Acta et Commentationes, Sciences of Education, nr. 1(31), 2023 ISSN 1857-0623
p. 37-45 E-ISSN 2587-3636

CZU: 378.02:004.43 DOI: 10.36120/2587-3636.v31i1.37-45

TENDINȚE ACTUALE ÎN STUDIEREA PROGRAMĂRII ORIENTATE PE OBIECTE

Ala GASNAȘ, dr., conf. univ.
https://orcid.org/0000-0002-7174-7027
Catedra de Informatică și Tehnologii Informaționale
Universitatea Pedagogică de Stat „Ion Creangă”

Rezumat. Programarea orientată pe obiecte (POO) este o paradigmă care ne permite să scriem programe prin modelarea lucrurilor din lumea reală sub formă de clase și obiecte. Scopul principal al programării orientate pe obiecte este de a dezvolta aplicații care să integreze diverse atribute, cum ar fi reutilizarea, mentenabilitatea și siguranța în exploatare. În acest articol sunt identificate diverse probleme și se propun varii soluții concrete în scopul de a îmbunătăți procesul de predare și învățare a cursului de programare orientată pe obiecte.

Cuvinte-cheie: programare orientată pe obiecte, metode de predare, probleme de învățare; paradigmă orientată pe obiecte; concepte orientate pe obiecte; limbaje POO.

CURRENT TRENDS IN THE STUDY OF OBJECT-ORIENTED PROGRAMMING

Summary. Object-oriented programming (OOP) is a paradigm that allows us to write programs by modeling real-world things in the form of classes and objects. The main goal of object-oriented programming is to develop applications that integrate various attributes such as reusability, maintainability, and operational safety. Various problems were identified and various concrete solutions were proposed regarding the teaching and learning process of the object-oriented programming course.

Keywords: object-oriented programming, teaching methods, learning problems; object-oriented paradigm; object-oriented concepts; OOP languages.

1. Introducere

De mai bine de două decenii, paradigma orientată pe obiecte în programare a fost principala abordare în dezvoltarea de software. În domeniul informaticii, unul dintre cursurile care aplică această paradigmă este cursul *Programare orientată pe obiecte* (POO) [1, 2]. Scopul principal al POO este de a dezvolta un sistem software care să combine diverse atribute de calitate, cum ar fi reutilizarea, mentenabilitatea și siguranța în exploatare [3]. Acest sistem poate fi elaborat astfel încât să fie cât mai aproape de lumea reală, unde toate entitățile sunt tratate ca obiecte și fiecare obiect are caracteristici precum capacitatea de a realiza relații cu alte obiecte [4]. POO este o paradigmă de programare concepută pentru a reprezenta obiecte în blocuri, ținând cont de instanțele, clasele, proprietățile, metodele ș.a.m.d. conținute în fiecare dintre aceste blocuri [5]. În programarea orientată pe obiecte, un program este considerat a fi interacțiunea obiectelor. Fiecare obiect este o instanță a unei clase. Un obiect este caracterizat de stare (un set de variabile instanță) și funcționalitate (un set de metode).

Obiectele sunt entitățile de bază ale POO și sunt instanțe ale claselor, de aceea sunt necesare pentru dezvoltarea oricăror programe [6]. Obiectele acționează ca intermediari între programe și metodele și proprietățile acestora. Clasa este o schiță a unui anumit obiect, care conține proprietăți și metode și formează unitatea de bază a dezvoltării unei aplicații. Clasele sunt un element-cheie în dezvoltarea și întreținerea aplicațiilor POO [7] prin utilizarea proprietăților și metodelor asupra datelor. Clasele, la rândul lor, pot fi îmbunătățite prin utilizarea conceptului de moștenire și polimorfism.

