3 - Objektna orijentacija

Objektno-orijentisano programiranje je izuzetno loša ideja,
koja je mogla nastati samo u Kaliforniji.
Edsher Daikstra

3.1 Objektno-orijentisano programiranje

Objektno-orijentisano programiranje je nastalo na talasu inovacija koje su sredinom 20. veka težile da sistematizuju programiranje i programske jezike. Uzrok je predstavljala najpre intuitivno a kasnije i sistemski prepoznata *softverska kriza*. Sredinom 60-ih godina je praktično završen period inicijalnog razvoja računarstva. Do tada je bavljenje razvojem računarstva i programiranjem bilo praktično ista stvar, zato što je pisanje praktično svakog novog programa predstavljalo manji ili veći istraživački poduhvat. Sa porastom rasprostranjenosti računara došlo je do povećane potrebe za pisanjem programa i za programerima, koja nije mogla da bude zadovoljena na odgovarajući način bez unapređivanja metoda rada. Pisanje sve više i sve većih programa uz upotrebu zastarelih tehnologija imalo je za rezultat sve više neuspešnih projekata.

Prvi veliki iskorak prema unapređenju tehnologije programiranja bila je pojava *strukturnog programiranja*. Iako je strukturno programiranje u naučnim krugovima nastajalo postepeno još od kasnih 50-ih godina, tek je prepoznavanje softverske krize dovelo do značajnijeg pomaka u zastupljenosti strukturnog programiranja u široj praksi. Strukturno programiranje počiva na prepoznavanju osnovnih struktura algoritama i programskih jezika, koje su dovoljno izražajne da mogu da opišu svaki algoritam i program, a sa druge strane dovoljno konceptualno čiste i jednostavne da mogu da se lako razumeju i da može da se formalno dokaže njihova korektnost.

U PRIPREMI

Vrlo brzo je strukturiranje koda dopunjeno i strukturiranjem podataka, a kada je pokušano spajanje koda i podataka u jedinstvenu celinu praktično je stvorena osnova za razvoj objektno-orijentisanog programiranja (OOP). U ranim fazama je terminologija OOP bila drugačija od one koju danas poznajemo, pa su, na primer, klase nazivane „glavnim objektima", „modulima", „definicijama"... Kasnije, sa porastom broja novih programskih jezika, došlo je i do delimične standardizacije terminologije[5].

Paralelno sa porastom zastupljenosti OOP, težilo se i prenošenju objektno-orijentisane paradigme na druge oblasti računarstva, što je dovelo do rada na objektno-orijentisanim bazama podataka i na oblikovanju čitavih objektno-orijentisanih razvojnih metodologija.

3.2 Osnovni koncepti OOP

Objekti i klase

U objektno-orijentisanom programiranju u centru pažnje su *objekti*. Objekti nam služe da pomoću njih u programima modeliramo entitete iz domena problema. Svaki objekat bi trebalo da ima prepoznatljiv identitet, stanje i ponašanje, kao i životni vek, koji čine nastajanje, postojanje i nestajanje.

Radi ilustracije, navešćemo nekoliko definicija pojma objekta iz literature iz devedesetih godina XX veka:

- Objekat je apstrakcija nečega u domenu problema, koja opisuje sposobnost sistema da o tome čuva informaciju, interaguje sa time ili oboje [*Coad 1991*].
- Objekat je koncept, apstrakcija ili nešto sa jasnim granicama i smislom u odnosu na konkretan problem. Objekat ima dve svrhe: da pomogne razumevanju stvarnog sveta i pruži praktičnu osnovu za računarsku implementaciju [*Rumbaugh 1991*].
- Objekti se opisuju odgovorima koje mogu da daju na pitanja:
  - Ko sam ja?
  - Šta mogu da uradim?
  - Šta znam? [*Wirf-Brock 1990*]
- Objekti imaju stanje, ponašanje i identitet [*Booch 1990*].

[5] I dalje neki jezici imaju specifičnu terminologiju. Na primer, u programskim jeziku C++ se ne koristi termin „metod" već „funkcija-članica" i slično.

