STUDIA UNIVERSITATIS MOLDAVIAE, 2015, nr.9(89)
Seria "Ştiinţe ale educaţiei" ISSN 1857-2103 ISSN online 2345-1025 p.114-121

STUDIU COMPARATIV AL CARACTERISTICILOR DE BAZĂ ALE LIMBAJELOR DE PROGRAMARE ORIENTATE PE OBIECTE

Ala GASNAŞ

Universitatea de Stat din Tiraspol

În articol se face o analiză comparativă a limbajelor de programare C++, C#, Java, Object Pascal, PHP, JavaScript din punctul de vedere al caracteristicilor Programării Orientate pe Obiecte. Această analiză are un rol important în proiectarea curriculară a disciplinei *Programarea Orientată pe Obiecte*, fiind una din disciplinele fundamentale studiate în cadrul formării iniţiale a specialiştilor IT şi a cadrelor didactice în informatică.

Cuvinte-cheie: *tehnologii informaţionale, formare iniţială, strategii de predare, limbaje de programare, conceptele POO.*

COMPARATIVE STUDY OF THE BASIC CHARACTERISTICS OF OBJECT-ORIENTED PROGRAMMING LANGUAGES

In this article is reflected a comparative analysis of the programming languages C ++, C #, Java, Object Pascal, PHP, Javascript in terms of object-oriented programming features. This analysis has an important role in designing curricular discipline *Object-oriented programming* and is one of the fundamental disciplines covered in the initial training of IT specialists and teachers in computer science.

Keywords: *information technologies, initial training, teaching strategies, programming languages, OOP concepts.*

Preliminarii

Programarea orientată pe obiecte (POO) a devenit una dintre cele mai importante paradigme de programare utilizate pe larg şi este susţinută de mai multe limbaje de programare. În prezent, C++ este unul dintre cele mai disponibile limbaje de programare şi cea mai mare parte de programatori este familiarizată cu noţiunea de *programare orientată pe obiecte* prin experienţa lor de lucru cu C++. Limbajul C++ oferă un suport eficient pentru programarea orientată pe obiecte, însă există şi alte limbaje care implementează conceptele POO în diferite moduri.

Noţiunea de *limbaje orientate pe obiecte* a apărut odată cu apariţia limbajului Simula (creat în perioada 1962-1967), dar şi acum există dezacorduri privind caracterizarea programării orientate pe obiecte. În 2006, D.J. Armstrong [1] sugerează că nu există consens în ceea ce priveşte conceptele de bază ale programării orientate pe obiecte. Acest lucru face anevoioasă descrierea şi definirea limbajelor POO, deoarece nu există un acord asupra unei definiţii universale a POO. Unii autori (O.Nierstrasz [4]) consideră că toate limbajele care suportă încapsularea au un anumit grad de orientare pe obiecte. Încapsularea este o caracteristică de bază a programării orientate, dar aceasta nu înseamnă şi obiect-orientată – pentru aceasta e necesară prezenţa moştenirii. „Însă chiar şi prezenţa caracteristicilor de încapsulare şi moştenire nu face limbajul de programare complet funcţional în ceea ce priveşte POO. Principalele beneficii ale POO se manifestă numai atunci când în limbajul de programare este implementat polimorfismul" [18].

Limbajele orientate pe obiecte diferă prin modul de aplicare a caracteristicilor de bază POO. De exemplu, unele limbaje suportă moştenirea multiplă, pe când altele – nu. În acest articol sunt analizate caracteristicile de bază ale POO şi modalitatea de implementare a acestora prin limbajele C++, C#, Java, Object Pascal, PHP şi JavaScript.

Drept urmare, în procesul de formare iniţială a specialiştilor IT şi a cadrelor didactice în informatică se studiază un şir de limbaje de programare orientate pe obiecte, aşa cum: C++, C#, Java, Object Pascal, PHP, JavaScript etc.

Învăţarea unui limbaj de programare presupune examinarea sintaxei lui, dar şi a conceptului de implementare a algoritmilor cu acest limbaj. Deoarece conceptul POO se regăseşte în fiecare din limbajele menţionate, în procesul de învăţare-predare a acestor concepte este important studiul următoarelor aspecte:

a) Stabilirea caracteristicilor POO comune, deci identificarea celor mai importante particularităţii ale POO;
b) Determinarea caracteristicilor POO specifice fiecărui limbaj;
c) Identificarea celor mai potrivite limbaje de programare pentru studierea conceptului POO.

Acest studiu permite evitarea suprapunerilor de conținuturi curriculare la învățarea limbajelor POO, precum şi elaborarea ulterioară a unor strategii de predare a conceptului POO în cadrul formării inițiale a specialiştilor IT şi a cadrelor didactice în informatică.

1. CARACTERISTICI DE BAZĂ ALE PROGRAMĂRII ORIENTATE PE OBIECTE

Studiul eficient al limbajelor orientate pe obiecte necesită cunoaşterea profundă a următoarelor caracteristici de bază ale POO.