În programarea modernă, abordarea orientată pe obiecte este utilizată pe scară largă, ceea ce permite îmbunătățirea calității programelor, creșterea productivității programatorului și a eficienței muncii în echipă. Stilul orientat pe obiecte este folosit în dezvoltarea unei game largi de aplicații, iar studenții trebuie să fie instruiți să aplice cunoștințele în situații reale pentru a extinde domeniul de aplicare posibilă a POO. Pentru a face acest lucru, se recomandă rezolvarea problemelor care au obiecte ale căror prototipuri sunt obiecte și structuri naturale din viața reală.

Totodată, în procesul de predare a programării orientate pe obiecte, se întâlnesc anumite dificultăți în înțelegerea conceptelor de bază.

Înainte de studierea POO, studenții sunt obișnuiți să dezvolte programe în care folosesc o abordare diferită – abordarea procedurală sau structurată. În acest caz, programul este reprezentat ca un set de proceduri și funcții-subrutine care execută un anumit bloc de cod cu datele de intrare necesare. Programarea procedurală sau structurată este potrivită pentru programele ușoare, fără structură complexă. Dar dacă blocurile de cod sunt mari și există sute de funcții, eforturile de editare sunt enorme. Prin urmare, ar trebui să fie regândită logica de programare. Spre deosebire de programarea procedurală sau structurată, programarea orientată pe obiecte permite efectuarea modificărilor o singură dată – modificarea obiectului. El este elementul-cheie al programului. Toate operațiunile sunt reprezentate ca o interacțiune între obiecte. În același timp, codul este citit și înțeles mai ușor, iar programul este executat mai ușor. Se știe că trecerea de la vechiul stil procedural imperativ de programare la stilul orientat pe obiecte este încă o sarcină dificilă. De asemenea, s-a observat că este nevoie de o anumită perioadă pentru ca studentul mediu să treacă de la o abordare procedurală sau structurată la una orientată pe obiecte [8].

2. Abordări ale învățării programării orientate pe obiecte

Programarea structurată și programarea orientată pe obiecte reprezintă, în mare parte, un mod de gândire. În primul caz, sarcina este descrisă ca un algoritm, în al doilea caz este văzută ca o interacțiune și o interconectare între obiecte. Dacă învățarea începe mai întâi cu programarea structurată, iar apoi este urmată de una orientată pe obiecte, va fi mai greu de a trece de la un mod de gândire la altul.

Pentru a începe studierea programării orientate pe obiecte, un student trebuie să cunoască bazele programării și să fie capabil să reprezinte soluția problemei ca program.

Pentru ca un student să stăpânească programarea orientată pe obiecte satisfăcător, este necesar ca el să posede următoarele cunoștințe și abilități [9]:

- ✓ Din bazele programării:
  - tipuri de date;
  - operații (inclusiv cele matematice);
  - structuri de control;
  - lucrul cu tipuri de date complexe;
  - intrarea și ieșirea datelor;
  - depanarea programului;
  - scrierea textului programului conform convențiilor limbajelor de programare.
- ✓ Din programarea structurată:
  - scrierea sau transformarea textului programului în forma corespunzătoare paradigmei programării structurate;
- ✓ Din programarea procedurală:
  - principii de împărțire a programelor în rutine;
  - modalități de schimb de date între programul principal și rutine;
  - variabile locale și globale;
  - parametri formali și actuali.

