Miskoncepcije u uvodnoj nastavi programiranja

Žana Žanko
zana.zanko@skole.hr

Sažetak—**Koliko su učenje i poučavanje složeni, ali i bitni procesi u razvoju nekog društva, pokazuje i činjenica da se o njihovoj važnosti govori od davnina. Vještina poučavanja primjenjuje se u prevladavanju neznanja i zabluda do kojih dolazi kad mislimo da znamo ono što ne znamo. Neupitan je kontinuitet problema neznanja, ali i danas tako važnog problema u prihvaćanju i usvajanju znanstvenih koncepata – miskoncepcija. Miskoncepcije ili alternativne koncepcije obuhvaćaju sve učeničke predodžbe ili konstrukte koji nisu u skladu sa znanstvenim spoznajama. Jednom stvorenu i prihvaćenu miskoncepciju može biti teško zamijeniti konceptom jer učenici teže jednostavnijim rješenjima i pri konstrukciji svojih teorija ulažu znatan mentalni napor te stoga ne odustaju lako od njih. S druge strane, njihovi pogrešni odgovori mogu biti najbolji alat u poučavanju i kreiranju uspješnog učenja i usvajanju znanja koja odgovaraju znanstvenim spoznajama. Zato je izuzetno važno da nastavnici budu svjesni postojanja učeničkih miskoncepcija kako bi ih uključili u planiranje svog poučavanja i time ga učinili uspješnijim. To se posebno odnosi na ona područja koja su sama po sebi teška i kompleksna, a programiranje, sasvim sigurno, spada među njih.**

Ključne riječi—**učenje, poučavanje, koncept, predkoncepcije, miskoncepcije, alternativne koncepcije, programiranje, računalna znanost**

I. UVOD

Računala su dio svakodnevnice svih nas. Gotovo da nema aktivnosti ili zanimanja u današnjem svijetu u kojima se ne traži barem neka razina računalnih znanja. A ako su važna računala, važno je i programiranje [1]. Prema riječima dobitnika ACM-ove Turingove nagrade A. Perlisa, svi bi trebali učiti programirati kao neizbježan dio svog obrazovanja [2] s čime se slažu mnogi [3], [4], [5]. Programiranje predstavlja skup vještina pomoću kojih ljudi čine računala korisnima [1]. To je računalom potpomognuto rješavanje problema, otklanjanje pogrešaka, razvijanje logičkog, analitičkog i algoritamskog načina razmišljanja što ujedno podrazumijeva i razvoj strategija za rješavanje problema. Može se reći da programiranje mijenja način razmišljanja [6]. Učenje programiranja podrazumijeva ne samo učenje novih pojmova, već i vježbanje vještine primjene naučenog u novim situacijama [7]. Zbog svega navedenog ono je nova pismenost [8].

Rad je, u sklopu Istraživačkog seminara I, predan na ocjenu 10. lipnja 2016. godine Povjerenstvu u sastavu: prof.dr.sc. xxx, predsjednik, prof.dr.sc. xxx, član, izv.prof.dr.sc. Ivica Boljat, mentor.

Žana Žanko studentica je poslijediplomskog sveučilišnog studija Istraživanje u edukaciji u području prirodnih i tehničkih znanosti, usmjerenje Informatika, na Prirodoslovno-matematičkom fakultetu u Splitu.

Programiranje je temeljni dio kurikuluma računalne znanosti [9]. Nema sumnje da je programiranje izuzetno zahtjevno [10] čak i na jednostavnoj razini [11] jer je teško za učenje, razumijevanje i svladavanje, pogotovo za početnike [9], [12], [13] bez obzira na njihovu dob [14]. Zato ne iznenađuje podatak da mnogi učenici imaju poteškoće već u ranim fazama učenja i da je stopa njihovog neuspjeha i odustajanja visoka [9]. Pritom je posebno interesantno uočiti da su neki od koncepata učenicima problematični još od ranih 80-ih godina prošlog stoljeća pa sve do današnjih dana. Stoga je programiranje područje koje je izazov i za učenje i za poučavanje [15].