Clasele reprezintă tipuri de date abstracte, care înglobează comportamentul şi datele asociate unui concept. Comportamentul este descris cu ajutorul metodelor, iar datele – cu ajutorul atributelor. Atributele reprezintă caracteristicile unui obiect, iar metodele sunt acțiunile pe care el le poate face. Obiectele sunt elementele definite printr-o clasă. Fiecare astfel de obiect creat poartă denumirea de *instanță a clasei* (definirea unui obiect poartă numele de *istanțiere*).

O clasă prevede mecanismul de bază prin care atributele şi metodele formează un concept comun. Ea oferă o descriere a comportamentului obiectelor instanțiate. Paradigma orientată pe obiecte presupune că metodele dintr-o clasă nu se bazează pe algoritmi comuni. Metodele depind de nivelul de detaliere la care este modelat obiectul. Astfel, o clasă defineşte o grupare logică de metode şi atribute şi acționează ca un mijloc prin care sunt atinse abstractizarea şi încapsularea datelor.

Abstractizarea: Prin abstractizare se izolează aspectele esențiale de cele neesențiale, în funcție de perspectiva de abordare a problemei de rezolvat. Nivelul de detaliere a acestor aspecte depinde de obiectul ce este abstractizat şi de cerințele problemei. În consecință, pentru aceeaşi problemă pot exista mai multe abstractizări, deci mai multe modele. Prin esență, o abstractizare este o reprezentare incompletă a realității şi tocmai aceasta dă valoare modelului corespunzător.

Încapsularea este principiul conform căruia un obiect trebuie sa aibă interfața (metodele publice ale unei clase) complet separată de implementare. Toate datele şi codul implementării trebuie să fie complet ascunse în spatele interfeței. După crearea unei interfețe aplicația poate interacționa cu obiectele. Încapsularea serveşte la separarea interfeței de implementarea acesteia. Rezultă că un obiect este format din două părți distincte: interfața şi, respectiv, implementarea acestei interfețe. Încapsularea este conceptul complementar abstractizării. Abstractizarea este procesul prin care este definită interfața obiectului, în timp ce încapsularea defineşte reprezentarea obiectului împreună cu implementarea interfeței.

Încapsularea permite păstrarea unei limite clare între o clasă şi lumea exterioară şi oferă programatorilor libertatea de a schimba funcționarea internă a unei clase.

Moştenirea: În programarea orientată pe obiecte, conceptul de moştenire este deosebit de important. Moştenirea este mecanismul de bază cu ajutorul căruia pot fi create modelele ierarhice ale claselor. Posibilitatea de a crea o arborescență de clase, de la cea mai generală la cea mai specifică, permite un control mai bun al programelor, permițând, totodată, lizibilitatea programelor de către utilizatorii textelor-sursă sau dezvoltatorii de programe. Moştenirea este o formă de reutilizare a codului, în care noile clase sunt create din clase existente, numite ulterior clase de bază sau superclase. Crearea claselor noi (subclaselor sau claselor derivate), pornind de la o clasă de bază, permite moştenirea proprietăților originale ale clasei respective prin absorbirea atributelor şi comportamentelor lor, prin înlocuirea unor comportamente şi prin adăugarea unor atribute şi comportamente noi.

Prin urmare, moştenirea este mecanismul de a obține o clasă de la o altă clasă, păstrând în acelaşi timp toate proprietățile şi metodele clasei-părinte (de bază) şi adăugarea, dacă este necesar, a unor proprietăți şi metode noi. Prin ea se promovează reutilizarea codului şi dezvoltarea codului structurat.

Mecanismul moştenirii multiple permite unei clase să moştenească trăsături din multiple clase. Dar acesta este un mecanism mai periculos, deoarece proprietățile şi metodele unei superclase, prin moştenire multiplă, pot fi moştenite de către o clasă derivată din ea prin mai multe căi.

Polimorfismul: Într-o ierarhie de clase obținute prin moştenire, o metodă poate avea forme diferite de la un nivel la altul (specifice respectivului nivel de ierarhie) şi poate funcționa diferit în obiecte diferite. Polimorfismul este un fenomen în care unul şi acelaşi cod este realizat diferit, în funcție de clasa sau tipul obiectului utilizat la apelarea acestui cod. Polimorfismul este asigurat de faptul că în clasa derivată se schimbă realizarea metodei din clasa de bază, dar cu păstrarea „signaturii" metodei (acelaşi nume şi aceiaşi parametri de intrare/ieşire).