Deoarece majoritatea studenților sunt familiarizați cu programarea structurată, la început este bine ca aceștia să-și folosească cunoștințele obținute din cursurile anterioare prin realizarea unor lucruri deja cunoscute. Prin urmare, la orele inițiale ale disciplinei POO, inițial studenții sunt rugați să creeze și să implementeze un echivalent al unui calculator cu patru funcții, folosind o structură de control, o instrucțiune `switch` și să aplice o tehnică de returnare a valorii din fiecare funcție pentru operațiile aritmetice specificate: adunarea, scăderea, înmulțirea și împărțirea. Rezultatul acestui lucru este următorul:

```
double Adunare (double operandl, double operand2){
     return operandl+operand2;}
double Scadere (double operandl, double operand2){
     return operandl-operand2;}
double Inmultire (double operandl, double operand2){
     return operandl*operand2;}
double Impartire (double operandl, double operand2){
     return operandl/operand2;}
main (){
     double numl;
```

```
    double num2;
    cout<<"numl=";cin>>numl;
    cout<<"num2=";cin>>num2;
    char ch;
    char op;
    double raspuns;
    double Rezultat;
    do{
        cout <<"\nIntrodu operatia op";
        op =getche();
    switch(op){
        case '+': {raspuns = Adunare( numl, num2); break;}
        case '-': {raspuns = Scadere( numl,  num2); break;}
        case '*': {raspuns = Inmultire (numl, num2);break;}
        case '/': {raspuns = Impartire(numl, num2);break;}
    }
    cout << "\n Raspunsul este " << raspuns;
    cout << "\n Continui?(Y/N)";
    ch = getche();
    }
    while (ch != 'N');
}
```

Rezultatul returnat de la fiecare funcție matematică va fi imprimat în funcție de operație și de datele inițiale. Scopul acestui exercițiu este de a obișnui studenții să exerseze sintaxa limbajului.

3. Predarea conceptului de clase și obiecte

Clasa și obiectele constituie una dintre cele mai importante și dificile noțiuni [10]. Următoarea etapă este de a preda aceste concepte. Ideea este de a schimba obișnuința de programare a studenților de la programarea structurată la programarea orientată pe obiecte.

Prin urmare, trebuie să modificăm pasul anterior în așa fel încât să satisfacă noua idee. Începem prin a declara o clasă numită Calculator, în care toate variabilele sunt declarate ca date membru private, iar toate funcțiile membru din clasă – ca publice. De asemenea, trebuie să declarăm și să definim o funcție nouă, numită `Rezultat()`, care returnează valoarea rezultată a calculului. În funcția `main()` vom defini un obiect de tip Calculator și vom apela funcția `Rezultat()`. Rezultatul după modificare este o clasă numită Calculator ce încapsulează datele membru și funcțiile membru după cum urmează:

```
class Calculator{
     private:
     double operandl, operand2;
     double raspuns;
     char ch, op;
     public:
     double Adunare(double operandl, double operand2);
     double Scadere(double operandl, double operand2);
     double Inmultire(double operandl, double operand2);
     double Impartire(double operandl, double operand2);
     double Rezultat();
};
```

Funcția `Rezultat()` va afișa răspunsul pe rezultatul returnat de la fiecare funcție matematică. Implementarea funcției `Rezultat()` este efectuă în felul următor:

```
double Calculator::Rezultat(){
     switch(op){
     case '+': {raspuns = Ob1.Adunare(numl, num2);break;}
     case '-': {raspuns = Ob1.Scadere(numl, num2);break;}
     case '*': {raspuns = Ob1.Inmultire(numl, num2);break;}
     case "/": {raspuns = Ob1.Impartire(numl, num2);break;}
     }
     return raspuns;
}
```

În funcția `main()` se instanțiază un obiect de tip `Calculator` și se face apel la funcția `Rezultat()`:

```
main(){
     Calculator Ob1;
     ............
     do{
          cout <<"\nIntrodu operatia op";
          cin>>op;
          Solutia = Ob1.Rezultat();
          cout<<"Solutia=  "<<Solutia <<endl;
          cout << "\n Continui?(Y/N)";
          cin>>ch;
          }
     while (ch != 'N');
}
```

Rezultatul celui de-al doilea pas este faptul că studenții au trecut de la programarea structurată la programarea orientată pe obiecte. De asemenea, conceptul de clase și obiecte a fost predat și aplicat folosind modularitatea, fără a schimba semnificativ gândirea de programare a studenților.