UPRIPREMI

Jedan objekat predstavlja apstrakciju jednog entiteta iz domena problema. Međutim, programsko opisivanje svakog pojedinačnog entiteta bi bilo nezahvalan posao. Zato se kao dodatni nivo apstrakcije uvodi *klasa*. Klasa predstavlja apstrakciju nekog skupa objekata koji su međusobno *dovoljno slični*. Pri ustanovljavanju sličnosti, prevashodno se razmatra *ponašanje* objekata, a ne njihova struktura. Kada se ustanovi koji su to objekti koji su dovoljno slični, onda se prepoznaje *klasa* tih objekata i na programskom jeziku se opisuje njihovo ponašanje, ali i njihova struktura. Objekti se zatim prave kao primerci (instance) opisane klase.

Standardizacijom objektno-orijentisanih tehnika i tehnologija se bavi Grupa za upravljanje objektima (engl. *Object Management Group* – *OMG*[6]). Njihova zvanična definicija pojmova klase i objekta glasi[7]:

- *Klasa* je opis skupa objekata koji dele iste atribute, operacije, metode, odnose i semantiku. Svrha klase je da deklariše kolekciju metoda, operacija i atributa koja u potpunosti opisuje strukturu i ponašanje tih objekata.
- *Objekat* je primerak koji potiče iz klase, strukturiran je i ponaša se u skladu sa svojom klasom.

Struktura objekata se opisuje pomoću *atributa*. Atributi predstavljaju opisne karakteristike objekata. Definicijom klase se ustanovljava koje atribute moraju da imaju objekti te klase, a pri radu sa objektima se određuje koji objekat ima koje vrednosti tih atributa. Atributi su sredstvo za čuvanje stanja i znanja objekata.

Ponašanje objekata se opisuje pomoću *metoda*. Metodi predstavljaju algoritamske opise ponašanja objekata jedne klase. Pri opisivanju klase opisuju se i svi metodi koji modeliraju ponašanje objekata te klase.

U većini OO programskih jezika su koncepti klase i pripadanja objekta klasi veoma strogo definisani. Klasa objekta se određuje u trenutku njegovog pravljenja i kasnije ne može da se promeni, tako da svaki objekat pripada tačno jednoj klasi. U takvim jezicima postoji vrlo jasna korelacija između koncepta *klase* i koncepta *tipa* – klase predstavljaju jednu veoma važnu vrstu tipova i smatra se da objekat pripada klasi ako ima tip te klase.

---

[6] *OMG* je konzorcijum otvorenog tipa, u koji su učlanjene brojne značajne softverske kompanije. Osnovan je 1989. godine i bavi se razvojem i održavanjem standarda u oblasti objektno-orijentisanih tehnologija [*OMG*].

[7] Moramo da primetimo da je ova definicija cirkularna – klasa se definiše preko objekta a objekat preko klase. To je dobra ilustracija OO sveta, u kome je malo šta do kraja strogo i precizno definisano. O tome će biti više reči pri kraju ovog poglavlja.

U PRIPREMI

U nekim programskim jezicima atributi i metodi mogu da se definišu i na nivou pojedinačnih objekata, a ne samo na nivou klasa. Ako bismo pokušali da pravimo razliku između takvih i ostalih OO jezika, možda bismo mogli da kažemo da su samo takvi jezici pravi „objektno-orijentisani" programski jezici, dok su ostali jezici pre „klasno-orijentisani". Tipičan primer „pravog OO programskog jezika" u tom smislu je programski jezik *ECMAScript* (šire poznat kao *JavaScript*), u kome (u osnovnim varijantama) uopšte ne postoji koncept klase, već se sve vrti oko objekata.

Postoje i programski jezici kod kojih klase postoje, ali se pripadnost objekta klasi ustanovljava dinamički, proveravanjem ispunjenosti nekih definisanih semantičkih pravila. U takvim jezicima objekat može da promeni klasu tokom svog života. Na primer, ako bi se kvadratu promenila veličina stranica, onda bi on mogao da prestane da bude kvadrat i postao bi pravougaonik. Primer takvog jezika je *D*, objektno-orijentisan jezik za rad sa podacima[8], koji su definisali Dejt i Darvin [*Date 2006*].

Enkapsulacija i interfejs

Jedan od ozbiljnih problema sa klasičnim tehnikama programiranja, uključujući i strukturno proceduralno programiranje, jeste puna raspoloživost svih podataka i algoritama svim programerima. To je imalo za posledicu da svaki programer može da pristupa bilo kom podatku i bilo kojem aspektu ponašanja, bez ikakvog ograničenja. Takav način rada može da potpuno zaobiđe određena pravila upotrebe nekih struktura podataka ili pomoćnih aspekata ponašanja i da dovede do neispravnog stanja softvera.