Există mai multe tipuri de polimorfism, care au fost grupate în două categorii mari: Polimorfismul Ad-hoc şi Polimorfismul Universal [9]. *Polimorfismul Ad-hoc* se obține atunci când o funcție lucrează sau pare că lucrează cu mai multe tipuri diferite şi se poate comporta în mod independent pentru fiecare tip. *Polimorfismul Universal* este atunci când o clasă sau o funcție se comportă la fel pentru o mulțime de tipuri. Principala diferență dintre Polimorfismul Ad-hoc şi Polimorfismul Universal constă în faptul că funcțiile polimorfe ad-hoc execută un cod distinct pentru un mic set de tipuri, potențial independente, în timp ce funcțiile polimorfe universale execută acelaşi cod pentru un număr infinit de tipuri (de ex., pentru toate tipurile admisibile) [10].

Polimorfismul Ad-hoc, la rândul său, se divizează în două tipuri: supraîncărcarea şi conversia. *Supraîncărcarea* utilizează acelaşi nume de funcție pentru a se referi la diferite funcții reale, care se disting prin tipul şi numărul de argumente. *Conversia* permite valorii unui tip să fie convertit la valoarea altui tip. Conversia este o operație semantică necesară pentru a converti un argument de un tip la tipul aşteptat de o funcție. De exemplu, o valoare întreagă poate fi folosită în cazul în care este aşteptată o valoare reală, şi invers. Conversia poate fi statică, inserând în mod automat conversia tipului necesar între argumente şi funcții în timpul compilării, sau conversia tipului necesar poate fi determinată dinamic prin testele runtime (în timpul execuției). Conversia poate reduce dimensiunea programului şi poate îmbunătăți lizibilitatea lui, dar poate, de asemenea, cauza erori subtile de sistem şi uneori chiar periculoase.

Polimorfismul Universal are şi el două mari categorii: polimorfismul parametric şi polimorfismul de includere, care în limbajele orientate pe obiecte poartă denumirea de polimorfism de moştenire [11]. *Polimorfismul parametric* realizează mecanismul prin care se poate defini o metodă cu acelaşi nume în aceeaşi clasă (funcțiile trebuie să difere prin numărul şi/sau tipul parametrilor). Selecția funcției se realizează la compilare, ceea ce se numeşte *legarea timpurie* (early binging). *Polimorfismul de moştenire* este mecanismul prin care o metodă din clasa de bază este redefinită cu aceiaşi parametri în clasele derivate. Selecția funcției se va realiza la rulare, ceea ce este *legarea întârziată* (late binding, dynamic binding, runtime binding) [12].

Polimorfismul oferă o comoditate semnificativă în programarea orientată pe obiecte prin furnizarea structurilor pentru gestionarea diferitelor obiecte. Spre exemplu, metodele ce fac lucruri similare (de ex., operația de adunare) pot avea acelaşi nume cu diferite „signaturi" (în acest caz vorbim despre *polimorfism de supraîncărcare*); aceeaşi clasă/metodă (de ex., operația de sortare) poate lucra cu mai multe tipuri de obiecte (în acest caz vorbim despre *polimorfism parametric*); o subclasă poate fi substituită de o clasă-părinte (*polimorfism de moştenire*).

2. LIMBAJE DE PROGRAMARE ORIENTATE PE OBIECTE

Un limbaj de programare poate fi considerat orientat pe obiecte dacă are mecanisme pentru implementarea caracteristicilor POO: abstractizarea, încapsularea, moştenirea, polimorfismul.

O diferență în rândul limbajelor OOP aduc noțiunile de limbaj pur sau limbaj hibrid. Limbajele OOP pure sunt cele care permit doar un singur model de programare – OOP. Se pot declara clase şi metode, dar nu sunt prezente funcții şi proceduri simple, vechi şi date globale. Limbajul Java, C# sunt limbaje OOP pure (în sensul definiției de mai sus). C ++ şi Object Pascal, în schimb, sunt două exemple tipice de limbaje hibrid, care permit programatorilor să folosească limbajul tradițional C şi abordări de programare Pascal.

Pentru a studia cu succes un limbaj orientat pe obiecte, este necesar de a reflecta în procesul de învățare-predare care dintre caracteristicile POO menționate se regăsesc în unele dintre cele mai solicitate limbaje de programare.

Vom descrie momentele importante care trebuie să însoțească analiza caracteristicilor POO.

Limbajul C++ este un limbaj de programare orientat pe obiecte utilizat pe larg şi oferă o bază bună pentru comparație. C++, fiind derivat din limbajul C, combină avantajele oferite de acest limbaj (eficiența, flexibilitatea şi popularitatea) cu avantajele oferite de tehnica POO.

Acest limbaj este unul hibrid, deoarece susține atât programarea procedurală, cât şi cea orientată pe obiecte, spre deosebire de limbajul Java, care este un limbaj orientat pe obiecte pur.

Deşi adoptă principiile Programării Orientate pe Obiecte, C++ nu impune aplicarea lor strictă. Conceptul fundamental în C++, ca şi la alte limbaje POO, este clasa. El asigură încapsularea datelor, inițializarea datelor, gestiunea memoriei controlată de utilizator, mecanisme pentru supraîncărcarea operatorilor.