Trenutno poučavanje uvodnog programiranja daleko je od uspješnog [16]. Posljednjih nekoliko desetljeća provode se brojna istraživanja koja doprinose sve boljem razumijevanju učeničkih poteškoća u uspješnom svladavanju vještine programiranja. Među bitna otkrića tih istraživanja spada i činjenica da učenici imaju različita predznanja i pogrešna shvaćanja tijekom usvajanja potrebnih koncepata koja ometaju njihovo učenje. Veoma je važno da se nastavnici upoznaju s tim pogrešnim shvaćanjima, tj. miskoncepcijama učenika jer njihovo pravovremeno prepoznavanje, ispravljanje i uklanjanje vodi smanjenju njihovog utjecaja na proces sjecanja novih znanja te poboljšanju učinkovitosti učenja, poučavanja i konceptualnog razumijevanja.

Ovaj rad ide upravo u tom smjeru i daje pregled najčešćih miskoncepcija u uvodnoj nastavi programiranja, navodi moguće uzroke njihovog nastanka i tehnike kojima se mogu detektirati te potom zamijeniti konceptima. On ujedno pokazuje i da su neke miskoncepcije vrlo stabilne i otporne na protok vremena jer se kontinuirano ponavljaju.

II. KONCEPT, PREDKONCEPCIJE, MISKONCEPCIJE I/ILI ALTERNATIVNE KONCEPCIJE

Koncept (eng. *concept*) je utemeljena znanstvena spoznaja. Učenici, međutim, u razrede ne ulaze kao "prazne ploče". Oni često o nekom pojmu imaju određeno predznanje koje može, ali i ne mora, biti u skladu sa znanstvenim spoznajama. Takvo predznanje učenika naziva se *predkoncepcija* (eng. *preconception*) i rezultat je osobnog iskustva ili prethodnog učenja, a ne slučajno objašnjenje nekog pojma. Tijekom školovanja ne obraća se uvijek dovoljno pozornosti na predkoncepcije učenika koje, ako su ispravne, omogućuju nadogradnju znanja, ali, ukoliko su pogrešne, dovode do razvoja *miskoncepcija* (eng. *misconceptions*), odnosno, učeničkih koncepata koji nisu u skladu sa znanstvenim spoznajama. Miskoncepcije se često javljaju kod učenika jer

su to, iako netočna, jednostavna objašnjenja koje učenici lakše i brže usvajaju. Prema Fisher [17], miskoncepcije općenito imaju neke zajedničke osobine: 1) u suprotnosti su sa znanstvenim konceptima 2) postoji tendencija da se iste miskoncepcije često pojavljuju kod većeg broja ljudi 3) miskoncepcije su vrlo otporne na promjenu, a posebno kada se u poučavanju koristi tradicionalna nastava 4) miskoncepcije ponekad uključuju čitave alternativne sustave koji su čvrsto logički povezani i koje učenici često koriste 5) miskoncepcije mogu nastati kao rezultat automatske obrade jezične strukture bez korekcije smisla, mogu biti posljedicom određenih iskustava koja su obično zajednička većem broju pojedinaca ili nastaju na temelju nastave u školi pogrešnim tumačenjem i razumijevanjem nastavnih sadržaja.

Naziv *alternativne koncepcije* (eng. *alternative conceptions*) izraz je koji označava isto što i miskoncepcije, ali se koristi kako bi se naglasio legitimitet koncepcija učenika, osobito u ranim fazama učenja [18].

Miskoncepcije ili alternativne koncepcije nisu neuobičajena pojava. U stvari, one su normalan dio procesa učenja. Iza svake učenikove pogreške ili nerazumijevanja stoji prilika za učenje. Zato pogreške ne treba nužno uzimati negativno jer učenicima pružaju priliku da isprave postojeće znanje i ažuriraju svoje mentalne prezentacije određene teme ili koncepta. Iz pedagoške perspektive, budući su u mnogim slučajevima miskoncepcije konzistentne, sustavne i utemeljene na nekim modifikacijama ispravnog znanja te su stabilne u pokušajima da ih se promjeni, nastavnici bi trebali tražiti miskoncepcije i pomoći učenicima da ih isprave tako da, prije svega, riješe njihove uzroke i na taj način otvore put usvajanju koncepata [18].

III. MOGUĆI UZROCI MISKONCEPCIJA

Ne postoji općeprihvaćeno slaganje o tome zašto je programiranje učenicima teško i što uzrokuje njihove zajedničke greške i miskoncepcije. Problem je univerzalan i neovisan o državi i njenom obrazovnom sustavu [19] te o uzrastu učenika ili odabranom programskom jeziku [20]. Razlozi su različiti i to je pitanje vrlo kompleksno [9].

