CĂLIN BOLEA

SEMIOLOGIA PROGRAMĂRII ORIENTATE PE OBIECT

Călin Bolea
Babeș Bolyai University, Faculty of History and Philosophy, Cluj-Napoca, Romania
Email: calin.bolea@gmail.com

Rezumat: Programarea este astăzi una dintre profesiile cele mai interesante și provocatoare din punct de vedere filosofic. Semiologia, semiotica și hermeneutica sunt ramuri ale filosofiei care explorează domeniul programării, mai precis, programarea orientată spre obiecte. Clasele create prin programarea orientată pe obiect definesc ce este semnificativ pentru un obiect și modul în care interacționează cu alte obiecte. Modelele de design profită de această semnificație și o leagă la un sistem mai mare. Aceste elemente contribuie la conturarea unui context favorabil pentru explorarea modelelor de semne triadice și diadice. Mai mult, ținând cont de numărul mare de persoane care citesc și scriu codul, hermeneutica poate juca un rol important în înțelegerea procesului de dezvoltare. Toate acestea pot să contribuie la diminuarea prăpastiei dintre filosofia academică și viața și activitățile de zi cu zi și să ajute ambele lumi să înțeleagă mai bine obiectul lor de atenție. Explorarea acestor ramuri ale filosofiei prin paradigma orientată către obiect va avea nevoie de mai mult decât de o simplă privire de ansamblu asupra potențialului acesteia, însă discuția trebuie să pornească de la o introducere la nivel înalt înainte de a ajunge la influențe mai specifice. Unii cercetători au început deja acest demers și prevăd o creștere a lucrărilor academice care tratează un subiect similar.

Cuvinte-cheie: *OOP*, programare, semiotică, semiologie, hermeneutică, Ferdinand de Saussure, *coding*, semn

THE SEMIOLOGY OF THE OBJECT ORIENTED PROGRAMMING

Abstract: Programming is one of the most philosophically interesting crafts of our time. Semiology, semiotics and hermeneutics can be some of the branches of philosophy that explore this craft and more specifically, object oriented programming.

Revista de Filosofie Aplicată, vol. 3, issue 4 (Spring 2020): 104-116.
ISSN 2668-0009; ISSN-L 2668-0009

The classes created through object oriented programming define what is significant for an object and the way it interacts with other objects. Design patterns take advantage of that significance and tie it to a larger system. This makes a good context for exploring the triadic and dyadic sign models and with so many people reading and writing code, hermeneutics can play a big role in understanding the process of development. All this can bridge the gap between academic philosophy and day-to-day lives and jobs and help both worlds understand their object of attention better. Exploring this branches of philosophy through the object oriented paradigm will need more than a mere overview of it's potential, but the discussion needs to start from a very high-level introduction before getting to more specific influences. Some scholars have already started this journey and in the future, I foresee an increase in scholarly papers treating a similar subject.

Keywords: OOP, programming, semiotics, semiology, hermeneutics, Ferdinand de Saussure, coding, sign

1. Introducere

Practicând programarea, văd multe domenii unde conceptele pe care le folosesc zi de zi se regăsesc într-o formă sau alta. Parte din motivul pentru care am început să studiez filosofia e pentru a înțelege genul de gândire care m-ar ajuta să îmi fac meseria mai bine, iar când am descoperit ideile lui Charles Sanders Pierce, ale lui Ferdinand de Saussure și ale predecesorilor lor m-am gândit cum acestea ar putea fi utilizate asupra paradigmei de programare orientată pe obiecte. În același timp, consider că și hermeneutica ar putea fi de ajutor în a explora acest tip de paradigmă. Am decis să încerc să demonstrez că programarea orientată pe obiecte poate fi analizată prin aceste ramuri ale filosofiei.

Această lucrare nu e o analiză în sine, ci mai mult un exercițiu de identificare a unei arii de interes academic care, odată studiată ar crea o înțelegere ubicuă a programării printre oamenii tehnici și filosofi. Această lucrare nu e nici un efort de exegeză a lucrărilor deja existente pe această temă. Dimensiunile modeste ale lucrării de față nu oferă suport pentru prezentarea ideilor lui Kumiko Tanaka-ishii, în a lui Semiotică a programării, sau întregii serii de lucrări în domeniu ale lui Peter Anderson, ci aș vrea să deschidă calea înțelegerii lor.