Limbajul C++, spre deosebire de alte limbaje POO, oferă atât moştenirea simplă, cât şi moştenirea multiplă. În acest limbaj lipseşte mecanismul garbage-collection (mecanismul de colectare a gunoiului (GC) este o

formă de management automat al memoriei), dar este implementată tratarea excepţiilor. C++ foloseşte legarea întârziată (late binding), ceea ce înseamnă că programatorul trebuie să precizeze clasa specifică obiectului, sau cel puţin clasa generală la care poate aparţine obiectul. „Pe de o parte, acest lucru face să crească eficienţa la lansare şi codul să fie de dimensiuni mici; pe de altă parte însă, pierde din puterea de reutilizare a claselor" [2]. C++ poate utiliza pointerii pentru a manipula memoria, o activitate necesară pentru scrierea componentelor sistemului de operare low-level.

Limbajul Java are o sintaxă similară cu C++, astfel studierea lui devine mult mai uşoară pentru cei care au deprinderi de lucru cu limbajul C++. Java este un limbaj orientat pe obiecte pur. Limbajul dat nu acceptă construcţii low-level precum le avem în C++ (cum ar fi pointerii), ceea ce permite colectorului garbage să realoce obiectele referite. În Java, clasa este structura fundamentală. Toate clasele din Java şi C# descend dintr-o clasă de bază numită *obiect* [3]. Limbajul Java oferă moştenire, tratarea excepţiilor, modularitate, colectorul garbage, polimorfism etc. Dar Java susţine doar moştenirea simplă, spre deosebire de C++, în schimb oferă interfeţe (interfaţa defineşte un set de metode care vor fi implementate de una sau mai multe clase).

Limbajul C# a fost conceput pentru a fi un limbaj de programare orientat pe obiecte pur. Este un limbaj ce permite programarea structurată, modulară şi orientată obiectual, conform perceptelor moderne ale programării. Principiile de bază ale programării orientate pe obiecte – încapsulare, moştenire, polimorfism, sunt elemente fundamentale ale programării C#. La baza acestuia stă limbajul C, dar sunt preluate şi principiile de programare din C++. Sintaxa este apropiată şi limbajului Java. Ca şi limbajul Java, are colector garbage şi ca şi Java se compilează într-un limbaj intermediar [5]. În C# conceptele de clasă, moştenire şi polimorfism sunt la fel ca şi în celelalte limbaje POO.

Limbajul Object Pascal este un limbaj de programare derivat din Pascal ce permite folosirea structurilor din modelul programării orientate pe obiecte: obiecte, moştenire, polimorfism.

Un element al limbajelor OOP este modelul obiectual. Unele limbaje tradiţionale OOP permit crearea de obiecte în stiva, heap sau stocarea statică. În aceste limbaje o variabilă de tip clasă corespunde unui obiect din memorie. Acesta este modul în care funcţionează C ++. În ultimul timp se utilizează un alt model, numit model de referinţă obiect. În acest model fiecare obiect este alocat dinamic în heap, şi o variabilă de tip clasă este de fapt o referinţă sau un pointer la obiectul din memorie. Java, C# şi Object Pascal adoptă acest model de referinţă.

Object Pascal nu are colector garbage. Cu toate acestea, componentele Delphi (Delphi este un limbaj de programare şi un mediu de dezvoltare pentru programe. Limbajul Delphi este cunoscut ca Pascal orientat pe obiecte) susţin ideea unui obiect proprietar: proprietarul devine responsabil pentru distrugerea tuturor obiectelor pe care le deţine. Delphi utilizează, de asemenea, şi mecanismul de numărare a referinţelor pentru şiruri de caractere, tablouri dinamice şi interfeţe, eliberând obiectele din memorie, atunci când nu mai are linkuri [17]. Acest lucru face manipularea cu distrugerea obiectelor destul de simplă. Destructorii din Object Pascal sunt similari cu destructorii din C ++.

Limbajul PHP este un limbaj de scripting derivat din familia C. PHP rulează pe un server web şi poate fi integrat în HTML, pentru a da un conţinut dinamic paginii web.

Începând cu PHP 5, a fost rescris modelul obiectual pentru a permite o performanţă mai bună a limbajului şi introducerea mai multor caracteristici. PHP tratează obiectele la fel ca referinţele, ceea ce înseamnă că fiecare variabilă conţine mai degrabă o referinţă la obiect, decât o copie a întregului obiect [19].

Limbajul JavaScript este un limbaj de programare orientat pe obiecte independent de sistemul operaţional folosit în calculator. Spre deosebire de C # sau C ++, JavaScript foloseşte o altă abordare pentru a crea un limbaj orientat pe obiecte. Acesta este un limbaj bazat pe prototip. Conceptul de prototip implică faptul că un comportament poate fi reutilizat prin clonarea obiectelor existente, care servesc drept prototipuri. Fiecare obiect în JavaScript este descendentul unui prototip, care defineşte un set de funcţii membre, pe care le poate folosi obiectul [14]. În JavaScript nu există noţiune de clasă. Sunt doar obiecte. Fiecare obiect poate fi apoi utilizat ca un prototip pentru un alt obiect.