4. Predarea conceptului de moștenire

Moștenirea este o caracteristică importantă atât pentru reutilizarea codului, cât și pentru extinderea acestuia [11]. Moștenirea este procesul de creare a unor noi clase, numite *clase derivate*, din clasele existente sau din clasele de bază [12]. Cea mai importantă caracteristică a moștenirii este reutilizarea codului. Obiectivul nostru în următorul pas este de a preda conceptul de moștenire folosind caracteristicile lui. Putem preda cu ușurință acest concept prin extinderea clasei `Calculator` cu o nouă clasă derivată și prin reutilizarea codului funcțiilor sale membre.

Așadar, pentru a ne atinge obiectivele, ar trebui luate în considerare următoarele:

- ✓ Declararea unei clase *Calcul_Stiintific*, care este moștenită din clasa `Calculator`.
- ✓ Declararea și definirea unei funcții *Adunare*() în clasa derivată *Calcul_Stiintific*, care suprascrie funcția `Adunare()` din clasa de bază. Această funcție ar trebui să returneze suma numerelor introduse după următoarea formulă: „adunați" este suma `x[i]` pentru `i=0` la `i=n-1`.
- ✓ Extinderea clasei *Calcul_Stiintific* prin declararea și implementarea următoarelor trei funcții științifice suplimentare și aplicarea unei tehnici de returnare a valorii:
  - `patrat()`, care returnează pătratul unui număr;
  - `radacina_patrata()`, care returnează rădăcina pătrată a unui număr;
  - `putere()`, care returnează puterea exponentului.
- ✓ Suprascrierea funcției Rezultat() a clasei de bază pentru a simula calculatorul.

Extensibilitatea va fi aplicată prin derivarea unei noi clase *Calcul_Stiintific* fără modificarea clasei de bază. Funcțiile membre ale acestei clase derivate sunt potrivite pentru a îndeplini funcțiile științifice: `patrat()`, `radacina_patrata()` și `putere()`. Clasa derivată are următoarea declarație:

```
class Calcul_Stiintific : public Calculator{
     public:
     double Adunare(double operandl, double operand2);
     double Patrat (double operandl);
     double Putere (double operandl, int operand2);
     double Radacina_patrata(double operandl);
     double Raspuns();
};
```

Reutilizarea codului este realizată, deoarece putem efectua mai multe adunări decât două numere. Funcția membru Adunare() a clasei derivate înlocuiește funcția clasei de bază pentru a efectua mai multe adunări. Implementarea se efectuează în felul următor:

```
double Adunare(double numl, double num2)
{
     double rezultat_cs, rezultat;
     rezultat_cs = Calculator::Adunare(numl, num2);
}
```

Funcția membru Rezultat() a clasei derivate este redefinită pentru a apela noile funcții membre extinse în scopul de a efectua un calcul științific. Calculul se efectuează pentru pătratul, rădăcina pătrată sau puterea unui număr nou introdus:

```
double Calcul_Stiintific ::Raspuns()
{
     ......................
     Calcul_Stiintific Obdr;
     raspuns_cs = Calculator::Raspuns;
     ......................
     if (op == 'p')
          {raspuns = Obdr.Patrat(raspuns_cs);}
     else if (op == 'r')
          {raspuns = Obdr.Radacina_patrata(raspuns_cs);}
     else if (op == 'e')
          {raspuns = Obdr.Putere(raspuns_cs);}
     return raspuns;
}
```

În `main()` este inițializată o matrice de obiecte `Ob1`:

```
main()
{
     ..............
     Calcul_Stiintific Obdr [max];
     do
     {
          Solutie = Obdr[n++].Raspuns();
          cout << "Raspunsul este " << Solutie << endl;
          cout << "Continuati?: "; cin>>ch;
     }
     while (ch != 'N');
     ..............
}
```

Rezultatele acestui pas sunt:

1. Aplicarea conceptului de moștenire prin extinderea clasei de bază `Calculator` cu o nouă clasă derivată *Calcul_Stiintific* pentru a efectua noi funcții științifice suplimentare.
2. Reutilizarea codului funcțiilor membre ale clasei de bază.
3. Suprascrierea unor metode din clasa de bază cu metode noi pentru a realiza mai multă manipulare.