2. Programarea Orientată pe Obiecte (OOP)

Multe din limbajele moderne de programare sunt exhaustive în sensul în care se poate programa în ele orice susține platforma fizică și limitările intelectuale ale programatorului, dar limbajul nu e totul. Limbajele de programare conțin relativ puține cuvinte, spre deosebire de limbile vorbite de creatorii lor, din mai multe motive utilitare, tehnice și economice, în schimb, cuvintele lor sunt foarte puternice: reținerea unei valori în memorie cu siguranța că nu o vei uita, executarea unei mișcări repetate de câte ori ai vrea gândind o singură dată mișcarea și un număr de repetări, operații matematice la îndemână, puterea absolută de diferențiere, toate astea sunt virtuți

care pentru oameni sunt în afara potențialului lor. Cu toate acestea, pentru definirea unor concepte mai complexe, limbajele de programare sunt în dezavantaj față de limbile oamenilor. Definirea conceptului de scaun, în toate reprezentările și semnificațiile lui ajunge să fie atât de grea pentru un programator pe cât i-a fost și structuralistului care încerca să-și dea seama ce e scaunul din fața lui. Astfel, orice obiect sau concept care trebuie reprezentat în felul ăsta presupune o dificultate ridicată în orice limbaj de programare. Aici intervine doctrina programării orientate pe obiect, la care de acum, de dragul tradiției tehnice, o să mă refer prin abreviația termenului englezesc, OOP (object oriented programming). Filosofia OOP-ului presupune gândirea tuturor cuvintelor de care ai avea nevoie pentru a semnifica unei mașini un concept în termeni de obiecte și relația dintre ele. Obiectul, în doctrina OOP, ajunge să fie asemănător Ideilor sau Formelor lui Platon, iar munca unui programator e să găsească cea mai bună reprezentare a Ideii astfel încât arhitectura Formelor să ia formă. În acest fel, o clasă nu e Ideea în sensul în care e folosit de Platon (și corolar, un obiect o instanță a Ideii), ci mai degrabă clasa e Forma imperfectă, la fel cum eu și tu suntem imperfecți față de Forma omenească. Această imperfecțiune a claselor e o moștenire de la creatorii lor. Limitările oamenilor se regăsesc în creația lor, iar astfel se explică multitudinea de greșeli de programare (utilizate sub forma englezească *bugs*) pe care orice program le are.

Parte intrinsecă din paradigma programării orientate pe obiecte sunt și modelele de design, cunoscute sub denumirea englezească de design patterns. Acestea sunt, simplu spus, soluții recurente la probleme generice. Modelele de design reprezintă una din cele mai grele părți din programare pe care trebuie să le învețe practicanții. Aplicabilitatea lor nu se reduce la programarea orientată pe obiecte, dar în contextul acesta s-au definitivat. Dificultatea înțelegerii lor provine din nivelul foarte avansat de abstractizare. Foarte puține implementări sunt identice, iar asta înseamnă și că învățarea după exemple devine problematică. Același model de design poate fi util sau poate duce la un design prost, în funcție de niște detalii greu de observat. Practicarea consecventă facilitează acest proces de aplicabilitate, dar e esențial ca un practicant să își sedimenteze o bază

teoretică a acestor modele. Fără această parte, produsul finit va avea de suferit. Multe ramuri ale filosofiei pun întrebări care se regăsesc în genul de fundație pe care un programator cred că ar trebui să o dezvolte.

Umberto Eco zicea că "semiotica are de-a face cu orice lucru care poate fi considerat drept semn. Este semn orice lucru care poate fi considerat drept un substitut semnificant pentru altceva."[1] Această formă de gândire orientată pe obiecte și pe relația și comunicarea dintre ele, împreună cu multitudinea posibilităților de semnificație care sunt limitate doar de posibilitățile fizice și intelectuale, formează un adevărat sistem care, cred eu, merită analizat printr-o lentilă semiotică, pentru puternica semnificație pe care o au, dar și hermeneutică, pentru că interpretarea 'textelor' scrise în acest fel cere un anumit *techne*.[2] Nu e doar vorba de competență, ci ajunge să fie ceva mai mult, partea din anticul cuvânt grecesc care semnifică latura artistică a unei meserii.