Da bi se prevazišao taj problem, u OO tehnologijama su oblikovani koncepti enkapsulacije i interfejsa. *Enkapsulacija* predstavlja princip skrivanja detalja implementacije klase od korisnika te klase, tj. od programera koji će raditi sa objektima te klase. Sve ono što zahteva dublje poznavanje interne implementacije klase se sakriva. Korisnicima se dozvoljava da koriste samo pažljivo definisan skup metoda, koji omogućavaju upotrebu objekata klase, a ne zahtevaju dublje poznavanje pojedinosti njene implementacije. Taj skup metoda predstavlja *interfejs* klase.

Interfejs klase bi trebalo da što bolje odgovara javnim aspektima ponašanja objekata te klase. On mora da omogući korisnicima klase da urade sa objektima sve što predstavlja odlike ponašanja te klase, a da istovremeno od njih bude skriveno što više informacija o internim elementima implementacije.

[8] Primetimo da taj jezik, osim imena, nema skoro ništa drugo zajedničko sa sve popularnijim programskim jezikom *D*, koga su razvili Volter Brajt i Andrej Aleksandresku [*Alexandrescu 2010*].

U PRIPREMI

Jedno od pravila enkapsuliranja nalaže da korisnici ne smeju neposredno da koriste atribute klase, već da sve upotrebe atributa moraju da budu enkapsulirane u metode interfejsa. Dosledna primena ovog principa nalaže da enkapsulacija atributa mora da se preduzima čak i kada bi atributi trebalo da predstavljaju interfejs klase. U tim slučajevima se prave tzv. pristupni metodi za čitanje i menjanje vrednosti atributa[9].

Enkapsulacijom se sprečava nekontrolisano menjanje stanja objekata, mimo pravila koja su interno ustanovljena implementacijom klase. Istovremeno, u sve metode koji čine interfejs je ugrađeno rukovanje stanjem i skrivenim metodima uz puno poštovanje svih tih internih pravila. Na taj način se korisnik klase oslobađa obaveze staranja o tim pravilima. Smanjivanje količine potrebnog znanja o internim elementima implementacije omogućava programerima da uz manje učenja, lakše i pouzdanije koriste klase koje su napravili drugi programeri. Sa druge strane, autorima klase omogućava da bez većih ograničenja mogu da menjaju elemente interne strukture i implementaciju klase, a da to nema uticaja na njene korisnike – jedino što može neposredno da utiče na upotrebu jesu eventualne promene interfejsa klase.

Osim vida zaštite od neispravne upotrebe, enkapsulacija i interfejs pružaju korisnicima klase dodatni nivo apstrakcije. Dobar interfejs ne pruža korisniku nikakve naznake o načinu implementacije klase, već na zaokružen način predstavlja celinu klase isključivo u kontekstu njene funkcionalnosti, tj. mogućih načina upotrebe. Mogli bismo da uporedimo interfejs klase sa daljinskim upravljačem televizora – korisniku je važno da može da promeni program ili jačinu zvuka, a ni najmanje ga ne zanima kako televizor radi.

Enkapsulacija i interfejs se ne primenjuju samo na nivou klase, već i na nivou komponenti, pa i servisa. Svaka celina programa koju pišemo bi trebalo da enkapsulira elemente interne implementacije i da pruža korisnicima samo zaokružene interfejse. Na taj način se podiže nivo apstrakcije svake celine koda i omogućava se niži nivo međusobne spregnutosti, a time i viši nivo međusobne nezavisnosti delova programa, što je veoma važno za lakše pisanje i održavanje programa.

Enkapsulacija i interfejs mogu da se koriste i kao sredstvo za analizu kvaliteta arhitekture programa. Ako je interfejs sačinjen tako da ne predstavlja jednu nedeljivu

[9] Pristupni metodi se na engleskom jeziku obično nazivaju *getter* i *setter*. U programskim jezicima kao što su *Java* ili *C#* uobičajeno je da se za svaki atribut prave pristupni metodi, bilo javni ili privatni. Sa druge strane, kada se koristi programski jezik *C++*, onda se teži da se pravi što manje pristupnih metoda i to samo ako su baš neophodni, tj. ako atribut predstavlja deo interfejsa.

