Laborator 8

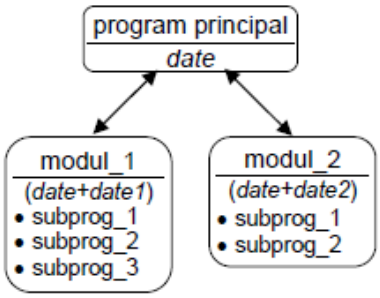
Principiile programării orientate pe obiecte

Evoluţia tehnicilor de programare

Programarea nestructurată (un program simplu, ce utilizează numai variabile globale); complicaţiile apar când prelucrarea devine mai amplă, iar datele se multiplică şi se diversifică.

Programarea procedurală (program principal deservit de subprograme cu parametri formali, variabile locale şi apeluri cu parametri efectivi); se obţin avantaje privind depanarea şi reutilizarea codului şi se aplică noi tehnici privind transferul parametrilor şi vizibilitatea variabilelor; complicaţiile apar atunci când la program sunt asignaţi doi sau mai mulţi programatori care nu pot lucra simultan pe un acelaşi fişier ce conţine codul sursă.

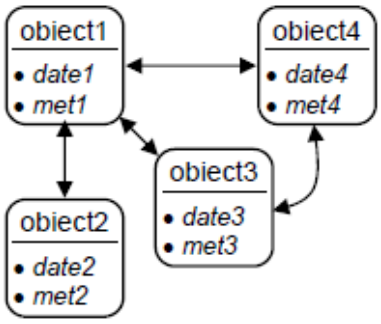
Programarea modulară (gruparea subprogramelor cu funcţionalităţi similare în module, implementate şi depanate separat); se obţin avantaje privind independenţa şi *încapsularea* (prin separarea zonei de implementare, păstrând vizibilitatea numai asupra zonei de interfaţă a modulului) şi se aplică tehnici de asociere a procedurilor cu datele pe care le manevrează, stabilind şi diferite reguli de acces la date şi la subprograme.



Se observă că modulele sunt „centrate” pe proceduri, acestea gestionând şi setul de date pe care le prelucrează (*date+date1* din figură). Dacă, de exemplu, dorim să avem mai multe seturi diferite de date, toate înzestrate comportamental cu procedurile din modulul modul_1, această arhitectură de aplicaţie nu este avantajoasă.

Programarea orientată obiect – POO (programe cu noi *tipuri* ce integrează atât datele, cât şi metodele asociate creării, prelucrării şi distrugerii acestor date); se obţin avantaje prin *abstractizarea* programării (programul nu mai este o succesiune de prelucrări, ci un ansamblu de obiecte care prind

viaţă, au diverse proprietăţi, sunt capabile de acţiuni specifice şi care interacţionează în cadrul programului); intervin tehnici noi privind instanţierea, derivarea şi polimorfismul tipurilor obiectuale.



Tipuri de date obiectuale. Încapsulare

Definiţie: Un **tip de date abstract** (**ADT**) este o entitate caracterizată printr-o *structură de date* şi un *ansamblu de operaţii* aplicabile acestor date.

Considerând, în rezolvarea unei probleme de gestiune a accesului utilizatorilor la un anumit site, tipul abstract *USER*, vom observă că sunt multe date ce caracterizează un utilizator Internet.

Totuşi se va ţine cont doar de datele semnificative pentru problema dată. Astfel, „culoarea ochilor" este irelevantă în acest caz, în timp ce „data naşterii" poate fi importantă. În aceeaşi idee, operaţii specifice ca „se înregistrează", „comandă on-line" pot fi relevante, în timp ce operaţia „mănâncă" nu este, în cazul nostru. Evident, nici nu se pun în discuţie date sau operaţii nespecifice („numărul de laturi" sau acţiunea „zboară").

Definiţie: Operaţiile care sunt accesibile din afara ADT formează **interfaţa** acesteia. Astfel, operaţii interne cum ar fi conversia datei de naştere la un număr standard calculat de la 01.01.1900 nu fac parte din interfaţa tipului de date abstract, în timp ce operaţia „plasează o comandă on-line" face parte, deoarece permite interacţiunea cu alte obiecte (SITE, STOC etc.).