3. ANALIZE COMPARATIVE

Pentru a efectua o comparare a caracteristicilor POO, propunem o analiză a gradului şi specificul de implementare a acestor caracteristici în limbajele de programare orientate pe obiecte descrise anterior.

Moştenirea şi polimorfismul în limbajele POO: Moştenirea este un mecanism care realizează relaţia de ierarhie în modelul de clasă. Relaţia ierarhică poate fi extinsă la toate clasele din limbajele Java şi C#. Aceste

limbaje conțin o clasă generică, care este un strămoş pentru toate clasele din limbaj. Prezența unui singur strămoş asigură toate clasele cu anumite funcționalități minime, care sunt moştenite de la acest strămoş. C++ nu are o clasă-strămoş. Avantajul acestei abordări este faptul că pentru funcționarea unei aplicații nu este nevoie de a se lega cu întreaga ierarhie de obiecte. În cazul Java, la lansarea fişierului executabil, pentru orice aplicație toate clasele din ierarhie trebuie să fie prezente.

Accesibilitatea: În limbajele C++ şi Java, prin cuvintele-cheie *public*, *private* şi *protected*, aplicate la metodele şi atributele clasei de bază, se controlează nivelul de vizibilitate. Atât C++, cât şi Java definesc 3 niveluri de acces: *public*, *private* şi *protected*. Cu nivelul de acces *public*, membrii clasei de bază sunt vizibili pentru toate clasele. Caracteristica *private* nu permite vizibilitatea la nicio altă clasă în afară de clasa în care este declarată. Totuşi, în C++ există o excepție. Limbajul C++ are, de asemenea, noțiunea de *clasă-prietenă*. Acest limbaj permite claselor-prietene accesul chiar şi la metodele şi atributele private. Caracteristica *protected* specifică faptul că membrul este vizibil pentru codul din aceeaşi clasă şi clasele descendente.

În limbajul C# sunt definite 5 niveluri de acces pentru membrii clasei: *private*, *protected*, *internal*, *protected internal* şi *public*. Modificatorul de acces *internal* specifică faptul că membrul este accesibil din interiorul clasei sau intr-un bloc funcțional al unei aplicații. Un membru declarat ca *protected internal* va fi accesibil oricărui membru al clasei care îl conține şi al claselor derivate, precum şi în blocul funcțional.

Atunci când o clasă este moştenită, restricțiile vizibilității sunt moştenite de către subclase. Subclasa poate accesa variabilele protejate, dar nu le poate accesa pe cele private. Subclasa poate reduce vizibilitatea membrilor săi prin schimbarea caracteristicilor *public* cu caracteristici *protected* sau *private*. Caracteristica vizibilității nu poate fi crescută. Membrii, care posedă caracteristica *private,* nu sunt vizibili nici chiar în subclasă şi uneori ea poate fi cel mai bun mod de extensibilitate a clasei.

Există situații în care unele metode dintr-o superclasă (clasa de bază) e necesar să fie exportate la toți descendenții, dar la fel de esențial este ca aceste metode să nu fie redefinite. Acelaşi lucru este valabil şi pentru atribute. În limbajele C# şi Java există anumite cuvinte-cheie ("sealed", respectiv "final") care denotă că aceste metode ori clase nu pot fi redefinite [6, 7]. Limbajul C++ nu are această abilitate.

În Obiect Pascal, pentru moştenire se utilizează o sintaxă specială, adăugând în paranteze numele clasei de bază. Acest limbaj acceptă doar un singur tip de moştenire – *public*. Clasele în Object Pascal sunt derivate dintr-o clasă de bază comună. Accesul la metodele clasei de bază se face prin cuvântul *inherited*. După acest cuvânt se scrie numele metodei din clasa de bază [16].

Object Pascal, la fel ca şi limbajul Java şi C#, nu susține moştenirea multiplă, în schimb utilizează opțiunea de *interfețe multiple.*

PHP foloseşte pe larg principiul de moştenire în modelul său obiectual. Acest principiu va influența modul de interacțiune dintre clase şi obiecte. De exemplu, atunci când o clasă se extinde, subclasa moşteneşte toate metodele publice şi protejate din clasa de bază. Aceste metode îşi vor păstra funcționalitatea lor originală, cu excepția cazului în care clasa redefineşte aceste metode. Acest lucru este util pentru definirea şi abstractizarea funcționalității şi permite punerea în aplicare a funcționalităților suplimentare în scopuri similare [19].