3. Semiologia lui Saussure

Ferdinand de Saussure a fost un lingvist elvețian care a pus bazele semiologiei la începutul secolului al XX prin cursurile sale de la catedra de la Geneva, notate și editate de doi studenți de-a lui sub lucrarea *Curs de lingvistică generală*. Saussure a încercat să își scrie el însuși lucrările, dar nu a ajuns la o formă pe care să o considere demnă de publicat. Astfel, el și semiologia împart destinul lui Aristotel și a metafizicii lui.

Una din ideile fundamentale ale lui Sassure este importanța limbii ca element social. În cursul lui de lingvistică, el descrie cum, prin separarea limbii de vorbire separăm și socialul de individual și ce e esențial de ce e mai mult sau mai puțin accidental.[3] Este contra-intuitiv să presupunem că utilizarea unui limbaj OOP ar presupune despărțirea socialului de individual, dar vom vedea că nu este așa. În primul rând, dacă am face o analogie între scrierea de cod și vorbire, în sensul pe care îl dă Sassure termenului (*parole*) vom putea identifica și dimensiunea limbii OOP în felul în care Sassure vedea limba (*langue*),

ca ceva virtual, care nu aparţine niciunui individ, dar aparţine comunităţii.

Un exemplu care să susţină impactul societăţii asupra limbajului OOP ar putea fi elementele introduse fără un scop utilitar, dar cu unul social, cum ar fi glumele de 1 aprilie; una din glume chiar a introdus un standard care presupune ca toate obiectele care îl respectă să ştie să se îmbrăţişeze între ele pentru a-şi oferi suport emoţional.[4]

Un alt exemplu e puternica polemică care poate fi generată de alegeri arbitrare, cum ar fi dacă un nivel de indentare când scrii cod să fie reprezentat prin caracterul tab sau prin patru spaţii. Această polemică nu a murit nici după 20 de ani de argumentări.[5]

Putem duce analogia mai departe şi să privim relaţia dintre limbă şi vorbire la fel ca relaţia dintre scrierea de cod (*parole*) şi limba OOP (*langue*), relaţie care pare complet diferenţiată, în sensul că nu permite ambiguităţile de care se ruşinează limbile umane şi, în acelaşi timp, arbitrară, din moment ce simbolul folosit în vorbire nu are nimic în comun cu conceptul pe care îl reprezintă în sistemul limbii. Un aparent neajuns al analogiei ar putea fi identificat in raportul dintre negativismul obiectului vorbirii saussurian şi cel OOP şi se referă la faptul că, pe când cel saussurian e negativ în sensul în care presupune definirea obiectului vorbirii prin tot ce nu e el, codul e imperativ prin natura lui. Tocmai de aceea nu putem aplica semiologia lui Sassure paradigmei procedurale, care reprezintă doctrina imperativă în lumea programării, dar paradigma OOP utilizează codul pentru definirea de obiecte şi a relaţiei dintre ele, astfel încât lasă loc definiţiilor negative.

O altă idee pe care voi încerca să o transpun peste limbajele OOP este modelul diadic al semnului, mai precis diferenţa dintre semnificat şi semnificant. Umberto Eco spunea despre acest model: “Definiţia lui Saussure este foarte importantă şi a servit la dezvoltarea unei conştiinţe semiotice. Definiţia dată de el semnului, ca entitate cu două feţe (semnificant şi semnificat), a anticipat şi determinat toate definiţiile ulterioare date funcţiei-semn. (...) Saussure nu a definit niciodată clar semnificatul, lăsându-l la jumătatea drumului dintre imagine mentală, concept şi realitate psihologică necircumscrisă altminteri ; în schimb, a subliniat cu putere faptul că semnificatul este ceva ce are legătură cu activitatea mentală a indivizilor în sânul

societății."[6] Dacă considerăm semnul ca fiind semnul obiectual al paradigmei OOP, atunci semnificatul poate fi însuși obiectul reprezentat de o clasă și cum acesta se integrează în arhitectura programului astfel încât își îndeplinește funcția, desăvârșind astfel sistemul lingvistic de care aparține. Semnificantul, în cazul acesta, ia forma imaginii-sunet a codului căruia i se acordă o grijă deosebită. Comunitățile programatorilor lucrează în mod continuu la crearea unor standarde care să ajute codul să semnifice mai bine obiectele construite.