Definiţie: Numim **instanţă** a unui tip de date abstract o „concretizare" a tipului respectiv, formată din valori efective ale datelor.

Definiţie: Un **tip de date obiectual** este un tip de date care implementează un tip de date abstract.

Definiţie: Vom numi **metode** operaţiile implementate în cadrul tipului de date abstract.

Definiţie: Numim **membri** ai unui tip de date obiectual datele şi metodele definite mai sus.

Folosirea unui tip de date obiectual tip presupune:

- existenţa definiţiei acestuia
- apelul metodelor
- accesul la date.

Exemplul 58:

Un exemplu de-acum clasic de tip de date abstract este STIVA. Ea poate avea ca date: numerele naturale din stivă, capacitatea stivei, vârful etc. Iar operaţiile specifice pot fi: introducerea în stivă (*push*) şi extragerea din stivă (*pop*). La implementarea tipului STIVA, vom defini o structură de date care să reţină valorile memorate în stivă şi câmpuri de date simple pentru: capacitate, număr de elemente etc. Vom mai defini metode (subprograme) capabile să creeze o stivă vidă, care să introducă o valoare în stivă, să extragă valoarea din vârful stivei, să testeze dacă stiva este vidă sau dacă stiva este plină etc.

Definiţie: Crearea unei instanţe noi a unui tip obiectual, presupune operaţii specifice de „construire" a noului obiect, metoda corespunzătoare purtând numele de **constructor**.

Definiţie: La desfiinţarea unei instanţe şi eliberarea spaţiului de memorie aferent datelor sale, se aplică o metodă specifică numită **destructor** (datorită tehnicii de supraîncărcare, limbaje de genul C++, Java şi C# permit existenţa mai multor constructori).

O aplicaţie ce utilizează tipul obiectual STIVA, va putea construi două sau mai multe stive (de cărţi de joc, de exemplu), le va umple cu valori distincte, va muta valori dintr-o stivă în alta după o anumită regulă desfiinţând orice stivă golită, până ce rămâne o singură stivă. De observat că toate aceste prelucrări recurg la datele, constructorul, destructorul şi la metodele din interfaţa tipului STIVA descris mai sus.

Definiţii: Principalul tip obiectual întâlnit în majoritatea mediilor de dezvoltare (Visual Basic, Delphi, C++, Java, C#) poartă numele de clasă (**class**). Există şi alte tipuri obiectuale (**struct, object**). O instanţă a unui tip obiectual poartă numele de **obiect**.

Definiţie: La implementare, datele şi metodele asociate trebuie să fie complet şi corect definite, astfel încât utilizatorul să nu fie nevoit să ţină cont de detalii ale acestei implementări. El va accesa datele, prin intermediul proprietăţilor şi va efectua operaţiile, prin intermediul metodelor puse la dispoziţie de tipul obiectual definit. Spunem că tipurile de date obiectuale respectă **principiul încapsulării**.

Astfel, programatorul ce utilizează un tip obiectual CONT (în bancă) nu trebuie să poarte grija modului cum sunt reprezentate în memorie datele referitoare la un cont sau a algoritmului prin care se realizează actualizarea soldului conform operaţiilor de depunere, extragere şi aplicare a dobânzilor. EL va utiliza unul sau mai multe conturi (instanţe ale tipului CONT), accesând proprietăţile şi metodele din interfaţă, realizatorul tipului obiectual asumându-şi acele griji în momentul definirii tipului CONT.

Permiţând extensia tipurilor de date abstracte, clasele pot avea la implementare:

- date şi metode caracteristice fiecărui obiect din clasă (**membri de tip instanţă**),
- date şi metode specifice clasei (**membri de tip clasă**).

Astfel, clasa STIVA poate beneficia, în plus, şi de date ale clasei cum ar fi: numărul de stive generate, numărul maxim sau numărul minim de componente ale stivelor existente etc.