În PHP moştenirea se implementează folosind cuvântul-cheie *extends*. În acest limbaj este posibil să se pună în aplicare moştenirea pe mai multe niveluri, adică moştenirea ierarhică, dar nu şi cea multiplă.

Limbajele bazate pe prototipuri creează obiecte, fără a recurge la clase. Aceste limbaje sunt bazate pe definirea de prototipuri şi manipularea lor. Atunci când este nevoie de un nou obiect, obiectul existent este clonat şi apoi modificat. În astfel de limbaje nu există noțiunea de tip ci doar de similitudine de obiecte. Când este creat un obiect nou, el poate defini un comportament nou şi să păstreze un anumit comportament vechi. JavaScript este un limbaj orientat pe obiecte fără clase (class – free), şi în loc de moştenirea clasică foloseşte moştenirea prototipică, iar obiectele moştenesc de la alte obiecte [14].

Clase abstracte: Sunt situații când în clase este dată doar metoda ce trebuie să fie moştenită, fără a fi precizată şi implementarea ei. Subclasa, în acest caz, este forțată să ofere implementări pentru toate metodele din superclasă. Acest mecanism poate fi utilizat pentru a se asigura că subclasa corespunde unui anumit design (construcții). Java şi C# utilizează cuvântul-cheie „abstract" pentru a indica faptul că metoda sau clasa nu are nicio implementare şi, prin urmare, subclasele trebuie să furnizeze definiția. Dar aceste clase abstracte nu pot fi instanțiate. Chiar dacă în clasă este doar o metodă abstractă, clasa nu poate fi instanțiată. Java face acest lucru explicit prin marcarea întregii clase ca fiind abstractă.

În limbajul C++ există posibilitatea de a defini clase abstracte, care sunt destinate creării de noi clase prin derivare, ele neputând fi instanțiate şi utilizate ca atare. Ele constituie o bază în cadrul elaborării ierarhiilor de clase, putând fi folosite, spre exemplu, pentru a impune anumite restricții în realizarea claselor derivate. Pentru efectuarea acestui lucru, se foloseşte cuvântul-cheie „virtual". În vederea construirii unor astfel de clase, s-a introdus şi conceptul de *funcție virtuală pură*. O astfel de funcție este declarată în cadrul clasei, dar nu este definită. O clasă care conține o funcție virtuală pură este considerată abstractă. Funcțiile virtuale pure trebuie definite în clasele descendente, altfel şi acestea vor fi considerate abstracte.

În limbajul Object Pascal clasele abstracte sunt clase care posedă metode abstracte sau care moştenesc metode abstracte. Pentru descrierea acestor metode se foloseşte cuvântul-cheie *abstract*. Spre deosebire de C++ şi Java, în Object Pascal poate fi creat un obiect al unei clase abstracte (deşi, în acest caz, compilatorul va da un mesaj de avertizare).

Noțiunea de clase abstracte şi metode este introdusă şi în PHP 5. În acest limbaj nu este permisă instanțierea unei clase abstracte. Ca şi în C++, orice clasă care conține cel puțin o metodă abstractă trebuie să fie abstractă. Metodele definite ca abstracte, declară doar „signatura" metodei, implementarea acesteia se face în clasele descendente (derivate). Când se moşteneşte de la o clasă abstractă, toate metodele marcate abstract în declarația clasei de bază trebuie să fie definite în clasele derivate. În plus, aceste metode trebuie să fie definite cu aceeaşi vizibilitate. De exemplu, în cazul când metoda abstractă este definită ca fiind *protected*, implementarea ei trebuie definită ca *protected* sau *public*, dar nu *private* [19].

Moştenirea multiplă: Moştenirea multiplă presupune posibilitatea ca o clasă să moştenească din mai multe clase-părinte. Acest mecanism trebuie folosit, însă, cu atenție, pentru că poate cauza o serie de ambiguități. Atunci când o clasă moşteneşte de la mai multe clase, aceasta nu este o versiune a uneia dintre clasele-părinte. Aceste probleme apar din cauza faptului că moştenirea multiplă defineşte o moştenire structurată ca un graf orientat aciclic, şi nu ca un arbore. Într-un arbore există o cale unică de la orice clasă derivată la orice nod-strămoş. Într-o structură aciclică, însă, pot exista mai multe căi între o subclasă şi strămoşii săi. În cazul când subclasa se referă la o caracteristică moştenită, căutarea acestei caracteristici pune probleme semnificative.

În moştenirea multiplă, bazată pe structura de graf, apar probleme uzuale, precum conflicte de nume şi moştenirea repetată. Conflictele solicită utilizarea diferitelor tehnici pentru soluționarea lor. Limbajul C++ foloseşte pentru rezolvarea unor astfel de conflicte redenumirea metodelor. Problema de moştenire repetată este rezolvată în C++ prin utilizarea moştenirii virtuale. Atunci când o clasă este definită a fi moştenită virtual, C++ se asigură că există doar o singură copie a claselor în această cale. Aceasta rezolvă problemele moştenirii repetate.