Un neajuns al limbajului OOP din punctul de vedere a lui Saussure cred că ar fi lipsa totală a fonologiei. Paradigma OOP transcende percepția senzorială, dar în aceeași timp prezintă un handicap. Conceptele semiologice ale lui Saussure care sunt orientate înspre comunicarea cu alți oameni, par că lipsesc în contextul OOP, dar argumentul meu e că elementul de comunicare cu alți oameni e salvat de puternica hermeneutică pe care o atrage genul acesta de 'vorbire'. Programatorii citesc foarte mult cod. Contrar opinii publice, ei citesc mai mult decât scriu. Nu îmi imaginez cum ar putea altfel. Un scriitor care citește puțin nu prea are cum să fie un scriitor împlinit. Raportul între cât cod scrie un programator și cât citește e indicat să fie chiar și de 1 la 10[7].

4. Hermeneutica codului

Aflăm din *Hermeneutica* lui Friedrich Schleiermacher ca ea este pentru interpretare ceea ce logica este pentru gândire.[8] Pusă în contextul programării, ea devine parte vitală din procesul de dezvoltare. La fel ca pentru Schleiermacher, interpretarea codului trece de litera textului, ajungând să reprezinte fidel modul de gândire al autorului atât de bine încât codul scris de un programator devine o carte de vizită, referințele către codul personal fiind elemente din portofoliul oricărui programator. E un mod foarte bun de a observa atitudinea și modul în care cineva gândește, dacă respectă anumite standarde, dacă reușește să fie fluent în lingvistica codului. Ne putem

astfel da seama că programarea și interpretarea codului merg mână în mână, iar arhitecții software, care au viziunea de ansamblu și gândesc direcția tehnică nu ar putea să facă o treabă bună dacă nu ar fi și buni hermeneuți.

Spre deosebire de semiologie, hermeneutica a fost conștientizată formal în ecosistemul dezvoltării software sub forma de revizuire a codului (*code review*). Astfel, într-o echipă, codul scris de un membru al echipei nu va ajunge în trunchiul comun al codului echipei până când nu va fi primit acceptul cel puțin a unui alt membru al echipei, iar, în cazul unui un impact asupra sistemului lingvistic, există organizații care decid prin vot dacă anumite schimbări să fie adoptate sau nu.

Urmând în mod natural indicațiile lui Schleiermacher, hermeneutica codului prin aplicarea ei sub formă de revizuire de cod, ne îndeamnă să vedem codul ca fiind mai mult decât intenția autorului; astfel, ajungem să înțelegem textul și ca pe o urmă a personalității autorului și să îl privim într-un context istoric, să îl înțelegem prin prisma codului scris într-o unitate de sens superioară, cea a obiectului programatic și a integrării lui în unitățile cu un nivel mai sus, modelele de design (*design patterns*), care, după cum am mai spus, sunt reprezentările unor soluții recurente la probleme comune[9], iar progresia poate continua.

Hermeneutica mai atinge domeniul informatic și prin partea de *design software*. Dinamica ecosistemului programatorilor a fost schimbată de conceptul programelor cu sursă deschisă, cunoscute sub forma englezească de *open-source software*, care presupune accesul public al codului sursă, schimbare majoră, cu consecințe puternice în domeniu. Accesul public al sursei presupune o invitație la colaborare, unde oricine poate contribui la un proiect. Până în momentul apariției conceptului de *open-source*, hermeneutica codului era restrânsă la o echipă sau organizație; din acest punct ea devine publică, încurajând practica hermeneutică indiferent de contextul economico-social al persoanelor implicate. Calitatea programelor a crescut considerabil, iar atitudinea programatorilor față de creația lor s-a schimbat. Codul nu mai era scris doar pentru a funcționa, ci și pentru a fi expus și analizat, a devenit o formă de expresie mai complexă.