U PRIPREMI

celinu, već ga čine elementi različitih aspekata ponašanja klase, onda to vrlo često može da nam ukaže na potencijalan problem neispravne dekompozicije – možda bi ta klasa trebalo da se podeli na dve ili više manjih? Slično tome, ako je za obavljanje nekog posla potrebno da se koristi interfejs većeg broja klasa i objekata, možda bi trebalo da se napravi nova klasa koja bi prikrila složene aspekte implementacije i upotrebe tih klasa?

Specijalizacija i generalizacija

Kao što klasa predstavlja poseban oblik tipa, tako se i među klasama uspostavljaju odnosi *potklasa* i *natklasa*, slično odnosima *podtip* i *nadtip*. Nasleđivanje klasa je osnova za uspostavljanje hijerarhijskog polimorfizma, koji je osnovni vid polimorfizma u većini OO programskih jezika[10]. Hijerarhijski polimorfizam počiva na osobini objektno-orijentisanih programskih jezika da programski kod, koji je napisan da radi sa objektima jedne klase, može da radi i sa objektima svih njenih potklasa.

Nasleđivanje klasa se uspostavlja na osnovu jednosmerne parcijalno uređene binarne relacije *jeste*. Kažemo da klasa A *jeste* klasa B ako i samo ako svaki objekat klase A ima sve osobine koje imaju i svi objekti klase B. Ako klasa A *jeste* klasa B, onda se kaže i da je klasa A *izvedena* iz B, a da je klasa B *osnovna* klasa za A. Slično, kažemo i da je klasa A *potomak* klase B, a da je klasa B *predak* klase A.

Relacija *jeste* se često referiše terminima *specijalizacija* i *generalizacija*, gde je *specijalizacija* praktično isto što i relacija *jeste*, dok je *generalizacija* ista relacija ali u suprotnom smeru. Znači, ako A *jeste* B, onda je A *specijalizacija* (ili *poseban slučaj*) klase B, a B je *generalizacija* (ili *uopštenje*) klase A.

3.3 Objektno-orijentisane metodologije

Prve objektno-orijentisane metodologije (OOM) su počele da se razvijaju ubrzo po nastajanju prvih OO programskih jezika. Da bi došle do toga da budu sveprisutne u razvoju softvera, OOM su morale da pređu dugačak put. Istoriju OOM bismo mogli da posmatramo kroz tri osnovna perioda, koji se delimično preklapaju.

Prvi period se prostire od početka razvoja pa do oko 1997. godine i karakterišu ga *početni koraci* u definisanju metodologija. U tom periodu se pojavljuje veliki broj različitih metodologija, kao i veliki broj potpuno nezavisnih notacija za zapisivanje projekata i planova, pa i različitih terminologija. Ispostavilo se da većina od tih metodologija nije bila ni kompletna ni dovoljno široka i obuhvatna da zaživi mnogo

[10] Polimorfizmu, sa posebnim akcentom na parametarskom poimorfizmu, je posvećeno poglavlje *11 - Polimorfizam*.

UPRIJEMI

dalje od ograničenog domena primene ili nekog užeg razvojnog okruženja u kome je nastala. Najznačajniji rezultat ovog perioda razvoja je nastajanje velikog broja metoda i tehnika razvoja. Skoro svaka tehnika koju danas koristimo se u nekom osnovnom obliku po prvi put pojavila u nekoj od metodologija iz tog perioda. Neke od najvažnijih metodologija iz tog perioda su predstavljene u knjigama Grejdija Buča [*Booch 1990*], Kouda i Jurdona [*Coad 1991*] i Martina i Oudela [*Martin 2003*]

Drugi period je trajao od 1995. godine do oko 2005. godine. Za njega je karakterističan rad na objedinjavanju metodologija i tehnika. U prvoj polovini ovog perioda je uočljivo usmeravanje velike energije na ujednačavanje notacije i definisanje (a kasnije i unapređivanje i standardizovanje) *UML-a* – objedinjenog jezika za zapisivanje modela [*UML*]. Ovaj period možemo da nazovemo *Oblikovanje UML-a*. Ovaj značajan poduhvat je pokrenut u okviru kompanije *Rational*, u kojoj su najpre Grejdi Buč i Džim Rambou radili na objedinjavanju svojih metodologija, da bi im se 1995. pridružio i Ajvar Jakobson. Njihov cilj je bio da razviju kombinovanu metodologiju, koja je najpre bila poznata pod imenom *Object Oriented Software Engineering* (*OOSE*), a koja je danas poznata pod imenom *Rational Unified Process* (*RUP*). U okviru razvoja metodologije je paralelno tekao i razvoj objedinjene notacije za zapisivanje svih elemenata projekta, koja je dobila ime *Unified Modeling Language* (*UML*). Radom na *UML*-u je upravljao konzorcijum partnera, među kojima su bile vodeće svetske softverske kompanije.