Modificatorul **static** plasat la definirea unui membru al clasei face ca acela să fie un membru de clasă, nu unul de tip instanţă. Dacă în cazul membrilor nestatici, există câte un exemplar al membrului respectiv pentru fiecare instanţă a clasei, membrii statici sunt unici, fiind accesaţi în comun de toate instanţele clasei. Mai mult, membrii statici pot fi referiţi chiar şi fără a crea vreo instanţă a clasei respective.

Supraîncărcare

Deşi nu este o tehnică specifică programării orientată obiect, ea creează un anumit context pentru metodele ce formează o clasă şi modul în care acestea pot fi (ca orice subprogram) apelate.

Definiţie: Prin **supraîncărcare** se înţelege posibilitatea de a defini în acelaşi domeniu de vizibilitate mai multe funcţii cu acelaşi nume, dar cu parametri diferiţi ca tip şi/sau ca număr.

Definiţie: Ansamblul format din numele funcţiei şi lista sa de parametri reprezintă o modalitate unică de identificare numită **semnătură** sau **amprentă**.

Supraîncărcarea permite obţinerea unor efecte diferite ale apelului în contexte diferite

Capacitatea unor limbaje (este şi cazul limbajului C#) de a folosi ca „nume” al unui subprogram un operator, reprezintă supraîncărcarea operatorilor. Aceasta este o facilitate care „reduce” diferenţele dintre operarea la nivel abstract (cu DTA) şi apelul metodei ce realizează această operaţie la nivel de implementare obiectuală. Deşi ajută la sporirea expresivităţii codului, prin supraîncărcarea operatorilor şi metodelor se pot crea şi confuzii.

Apelul unei funcţii care beneficiază, prin supraîncărcare, de două sau mai multe semnături se realizează prin selecţia funcţiei a cărei semnătură se potriveşte cel mai bine cu lista de parametri efectivi (de la apel).

Astfel, poate fi definită metoda „comandă on-line” cu trei semnături diferite:

- comanda_online(cod_prod) cu un parametru întreg (desemnând comanda unui singur produs identificat prin cod_prod
- comanda_online(cod_prod,cantitate) cu primul parametru întreg şi celalalt real
- comanda_online(cod_prod,calitate) cu primul parametru întreg şi al-II-lea caracter.

Moştenire

Definiţie: Pentru tipurile de date obiectuale **class** este posibilă o operaţie de extindere sau specializare a comportamentului unei clase existente prin definirea unei clase noi ce moşteneşte datele şi metodele clasei de bază, cu această ocazie putând fi redefiniţi unii membri existenţi sau adăugaţi unii membri noi. Operaţia mai poartă numele de **derivare**.

Definiţii: Clasa din care se moşteneşte se mai numeşte clasă **de bază** sau **superclasă**.

Clasa care moşteneşte se numeşte **subclasă**, **clasă derivată** sau **clasă descendentă**.

Definiţie: Ca şi în Java, în C# o subclasă poate moşteni de la o singură superclasă, adică avem de-a face cu moştenire simplă; aceeaşi superclasă însă poate fi derivată în mai multe subclase distincte. O subclasă, la rândul ei, poate fi superclasă pentru o altă clasă derivată. O clasă de bază împreună cu toate clasele descendente (direct sau indirect) formează o **ierarhie de clase**. În C#, toate clasele moştenesc de la clasa de bază **Object**.

- În contextul mecanismelor de moştenire trebuie amintiţi modificatorii **abstract** şi **sealed** aplicaţi unei clase, modificatori ce obligă la şi respectiv se opun procesului de derivare. Astfel, o clasă abstractă trebuie obligatoriu derivată, deoarece direct din ea nu se pot obţine obiecte prin operaţia de instanţiere, în timp ce o clasă sigilată (**sealed**) nu mai poate fi derivată (e un fel de terminal în ierarhia claselor).

Definiţie: O **metodă abstractă** este o metodă pentru care nu este definită o implementare, aceasta urmând a fi realizată în clasele derivate din clasa curentă care trebuie să fie şi ea abstractă (virtuală pură, conform terminologiei din C++).

Definiţie: O **metodă sigilată** este o metodă care nu mai poate fi redefinită în clasele derivate din clasa curentă.