5. Semiotica lui Peirce

Charles Sanders Peirce a fost un filosof american, cu un trecut în chimie care a preluat ideile lui Saussure și le-a îmbunătățit acolo unde a crezut de cuviință. Astfel a luat naștere o semiotică diferite de cea a lui Saussure, dar aflată într-o strânsă relație cu ea, similară cu relația dintre Heraclit și Parmenide sau cea dintre stoici și Epicur. Una din diferențele majore este aceea că, spre deosebire de Saussure și modelul lui dual semnificat și semnificant, Peirce propune un model triadic, format din semn, relația semnului cu obiectul și relația semnului cu interpretantul, unde semnul poate să intre în una din categoriile: iconice, simboluri sau indecși. O altă diferență este cea dintre pansemia universală pe care o prezintă Peirce și cea concentrată pe lingvistică, pe care o susține Saussure.[10]

Relația dintre semn și obiect în paradigma OOP e una din întrebările cele mai des puse la interviuri: Care e diferența dintre o clasă și un obiect? Diferența constă în faptul că clasa e semnul, reprezentarea și obiectul e, în termeni aristotelici, instanțierea potențialității semnului. Un programator scrie codul pentru clase, și cum relaționează ele cu alte clase la un anumit declanșator, dar mașina va instanția obiectul în realitatea virtuală; astfel totalitatea semnelor din limbajele OOP sunt instrucțiuni a priori. Experienței îi revine modestul rol de finisaj mai degrabă decât un rol ontologic.

Relația semnului cu interpretantul poate fi privită tehnic ca ideea de referință. Interpretantul lui Peirce e un alt semn, astfel încât, în paradigma OOP, interpretantul va fi alt obiect, sau corolar. Toți posibilii interpretanți ai unui obiect programatic sunt toate celelalte obiecte care relaționează în vreun fel cu obiectul respectiv. Triada e completată de semn în sine și de natura lui. Semnul paradigmei OOP e, în teoria lui Peirce, e un index. Obiectul nu ar putea să existe dacă nu ar fi existat semnul, dar poate exista fără semn odată ce acesta a fost instanțiat.

Pansemia e clar respectată. Nu există obiecte care să nu semnifice ceva. Totul are o semnificație, sau cel puțin poate avea. Problemele de care se lovesc programatorii nu țin de lipsa de semnificație, chiar din

contră, pericolul e polisemia, care poate dăuna limbajului. Comunitatea a ajuns astfel la un principiu care reglează producția de semne ambigue: principiul de responsabilitate unică (*single responsability principle*), care dictează că un semn poate avea o singură responsabilitate, principiu care se traduce în claritatea unităților de înțelegere, reducerea numărului de dependințe și creșterea mentenabilității sistemului. Un corolar ar fi că ar trebui să existe un singur motiv pentru care un 'cuvânt' programatic ar trebui să fie modificat. Cu toate astea, principiul de responsabilitate unică nu încalcă principiul polisemiei. În programarea orientată pe obiecte fiecare obiect are mai multe dimensiuni. Obiectul poate exista dintr-un singur motiv, dar semnificațiile lui vor fi multiple. Nevoia din care se naște un obiect poate semnifica o altă nevoie care naște un alt obiect, sau o nevoie de modificare a arhitecturii întregului sistem. Posibilitățile de semnificare sunt limitate doar de perspicacitatea, imaginația și inspirația programatorului.

Concluzie

Julia Annas scrie în cartea ei, *Ancient Philosophy: A Very Short Introduction*: "The philosophically interesting kind of knowledge involves a complex of items grasped together in a way that enables the knower to relate them to one another and to the structure as a whole. In all but the simplest cases this grasp will require an articulate ability to do this relating, one which will explain why the different items play the roles they do in the system"[11]. Cred că programarea orientată pe obiecte se potrivește perfect descrierii propusă de Annas, cu atât mai mult cu cât, în aplicarea conceptelor OOP, nu vei fi limitat la o singură meserie sau o singură dimensiune, ci vei trece prin experiența asta de fiecare dată când vei încerca să creezi o arhitectură nouă. E un exercițiu filosofic de fiecare dată. Nu vei putea practica acest tip de programare fără să fi pus în postura de a răspunde întrebărilor pe care le pune și semiotica și semiologia, dar vei fi pus față în față și cu practica hermeneutică. Din acest punct de vedere, nu cred că progra-

marea orientată pe obiecte și-a terminat traseul academic, ci dimpotrivă, atenția filosofică asupra ei este de-abia la început. Această colaborare cred că va fi productivă nu doar pentru dezvoltarea tehnicii, dar și pentru dezvoltarea ramurilor filosofiei cu care se intersectează.