Nezadrživo širenje *UML-a* je počelo 1997. godine, objavljivanjem prve zvanične verzije. Taj trenutak se može smatrati i završetkom prvog perioda razvoja OOM. Ubrzani razvoj *UML*-a je usporen najpre objavljivanjem stabilnih verzija 1.1, 1.2 i 1.3 do 1999. godine, a zatim posebno 2005. godine objavljivanjem verzije 2.0. Tokom ovog perioda se drastično smanjio broj novonastalih metodologija.

Treći period traje od 2000. godine i karakteriše ga sveprisutnost *UML*-a, kao jedne od najrasprostranjenijih razvojnih softverskih tehnologija. Mogli bismo ga nazvati *Post-UML period*. Praktično sve što se od 2000. godine radi u oblasti OOM, radi se uz pretpostavku da se kao notacija koristi *UML*. Više se ne posvećuje vreme izmišljanju novih notacija, već se uz pretpostavljanje ujednačene notacije radi na pravljenju novih metodologija. Razvija se veliki broj *agilnih* metodologija, pri čemu praktično sve one koriste brojne elemente *RUP-a* i drugih metodologija koje su iz njega potekle.

Danas se u praksi primenjuje mnogo različitih metodologija, pri čemu su one većinom objektno-orijentisane. Važna posledica razvoja *UML*-a je da se rezultati rada svih tih metodologija, pa i komunikacija među razvijaocima koji ih primenjuju, odvijaju sa punim međusobnim razumevanjem, bez straha da može doći do nesporazuma usled različite notacije. Štaviše, danas i neke metodologije koje po svojoj prirodi nisu objektno-orijentisane teže da u što većoj meri koriste *UML*.

UPRPEVI

Savremene OOM se odlikuju velikim brojem zajedničkih karakteristika. Pre svega, tu je opšta težnja da se sve modelira objektima i klasama. Obično se posmatraju četiri osnovna skupa objekata u sistemu:

- entiteti – sve vrste podataka koji postoje u softverskom sistemu;
- subjekti – različite vrste korisnika softverskog sistema;
- servisi – usluge koje softverski sistem pruža korisnicima i
- interfejsi – podsistemi putem kojih subjekti koriste servise.

Praktično sve OOM teže da na neki način skrate trajanje razvojnog ciklusa. I agilne i ne-agilne metodologije podstiču upotrebu iterativnog razvoja. Agilne metodologije propisuju i određivanje koraka prema rokovima i raspoloživim sredstvima. Takođe, insistira se i na pojačanoj komunikaciji među subjektima u svim fazama razvoja.

U poglavlju *5 - UML* ćemo predstaviti neke od osnovnih dijagramskih tehnika *UML-a*. U poglavljima *4 - Uvod u projektovanje softvera*, *6 - Principi projektovanja* i *7 - Obrasci za projektovanje* ćemo posvetiti pažnju projektovanju softvera i posebno modeliranju strukture softvera. U poglavlju *8 - Agilni razvoj softvera* predstavićemo koncepte agilnih metodologija.

3.4 Slabosti objektno-orijentisanih koncepata

Objektno-orijentisano programiranje i uopšte objektno-orijentisani koncepti imaju i neke relativno značajne slabosti. Tradicionalno su se isticali nedostaci poput veličine izvršnog programa, brzine izvršnog programa, pa i nesrazmerno velikog ulaganja programerskog rada, ali prva dva od ovih problema su važna samo u ograničenom domenu, dok je treći prevaziđen rapidnim razvojem različitih tehnika i tehnologija razvoja softvera, od kojih neke predstavljamo i u ovoj knjizi.