Polimorfism. Metode virtuale

Definiţie: Folosind o extensie a sensului etimologic, un obiect polimorfic este cel capabil să ia diferite forme, să se afle în diferite stări, să aibă comportamente diferite. **Polimorfismul obiectual**, care trebuie să fie abstract, se manifestă în lucrul cu obiecte din clase aparţinând unei ierarhii de clase, unde, prin redefinirea unor date sau metode, se obţin membri diferiţi având însă acelaşi nume.

Astfel, în cazul unei referiri obiectuale, se pune problema stabilirii datei sau metodei referite.

Comportamentul polimorfic este un element de flexibilitate care permite stabilirea contextuală, în mod dinamic, a membrului referit. Acest lucru este posibil doar în cazul limbajelor ce permit „legarea întârziată”. La limbajele cu „legare timpurie”, adresa la care se face un apel al unui subprogram se stabileşte la compilare. La limbajele cu legare întârziată, această adresă se stabileşte doar in momentul rulării, putându-se calcula distinct, în funcţie de contextul în care apare apelul.

Exemplul 59:

Dacă este definită clasa numită PIESA (de şah), cu metoda nestatică **muta (pozitie_initiala, pozitie_finala)**, atunci subclasele TURN şi PION trebuie să aibă metoda muta definită în mod diferit (pentru a implementa maniera specifică a pionului de a captura o piesă „en passant”, sau, într-o altă concepţie, metoda **muta** poate fi implementată la nivelul clasei PIESA şi redefinită la nivelul subclasei PION, pentru a particulariza acest tip de deplasare care capturează piesa peste care trece pionul în diagonală). Atunci, pentru un obiect T, aparţinând claselor derivate din PIESA, referirea la metoda **muta** pare nedefinită. Totuşi mecanismele POO permit

stabilirea, în momentul apelului, a clasei proxime căreia îi aparţine obiectul T şi apelarea metodei corespunzătore (mutare de pion sau tură sau altă piesă).

Pentru a permite acest mecanism, metodele care necesită o decizie contextuală (în momentul apelului), se declară ca metode virtuale (cu modificatorul **virtual**). În mod curent, în C# modificatorului **virtual** al funcţiei din clasa de bază, îi corespunde un specificator **override** al funcţiei din clasa derivată ce redefineşte funcţia din clasa de bază.

O metodă ne-virtuală nu este polimorfică şi, indiferent de clasa căreia îi aparţine obiectul, va fi invocată metoda din clasa de bază.

Principiile programării orientate pe obiecte

Ideea POO este de a crea programele ca o colecţie de obiecte, unităţi individuale de cod care interacţionează unele cu altele, în loc de simple liste de instrucţiuni sau de apeluri de proceduri.

Obiectele POO sunt, de obicei, reprezentări ale obiectelor din viaţa reală (*domeniul problemei*), astfel încât programele realizate prin tehnica POO sunt mai uşor de înţeles, de depanat şi de extins decât programele procedurale. Aceasta este adevărată mai ales în cazul proiectelor software complexe şi de dimensiuni mari.

Principiile POO sunt:

1. **abstractizarea** - principiu care permite identificarea caracteristicilor şi comportamentului obiectelor ce ţin nemijlocit de domeniul problemei. Rezultatul este un model. În urma abstractizării, entităţilor din domeniul problemei se definesc prin clase.
2. **încapsularea** – numită şi ascunderea de informaţii, este caracterizată prin 2 aspecte:

a. Gruparea comportamentelor şi caracteristicilor într-un tip abstract de date

b. Definirea nivelului de acces la datele unui obiect

3. **moştenirea** – organizează şi facilitează polimorfismul şi încapsularea permiţând definirea si crearea unor clase specializate plecând de la clase (generale) care sunt deja definite - acestea pot împărtăşi (şi extinde) comportamentul lor fără a fi nevoie de redefinirea aceluiaşi comportament.
4. **Polimorfismul** - posibilitatea mai multor obiecte dintr-o ierarhie de clase de a utiliza denumiri de metode cu acelaşi nume dar, cu un comportament diferit.