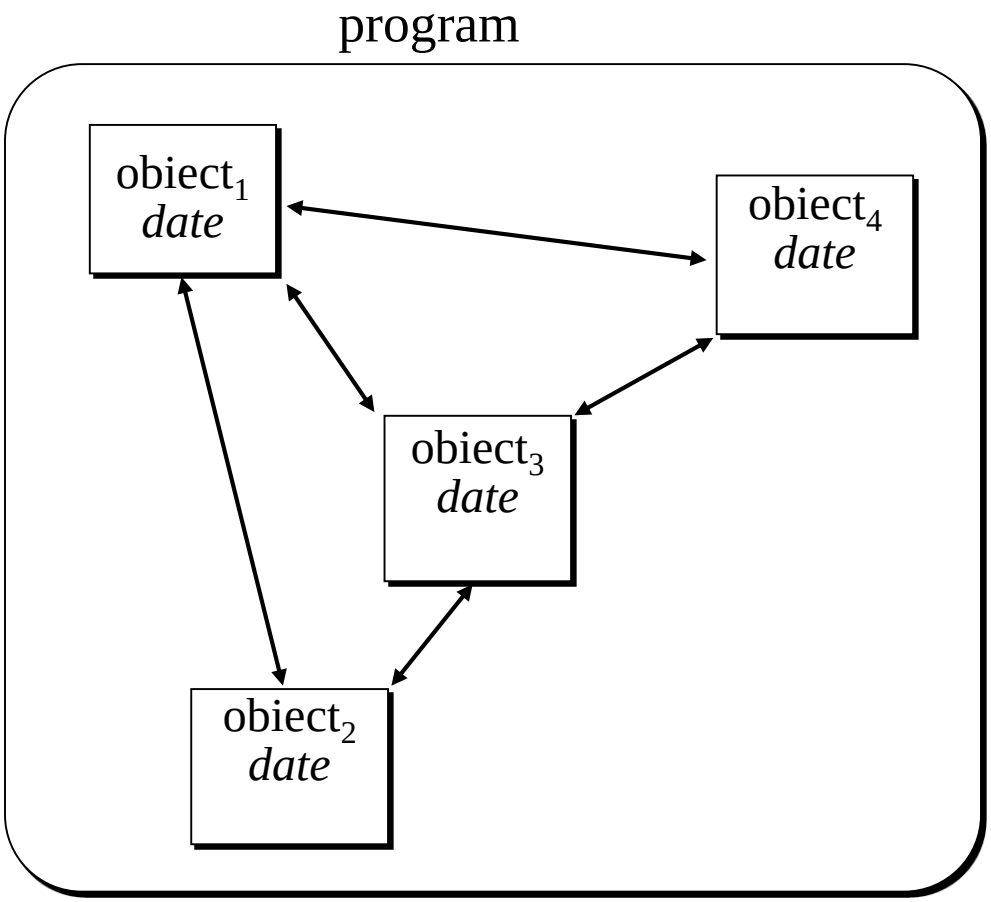
Ceea ce am arătat cu privire la strategia de traversare este valabil şi pentru alte feluri de strategii. De exemplu, **inserţia** poate fi făcută astfel încât să existe sau nu o ordonare a elementelor.

1.6 Programare orientată pe obiecte (POO)

1.6 POO – introducere

Programarea orientată pe obiecte rezolvă unele din problemele menţionate.

Spre deosebire de alte tehnici, avem acum o reţea de *obiecte* care interacţionează, fiecare având starea sa proprie.



1.6 POO – introducere (continuare)

Considerând din nou exemplul privitor la liste, în programarea modulară trebuie să se creeze şi să se distrugă în mod explicit reprezentantele de liste. Apoi se utilizează procedurile modulului pentru modificarea fiecăreia dintre reprezentante.

Spre deosebire de aceasta, în POO putem avea smultan oricât de multe obiecte listă este necesar. În loc să se apeleze o procedură căreia să-i fie furnizată reprezentanta corectă a unei liste, se va trimite un *mesaj* respectivului obiect listă. Se poate spune că fiecare obiect implementează propriul său modul care să permită, de exemplu, ca mai multe liste să coexiste.

Fiecare obiect trebuie să se creeze şi să se distrugă în mod automat. În consecinţă, nu mai este nevoie să se apeleze în mod explicit o procedură de creare şi terminare.

POO nu este doar o tehnică modulară de programare mai sofisticată. Caracteristicile şi conceptele care vor fi introduse în capitolele următoare fac ca POO să fie cu adevărat o nouă tehnică de programare, care să permită creşterea eficienţei activităţilor de proiectare şi programare a aplicaţiilor.