Veličina i sporost izvršnog koda su nekada imali veliki značaj, pa su bili među najvažnijim razlozima za ograničen uspeh projekta *Smalltalk*, jednog od prvih pravih objektno-orijentisanih programskih jezika sa pratećim vizualnim razvojnim i korisničkim okruženjem. Ipak, ispostaviće se da je 1995. godine programski jezik *Java* sa svojim izvršnim okruženjem (Java virtualna mašina – *JVM*) uspeo da se pozicionira na tržištu tamo gde *Smalltalk* nije uspeo 1980. godine, a da je osnovna arhitektura sistema praktično iskopirana. Razlika je, pre svega, u tome što je između ova dva projekta prošlo 15 godina, tokom kojih su razvijani sve brži računari. I danas postoje primene u kojima savremeni objektno-orijentisani programski jezici, kao što su *Java* i *C#*, imaju teškoća da se izbore za svoje mesto, ali su sve veća brzina računara i unapređivanje tehnika implementiranja (tj. interpretiranja internog koda) ovih jezika značajno smanjili opseg takvih primena. Sa druge strane, razvoj tehnologija

U PRIPREMI

prevođenja je doveo do toga da je programski jezik *C++* u nekim stvarima čak i efikasniji nego *C*. Neefikasnost OOP je danas prisutna više kao posledica neadekvatne upotrebe nego kao karakteristika koncepta ili jezika. Na primer, nije retkost da dosledno modeliranje OO alatima dovede do programskog koda koji sadrži previše nepotrebnih apstrakcija, što zaista može da uspori rad programa.

Jedna od oblasti gde mora veoma pažljivo da se radi da bi se dobio dobar rezultat jesu aplikacije koje intenzivno koriste relacione baze podataka. Relacione baze podataka rade sa skupovima podataka (relacijama), dok OO programi rade sa pojedinačnim objektima. Zato obično mora da se pravi dodatni sloj aplikacije, tzv. sloj objektno-relacionog preslikavanja (engl. *object-relational mapping – ORM*), koji služi da se od programera sakrije relaciona priroda podataka. Posledica skrivanja relacionog modela, koji je nimalo slučajno potvrđen kao najbolji za većinu vrsta baza podataka, može da bude suviše striktno korišćenje OO koncepata i mehanizama, tako da svaki pristup podacima ide od pojedinačnih objekata, preko *ORM*-a pa do sistema za upravljanje bazom podataka. Takav rad može veoma značajno da uspori rad sa podacima, u odnosu na neposrednu upotrebu i izvođenje operacija neposredno na relacijama, ali to je pre svega posledica drugačijeg nivoa i čak različite vrste apstrakcije, a ne nekih konkretnih slabosti OO programskih jezika.

Ako danas razmatramo slabosti OOP i OOM, onda se one najviše prepoznaju na konceptualnom nivou i najčešće imaju pretežno teorijski značaj. Sve konceptualne slabosti OOP potiču iz činjenice da ne postoji matematički stroga i precizna definicija koncepata na kojima počivaju objektno-orijentisane tehnologije. Za razliku od OOP, neke druge danas sveprisutne softverske paradigme, kao strukturno programiranje, funkcionalno programiranje ili relacione baze podataka, počivaju na vrlo čvrstim matematičkim osnovama. Posledica takvog stanja je da OOP i OOM imaju određene nedostatke koji u specifičnim slučajevima mogu da dovedu do protivrečnosti i nekih neprijatnih ograničenja.

Rečenica Edshera Daikstre, čijim citiranjem je započeto ovo poglavlje, nastala je kao posledica činjenice da je Daikstra bio veoma posvećen problemima dokazivanja korektnosti programa i promovisanju strukturnog i disciplinovanog programiranja [*Dijkstra 1976*]. Zbog toga je relativno rano primetio da OOP, iako se intuitivno čini da nas ono vodi prema savremenijem, produktivnijem pa i kvalitetnijem načinu programiranja, zbog nedostatka odgovarajuće matematičke formalizacije praktično onemogućava dokazivanje korektnosti programa, a na duže staze i otežava pouzdano razumevanje programskog koda. Nakon višegodišnjeg fokusiranja značajnog broja istraživača u oblasti računarskih nauka na matematičko formalizovanje i strukturiranje algoritamskih programskih jezika, OOP je u tom smislu izvesno predstavljalo značajan korak unazad. Ovde ćemo predstaviti neke od značajnijih aspekata problema do kojih dovodi nedovoljno strogo definisana semantika OO programskih jezika.