Spre deosebire de C++, limbajele C# şi Java, Obect Pascal, PHP nu implementează moştenirea multiplă. Ca o alternativă la moştenirea multiplă, limbajele date utilizează opțiunea de *interfețe multiple*. Deoarece o interfață nu specifică niciun fel de implementare, pot fi „combinate" mai multe interfețe. Combinarea unor interfețe ce conțin o metodă cu acelaşi nume este posibilă doar dacă metodele nu au tipuri întoarse diferite şi aceeaşi listă de argumente. Totuşi, e bine ca în interfețele diferite, care trebuie combinate, să nu existe metode cu acelaşi nume, deoarece acest lucru poate duce la confuzii evidente [8].

Polimorfismul: *Supraîncărcarea:* Limbajele C# şi C++ folosesc, în scopul implementării caracteristicii polimorfism, cuvinte-cheie. Supraîncărcarea operatorilor este criticată de multe ori pentru că permite programatorilor să acorde operatorilor o semantică complet diferită în funcție de tipurile operanzilor. Limbajul Java permite supraîncărcarea doar a operatorilor aritmetici.

Limbajele Java, C++ şi C# permit supraîncărcarea metodelor în mod similar. Atât timp cât „signaturile" metodelor sunt diferite, compilatorul tratează metodele cu nume supraîncărcate de parcă ar avea denumiri complet diferite. Mai mult decât atât, în aceste trei limbaje supraîncărcarea se poate întâmpla atunci când o metodă din clasa de bază este moştenită într-o subclasă care are o metodă cu acelaşi nume, dar cu signatură diferită. În acest caz, compilatoarele folosesc din nou legarea timpurie, pentru a face diferența dintre metodele supraîncărcate [13].

Prin intermediul Delphi şi Free Pascal, în Object Pascal au fost introduse o serie de caracteristici noi. Printre aceste caracteristici se numără şi supraîncărcarea [17].

Interpretarea termenului „supraîncărcare" în PHP este diferit decât în celelalte limbaje orientate pe obiecte. Supraîncărcarea tradițională oferă posibilitatea de a avea mai multe metode cu acelaşi nume, dar diferite cantități şi tipuri de argumente**.** Supraîncărcarea în PHP oferă mijloace dinamice de „a crea" proprietăți şi metode.

Metodele supraîncărcare sunt invocate atunci când interacționează cu proprietăți sau metode care nu au fost declarate sau care nu sunt vizibile în domeniul de aplicare curent [15]. În PHP toate metodele de supraîncărcare trebuie să fie definite ca publice.

Conversia: Conversia în diferite limbaje este implementată diferit. C++ permite conversia implicită între tipurile sale, dar şi pentru tipurile definite de utilizator, prin intermediul constructorilor de conversie şi al operatorilor de conversie definiți de utilizator. Totodată, Java a lărgit conversiile implicite între tipurile native, dar alte conversii necesită o sintaxă explicită.

Limbajul PHP foloseşte filtrele de conversie.

Polimorfismul de moştenire: Polimorfismul runtime (în timpul executării), sau redefinirea metodei, mai este numit şi legare întârziată (late binding) sau polimorfism dinamic.

Posibilitatea de a se referi la obiectul comun din diferite clase dintr-o ierarhie şi de a apela la metoda clasei respective este foarte utilă, în cazul când mai multe clase din această ierarhie redefinesc o metodă. Pentru aceasta, compilatorul trebuie să sprijine legarea întârziată, ceea ce înseamnă că în timpul executării se determină tipul real al obiectului şi metoda ce trebuie apelată.

În C++ legarea întârziată este accesibilă doar pentru metodele virtuale. Metodele declarate în clasa de bază ca virtuale (*virtual*) susțin această caracteristică (dar numai dacă descrierea metodelor este identică). Metode obişnuite, non-virtuale, nu permit legarea întârziată, la fel ca şi în Object Pascal.

În limbajul Object Pascal legarea întârziată este introdusă cu ajutorul cuvintelor-cheie *virtual* şi *dynamic*. În clasele derivate metodele redefinite trebuie marcate prin cuvântul-cheie *override.*

În limbajul Java, toate metodele folosesc legarea întârziată în mod obligatoriu, cu excepția cazului când este marcat ca *final* în mod clar.

În limbajul C# polimorfismul de runtime se implementează prin redefinirea metodei din clasa de bază în clasa derivată. Acest lucru poate fi realizat prin utilizarea principiului de moştenire şi a cuvintelor-cheie *virtual* şi *override*. Daca în clasa de bază declarăm metode folosind cuvântul-cheie *virtual,* atunci în clasa derivată aceste metode vor fi redefinite folosind cuvântul-cheie *override*.