A. *Sposobnosti učenika i metode poučavanja*

Programiranje zahtijeva ne samo jednu nego cijeli niz vještina koje tvore hijerarhiju [21] te nadilaze poznavanje same sintakse i semantike nekog programskog jezika [22]. Stoga ih ni u kom slučaju ne smijemo promatrati izolirano svaku za sebe što je čest slučaj [23]. Brojni autori smatraju da je za programera najvažnije razviti strategije rješavanja problema i sposobnosti logičkog razmišljanja [9], [18], [22], [23] pa njihov nedostatak može biti uzrok poteškoća [19], [24]. Poteškoće mogu uzrokovati i apstrakcije [13], [19] koje su temeljni koncept programiranja [25]. Prema Piaget-ovoj teoriji, postoje četiri faze kognitivnog razvoja čovjeka: senzomotorna (0.-2. g.), predoperacijska (2.-7. g.), konkretnih operacija (7.-11. g.) i formalnih operacija (>12 g.) [26]. U svakoj od tih faza razmišlja se na sličan način i slično se pristupa postavljenom problemu. Neka istraživanja, međutim, pokazuju da samo trećina svih adolescenata dolazi do posljednje faze, dok ni kod odraslih taj postotak nije bitno veći [27]. Shodno tome, problemi za mnoge učenike počinju već u ranoj fazi učenja kada trebaju razumjeti i primijeniti apstraktne programerske koncepte za stvaranje algoritama koji rješavaju konkretne probleme [1], [9], [13]. Kako mnogi učenici nemaju dovoljno razvijenu sposobnost apstraktnog razmišljanja, učenje programiranja koje je potpuno apstraktno čini im se teškim i mnogi među njima odustaju [9]. Nadalje, prema Bloom-ovoj taksonomiji [28], učenici koji žele uspješno savladati vještinu programiranja trebaju rano doseći 3., 4. i 5. razinu znanja (primjena, analiziranje, vrednovanje), no mnogi se ugodno osjećaju samo djelujući na 1. razini (pamćenje) i ponekad na 2. razini (razumijevanje) [23]. Najveći broj miskoncepcija općenito, uključujući i područje programiranja, nastaje upravo zbog usvajanja znanja bez razumijevanja što onemogućava razvoj valjanih koncepata. Problem je još izraženiji ukoliko se u poučavanju koriste tradicionalni oblici nastave [9], [29] jer učenik tada nije dovoljno zainteresiran da promjeni svoje mišljenje o nekom pojmu dok mu aktivan oblik rada omogućava samostalno donošenje ispravnih zaključaka i sustavno organiziranje znanja. Prema Saeed [10], glavni razlozi za koje se može pretpostaviti da su uzrok poteškoća su: sposobnosti učenika, metode poučavanja i složenost programiranja. Način poučavanja kao izvor problema proučavale su i Linn i Dalbey [30] koje smatraju da ne treba žuriti u radu i prelaziti na složenije situacije sve dok se temeljito ne usvoje osnovni koncepti. Njihovo istraživanje na 500 studenata pokazalo je da se zadržavanje na jednostavnim algoritmima može višestruko isplatiti na dulje staze. Winslow [22] se također zalaže za postupan prijelaz od jednostavnog prema kompleksnijem. Dodatna prepreka mogu biti velike i heterogene skupine učenika koje onemogućuju kreiranje poučavanja koje bi bilo uspješno za svakog od njih [13]. Dio problema može ležati i u matematičkoj pozadini učenika [31], [32], [33] te njenoj kombinaciji s nekim drugim faktorima [34]. Sporne mogu biti i vještine prostorne vizualizacije koje učenici posjeduju [35]. Sposobnosti predočavanja i shvaćanja procesa koji se događaju u pozadini izvršavanja programa temeljne su za uspješno učenje programiranja te stoga objašnjenje zašto mnogi učenici ne nauče programirati i zašto imaju poteškoća u razumijevanju treba tražiti i u njihovom nepoznavanju načina rada i mogućnosti samog računala [36], [37], [38].