U PRIPREMI

Jedna od konceptualnih slabosti OO tehnologija je nedovoljno precizno definisan pojam *identiteta objekta*, pa zato na neka pitanja ne postoji jednoznačan odgovor. Na primer, da li mogu da postoje dva objekta koji imaju *isti sadržaj* ali ne predstavljaju isti objekat, ili to nije dopušteno? Ili, drugačije formulisano, da li objekti koji imaju isti sadržaj (vrednosti svih atributa) automatski predstavljaju jednu istu instancu klase? Problem identiteta objekata je posebno važan u oblasti OO baza podataka i povezivanja OO programa sa bazama podataka.

I koncept hijerarhijskog polimorfizma ima više slabosti. Na primer, kada govorimo o nasleđivanju, kao osnovni kriterijum za uspostavljanje odnosa nasleđivanja se koristi relacija „*jeste*" i to prevashodno u pogledu ponašanja objekata. Tako, na primer, u odnosu na ponašanje objekata ustanovljavamo da je ispravno da kvadrat nasleđuje pravougaonik, zato što svaki kvadrat *jeste* istovremeno i pravougaonik, tj. sve što znamo da važi za pravougaonik, važi i za kvadrat[11].

Međutim, to nije sasvim jednostavno, kao što bi na prvi pogled moglo da izgleda. Ako bi, na primer, pravougaonik imao metode za menjanje veličine stranica `postaviSirinu` i `postaviDuzinu`, onda ti metodi ne bi mogli da rade na odgovarajući način u slučaju kvadrata. Ako razmotrimo šta bi trebalo da bude rezultat izvršavanja metoda:

```
kvadrat.postaviSirinu( 20 );
```

onda možemo da primetimo da postoje dve osnovne varijante:

1. Ako bi se na kvadratu izvršavao *nasleđeni* metod, onda bi se promenila širina kvadrata, dok bi istovremeno dužina zadržala staru vrednost:
   - Naš „kvadrat" sada ne bi više bio kvadrat, već samo pravougaonik!?
   - Da li tip objekta uopšte može da se promeni na taj način ili bi to i dalje ostao objekat klase „kvadrat" iako mu stranice nisu jednake?
2. Ako bi kvadrat imao *sopstvenu implementaciju* ovog metoda, onda bi njegovom primenom mogli da se promene odjednom i širina i dužina:
   - Onda naš „kvadrat" ostaje kvadrat.

[11] Početnička greška je da se umesto ponašanja razmatra struktura objekata, pa da se pretpostavi da pravougaonik može da nasledi kvadrat, zato što kvadrat ima samo *širinu*, a pravougaonik i *širinu* i *dužinu*, što je *„specijalan"* slučaj. Takav pristup je pogrešan i nije u skladu sa principima OOP, zato što pravila o „ponašanju" kvadrata ne važe za pravougaonik – na primer, obim ne može da se računa po formuli O = 4a. Nasleđivanje i relacija „jeste" se odnose striktno na ponašanje, a ne na strukturu objekata klase.

UPRIPREMI

- Ali, primetimo da za svaki pravougaonik važi pravilo: „ako se širina pravougaonika dvostruko uveća, onda će se dvostruko uvećati i površina“. Takvo pravilo, međutim, ne bi važilo za naš kvadrat, zato što bi se njegova površina uvećala četvorostruko, a to onda znači da se naš kvadrat ponaša drugačije od pravougaonika, pa onda on više *nije* pravougaonik!?

Očigledno je da oba pristupa dovode do problematičnih rezultata, što narušava konzistentnost odnosa nasleđivanja među klasama. Opisan problem je u literaturi poznat kao problem „kvadrat-pravougaonik“ ili „krug-elipsa“. Iako ovaj problem na prvi pogled može da izgleda elementarno i intuicija može da nam govori da postoji neko relativno jednostavno rešenje, stvari zapravo stoje sasvim drugačije i ne bismo mogli reći da postoji dobro rešenje ovog problema za uobičajene OO programske jezike. Vratićemo se na ovaj problem kada budemo razmatrali principe projektovanja (*6 - Principi projektovanja*) i videćemo da samo delimično možemo da ga rešimo.