În limbajul PHP, începând cu versiunea 5.3, a fost pusă în aplicare o caracteristică numită *late static binding* (legare statică întârziată). Late static binding poate fi folosită pentru a face referire la apelarea claselor în contextul moştenirii statice.

Din punctul de vedere al caracteristicilor de bază ale tehnologiei orientate pe obiecte, aceste limbaje includ în structura lor următoarele principii (a se vedea Tabelul)

Tabel

Caracteristica de bază	Limbajul C++	Limbajul C#	Limbajul Java	Limbajul Obj.Pascal	Limbajul PHP	Limbajul JavaScript
Abstractizarea	Instanțiere date Instanțiere metode Variabile de tip clasă Metode de tip clasă	Instanțiere date Instanțiere metode Variabile de tip clasă Metode de tip clasă	Instanțiere date Instanțiere metode Variabile de tip clasă Metode de tip clasă	Instanțiere date Instanțiere metode	Instanțiere date Instanțiere metode Variabile de tip clasă Metode de tip clasă	Nu suportă abstactizarea clasică, ci doar de prototip
Încapsularea	public, protected, private	public, protected, private, internal, protected internal	public, protected, private	Public	public, protected, private	public, private
Moştenirea	Simplă, multiplă	Simplă, interfețe	Simplă, interfețe	Simplă	Simplă, interfețe	De prototip
Polimorfismul	Da	Da	Da	Da	Da	De prototip

Concluzii

1. Caracteristicile de bază ale POO sunt încapsularea, abstractizarea, moştenirea şi polimorfismul. Ele vor constitui repere pentru elaborarea curriculei la disciplina *Programarea Orientată pe Obiecte*.

2. Se cunoaşte că fiecare limbaj a fost conceput pentru a soluționa probleme dintr-un anumit domeniu. Considerăm că, din punct de vedere metodologic, cele mai potrivite limbaje pentru studierea POO sunt C++ şi C#.

3. Există diferențe între modalitățile de implementare a caracteristicilor de bază ale POO pentru diferite limbaje de programare. Astfel, dacă în limbajele C# şi Java există anumite cuvinte-cheie, care denotă că unele metode dintr-o superclasă nu pot fi redefinite, limbajul C++ nu are această abilitate. Limbajele C# şi Java, Obect Pascal, PHP, spre deosebire de C++, nu implementează moştenirea multiplă. Spre deosebire de C++ şi Java, în Object Pascal poate fi creat un obiect al unei clase abstracte. Limbajul JavaScript are propriul concept obiectual şi se deosebeşte radical de celelalte limbaje examinate în articol.

4. Fiecare din limbajele de programare POO are domenii prioritare de aplicare, de aceea studierea caracteristicilor POO specifice este importantă pentru alegerea celui mai adecvat limbaj în funcție de domeniul problemei care urmează a fi soluționată. Acest aspect este important în procesul de învățare-predare a POO.

5. În proiectarea curriculară a disciplinei *Programarea Orientată pe Obiecte* de evidențiat caracteristicile de bază şi analiza comparativă a limbajelor de programare orientate pe obiecte.

Bibliografie:

1. ARMSTRONG, D.J. The quarks of object-oriented development. In: *Communications of the ACM*, 49(2):123–128, 2006.
2. ELLIS, M. A. and STROUSTRUP, B. *The annotated C++ reference manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.
3. GOSLING, J., JOY, B., STEELE G., and BRACHA, G. *Java (TM) Language Specification*, The (Java (Addison-Wesley)). Addison-Wesley Professional, 2005.
4. NIERSTRASZ, O. *A Survey of Object-Oriented Concepts*. Object-Oriented Concepts, Databases and Applications, p.3-22, 1989.
5. http://en.csharponline.net/CSharp_Overview# A_Brief_History_of_C.23
6. http://www.arh.pub.ro/lab/poo/lab2.html
7. http://www.math.uaic.ro/~cgales/csharp/Curs5.pdf
8. http://arh.pub.ro/lab/poo/lab2.pdf
9. http://www.itu.dk/courses/BPRD/E2009/fundamental-1967.pdf
10. http://dic.academic.ru/dic.nsf/ruwiki
11. https://ru.wikipedia.org/wiki
12. http://www.math.uaic.ro/~mapetrii/POO
13. http://www.derangedcoder.net/programming/general/comparingObjectOrientedFeatures.html
14. http://markdalgleish.com/2012/10/a-touch-of-class-inheritance-in-javascript/
15. http://php.net/manual/en/language.oop5.overloading.php
16. http://docwiki.embarcadero.com/RADStudio/XE5/en/Operator_Overloading_(Delphi)
17. http://www.tutorialspoint.com/pascal/pascal_object_oriented.htm
18. http://cs.mipt.ru/wiki/index.php?title
19. PHP: Hypertext Preprocessor. *PHP Manual: Chapter 19 – Classes and Objects*. http://php.net/manual/en/language.oop5.php.

Prezentat la 20.10.2015