Concluzii

Conceptele de programare orientată pe obiecte ar putea fi predate cu ușurință urmând strategia expusă mai sus pentru predarea acestor concepte. Fiecare pas este de fapt un progres către predarea și aplicarea unuia dintre conceptele POO.

Studenții devin mult mai interesați de scrierea programelor prin abordarea conceptelor POO decât prin abordarea conceptelor de programare structurată. Această strategie poate fi utilizată pentru a adopta diferite tipuri de proiecte.

Articol realizat în cadrul proiectului de cercetări științifice „Metodologia implementării TIC în procesul de studiere a științelor reale în sistemul de educație din Republica Moldova din perspectiva inter/transdisciplinarității (concept STEAM)”, inclus în „Program de stat” (2020-2023), Prioritatea IV: Provocări societale, cifrul 20.80009.0807.20, cu suportul financiar oferit de Agenția Națională pentru Dezvoltare și Cercetare

Bibliografie

1. KEUNG, J.; XIAO, Y.; MI, Q.; LEE, V.C.S. BlueJ-UML: Learning object-oriented programming paradigm using interactive programming environment. In: Proc. 2018 *International Symposium on Educational Technology* (ISET), 2018. pp. 47-51.
2. BOUALI, N.; NYGREN, E.; OYELERE, S.S.; SUHONEN, J.; V. CAVALLI-SFORZA. Imikode: A VR game to introduce OOP concepts. In: Proc. 19th *Koli Calling International Conference on Computing Education Research* (Koli Calling '19), 2019, pp. 1-2.
3. NICULESCU, V.; ŞERBAN, C.; VESCAN, A. Does cyclic learning have positive impact on teaching object-oriented programming? In: Proc. 2019 *IEEE Frontiers in Education Conference* (FIE), 2019, pp. 1-9.
4. SILVA, V.; DORA, F.A. An automatic and intelligent approach for supporting teaching and learning of software engineering considering design smells in object-oriented programming. In: Proc. 2019 IEEE *19th International Conference on Advanced Learning Technologies* (ICALT), 2019, pp. 321–323.

5. LOTFI, E.; MOHAMMED, B. Teaching object-oriented programming concepts through a mobile serious game. In: Proc. 3rd *International Conference on Smart City Applications* (SCA '18), 2018, pp. 1-6.
6. REYNA, A.M. et al. Object-oriented programming as an alternative to industrial control. In: Proc. 9th *International Conference on Electrical Engineering, Computing Science and Automatic Control* (CCE), 2012, pp. 1–7.
7. BUTLER, S.; WERMELINGER, M.; YU, Y.; SHARP, H. Mining java class naming conventions. In: Proc. 2011 27th IEEE *International Conference on Software Maintenance* (ICSM), 2011. pp. 93-102.
8. Wong, Y.S.; Yatim, M.H.M. A propriety multiplatform game-based learning game to learn object-oriented programming. In: *International Journal of Information and Education Technology*, Vol. 13, No. 2, February 2023 310 Proc. 2018 7th *International Congress on Advanced Applied Informatics* (IIAI-AAI), 2018, pp. 278–283.
9. DMITRIEVA, T.A.; PRUTZKOW, A.V.; PYLKIN, A.N. Two-Level Study of Object-Oriented Programming by University Students. DOI: 10.25559/SITITO.15.201901.200-206
10. GUZDIAL, M. Centralized Mindset: A Student Problem. with Object-Oriented Programming. In: *Proceedings of the Twenty-Six the SIGCSE Technical Symposium on Computer Science, Education, Association for Computing Machinery* - SIGPLAN Bulletin, 1995, Vol. 27, No. 1, pp. 182- 185.
11. MEYER, B. *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ, 1988.
12. LAFORE, R. *Object-Oriented Programming in C++*. Waite Group Press, Cirte Madera, CA 1994.2.