Drugi često spominjan problem se odnosi na to što objekti grade implicitno globalno stanje programa. Jedan od osnovnih principa strukturnog programiranja nalaže da se ne koriste globalne promenljive i globalno stanje programa. OOP je u osnovi oblikovano uz pretpostavku poštovanja tog principa. Programiranje sa objektima je zamišljeno tako da nijedan objekat ne može da pristupi strukturi drugog objekta, te da objekti međusobno razmenjuju poruke kojima zahtevaju jedni od drugih da se nešto izračuna ili da se promeni stanje nekog objekta. Međutim, da bi objekti mogli da šalju poruke jedni drugima, oni moraju da „znaju“ jedni za druge, ili da bar imaju na raspolaganju neki vid kataloga u kome mogu jedni druge da pronađu. Tako dolazimo do povezanog grafa koji čine svi međusobno povezani objekti (bilo da su povezani neposredno ili posredstvom jednog ili više kataloga), a taj graf nije ništa drugo nego velika i prilično složena povezana struktura koja predstavlja *globalno stanje programa*. Znači, ispada da je neposredna posledica enkapsulacije to što svaki objekat ima stanje, a posredna to što graf svih povezanih objekata čini globalno stanje našeg programa? Neki autori ističu da to baš i nije ono što želimo da imamo, mada možemo da primetimo da ova povezanost podataka suštinski ne može da se izbegne ni ako koristimo klasično strukturno programiranje – čak je tada vidljivija i očiglednija, zato što nije enkapsulirana, a kod OOP bar može da bude u priličnoj meri prikrivena (što, doduše, može da bude i dobro i loše).

Kada se nešto temeljnije razmotre uzroci koji dovode do predstavljenih (i nekih drugih) konceptualnih problema, onda može da se ustanovi da koncepti tipova, sistema tipova i hijerarhija tipova, sa svim odlikama OOP, mogu da se zasnuju formalno i matematički precizno tako da budu neprotivrečni, ali da to onda ima prilično visoku cenu, koja iz ugla današnjih tehnologija razvoja softvera i savremenih OO programskih jezika uglavnom nije prihvatljiva – uvođenje neophodne

U PRIPREMI

konstantnosti objekata. Kada se objektima oduzme promenljivost stanja, a identitet se identifikuje sa jednakošću vrednosti svih atributa, onda može da se izgradi jedan zaokružen sistem koji može da se koristi i za programiranje i za objektne baze podataka, ali takav sistem se značajno razlikuje od ustaljenih normi u svetu OOP. Nasuprot tome, takav pristup je uobičajen i široko se primenjuje u svetu funkcionalnog programiranja, gde umesto neformalnih koncepata na scenu stupa formalna i konceptualno zaokružena teorija tipova.

U brojnim člancima može da se pronađe više informacija o problemima OO koncepata, kao i predlozi za njihovo rešavanje. Zainteresovanim čitaocima preporučujemo radove Dejta i Darvina, u kojima se razmatraju neki od problema OOP, na primer rad „*Treći manifest*“ [*Date 1995*] ili knjigu „*Baze podataka, tipovi i relacioni model*“ [*Date 2006*]. Tu može da se vidi da možemo da formalno razmišljamo o OO programima i da definišemo formalnu semantiku OOPJ, u slučaju kada se formalni koncepti OOP uvedu kao nadgradnja funkcionalnog programiranja i strogo formalno definisanog sistema tipova.

3.5 Umesto zaključka

Praktično sve razvojne metodologije koje su danas u upotrebi su ili potpuno objektno-orijentisane ili bar počivaju na objektno-orijentisanim konceptima. Pokazalo se da je objektno-orijentisani pristup primenjiv u svim fazama razvoja softvera, od početnog razmatranja projekta, pa do njegove finalne implementacije i održavanja.

I pored određenih poznatih nedostataka, kroz praktične primene se ispostavilo da je objektno-orijentisani model programiranja i modeliranja softvera najbolji i najpouzdaniji poznati pristup razvoju softvera. Zbog toga se citat Edshera Daikstre, naveden na početku ovog poglavlja, obično navodi i tumači kao šala, iako je jedan od najvećih teoretičara računarskih nauka njime izrazio zapravo veoma ozbiljnu i objektivnu kritiku nepotpune i nedovoljno formalne teorijske zasnovanosti koncepata OOP.

Poslednjih godina u savremenom razvoju softvera sve više pronalaze svoje mesto koncepti i tehnike funkcionalnog programiranja. Sve više objektno-orijentisanih programskih jezika usvaja neke karakteristike koje su tradicionalno bile rezervisane za funkcionalne programske jezike. Čitaocima preporučujemo odličnu knjigu Ivana Čukića o funkcionalnom programiranju na programskom jeziku C++ [*Čukić 2018*], s tim da bi bilo dobro da pre nje najpre savladaju osnovne tehnike parametarskog polimorfizma (11 - Polimorfizam).

U PRIPREMI