Legătura dintre om și mașini va deveni din ce în ce mai strânsă. Comunicarea dintre oameni și tehnologiile pe care le vor folosi din ce în ce mai des trebuie să fie cât mai accesibilă. Semiologia va deveni o parte importantă din ecosistemul tehnic, iar programatorii vor trebui să se folosească de înțelegerea semiologiei asupra modului uman de comunicare și semnificare pentru a putea să desăvârșească ambiții tehnice precum inteligența artificială sau comunicarea prin impulsuri neurologice între persoane la distanțe mari.

Note

[1] Umberto Eco, *Tratat de semiotică generală* (București: Editura Științifică și Enciclopedică, 1982), 18.
[2] Simon Blackburn, *The Oxford Dictionary of Philosophy* (Oxford: Oxford University Press, 1996), 374.
[3] Ferdinand de Saussure, *Course in General Linguistics* (Columbia University Press, 2011), 14.
[4] "psr-8-hug.md - GitHub." https://github.com/php-fig/fig-standards/blob/master/proposed/psr-8-hug/psr-8-hug.md. Accesat 7 Feb. 2019.
[5] David Cassel, "Spaces vs. Tabs: A 20-Year Debate Reignited by Google's Golang" 17 Sep. 2016 available at https://thenewstack.io/spaces-vs-tabs-a-20-year-debate-and-now-this-what-the-hell-is-wrong-with-go/. Accesat 7 Feb. 2019.
[6] Umberto Eco, *Tratat de semiotică generală*, 26.
[7] Robert C. Martin, *Clean Code: A Handbook in Agile Software craftsmanship* (New Jersey: Prentice Hall, 2008), 14.
[8] Friedrich Schleiermacher, *Hermeneutica* (Iași: Polirom, 2001), 9.
[9] ***, "Design Patterns" - *SourceMaking.com* https://sourcemaking.com/design_patterns. Accesat 8 Feb. 2019.
[10] Aurel Codoban, *Semiologie. Teoria semnului și a interpretării* (Suport de curs, 2013), 15.

[11]Julia Annas, *Ancient Philosophy: A very short introduction* (Oxford: Oxford University Press, 2000), 72.

Bibliografie

*** "Design Patterns" - *SourceMaking.com* https://sourcemaking.com/design_patterns Accesat 8 Feb. 2019.

Andersen, Peter Bogh. 1991. „Computer Semiotics" în *Scandinavian Journal of Information Systems*, vol. 4, nr. 1, 3-30.

Annas, Julia. 2000. *Ancient Philosophy: A very short introduction*. Oxford: Oxford University Press.

Blackburn, Simon. 1996. *The Oxford Dictionary of Philosophy*. Oxford: Oxford University Press.

Cassel, David. 2016. "Spaces vs. Tabs: A 20-Year Debate Reignited by Google's Golang" 17 Sep. 2016, available at https://thenewstack.io/spaces-vs-tabs-a-20-year-debate-and-now-this-what-the-hell-is-wrong-with-go/ Accesat 7 Feb. 2019.

Cobley, Paul and Jansz, Litza. 2012. *Introducing Semiotics (A Graphic Guide)*. London: Icon Books Ltd.

Codoban, Aurel. 2013. *Semiologie. Teoria semnului și a interpretării.* Suport de curs.

Eco, Umberto. 1982. *Tratat de semiotică generală*. București: Editura Științifică și Enciclopedică.

Martin, Robert C.. 2008. *Clean Code: A Handbook in Agile Software craftsmanship*, New Jersey: Prentice Hall.

Saussure, Ferdinand de. 2011. *Course in General Linguistics*. New York: Columbia University Press.

Schleiermacher, Friedrich. 2001. *Hermeneutica*. Iași: Polirom.

Tanaka-ishii, Kumiko. 2010. *Semiotics of Programming*. New York: Cambridge University Press.

Wirfs-Brock, Rebecca and Wilkerson, Brian and Wiener Lauren. 1990. *Designing Object-Oriented Software*. New Jersey: Prentice Hall.