B. *Sintaksa programskih jezika*

Razlog neshvaćanju i miskoncepcijama kod programera početnika može ležati i u sintaksi programskih jezika [14]. Mayer [38] tvrdi da osnovnu poteškoću po tom pitanju čini nemogućnost nošenja učenika s gramatičkim pogreškama koje se javljaju u sintaksi korištenih jezika, a koje im nisu prirodne. Green [39] dodaje da programski jezik i ne može biti prirodan i neusiljen pa samim tim ni jednostavan za upotrebu te razotkriva neistinitost tvrdnji da funkcionira na način na koji razmišljaju ljudi. S posljednjim se slaže i Pea [20]. Bonar i Soloway [11] tvrde da programeri početnici imaju prethodna znanja i iskustva koja im razvijaju snažnu intuiciju u korištenju programskih jezika. Programski jezici, međutim,

nisu dizajnirani da odgovore na tu intuiciju jer na semantičkoj i pragmatičnoj razini postoje neusklađenosti između načina korištenja prirodnog i programskog jezika. Zato se mnoge pogreške i miskoncepcije mogu izravno povezati s neprikladnom primjenom pravila i strategija prirodnih jezika u programiranju. O greškama i zabludama kao posljedici prijelaza s prirodnog na programerski jezik govore u svojim istraživanjima i [40], [41], [42]. Kada je sintaksa u pitanju, do dodatne zbunjenosti može dovesti korištenje engleskih riječi kao što su *print*, *read* ili *input* u programskim jezicima i to iz dva razloga: 1) sve te riječi imaju vrlo specijalizirano značenje prilikom pisanja programa koje je daleko od njihove svakodnevne konotacije i 2) mnoge od tih riječi imaju više značenja u engleskom jeziku [36]. Nadalje, u razgovoru se riječ *then* koristi u smislu onoga što će se sljedeće napraviti kao u rečenici "Ušao sam u trgovinu i tada sam kupio papir", dakle, u engleskom jeziku ona sugerira vremenski slijed što se ne podudara s njenim značenjem u programiranju. U tom smislu, zabilježena je čak i konstrukcija *repeat…until…then* za osmišljavanje izjave koja će osigurati izvršavanje niza radnji nakon što petlja završi sa svojim izvođenjem. Dodatan primjer je i Boolean operator *and* koji se u svakodnevnom govoru ponekad koristi za spajanje radnji u niz, npr. "Operi ruke i postavi stol", a koji ne znači to isto prilikom pisanja programa. Riječ *repeat* također može voditi u miskoncepciju i to one početnike koji očekuju da već postoji nešto što će se onda ponavljati. Primjer jezične zablude je i pogreška nazvana "*While* demon" u kojoj početnici pretpostavljaju da akcije u tijelu petlje kontinuirano prate ne bi li izlazni uvjet postao ispunjen [11]. Ta interpretacija konzistentna je s načinom korištenja riječi *while* u engleskom jeziku koji podrazumijeva kontinuirano aktivno ispitivanje kao u rečenici "Dok cesta ima dvije trake, nastavi na sjever". *While* petlja, međutim, obavlja samo jedno ispitivanje po iteraciji, a poistovjećivanje riječi *while* u programiranju s riječi *while* u stvarnom životu može dovesti do očekivanja da se petlja završi u trenutku kada je ispitivanje izraza zadovoljeno. Prema onom što su utvrdili Kurland i Pea [42], problem može stvarati i riječ *stop* koja se koristi prilikom pisanja rekurzivnih programa. Naime, zbog njenog uobičajenog značenja u svakodnevnom govoru koji podrazumijeva zaustavljanje, učenici često misle da u toj točki program prestaje s izvršavanjem, a u stvari se samo kontrola njegovog tijeka vraća unatrag. Nepoznavanje same sintakse kao dio neuspjeha spominje se u [19]. O tome su detaljnije pisali i Brown i Altadmri [43] koji, između ostalog, navode sintaksne greške nastale kao posljedica nerazlikovanja = i ==, & i &&, neslaganja ili ispuštanja ili korištenja pogrešnih zagrada, pogrešne upotrebe ključnih riječi kao imena varijabli ili metoda, pogrešnog korištenja znaka ; u *if*, *for* i *while* konstruktima pa čak i pogrešna upotreba operatora <= i >=.

C. Predkoncepcije

Uzrok neispravnom shvaćanju može ležati i u predkoncepcijama jer se znanje koje učenici konstruiraju zasniva na njihovom prethodnom iskustvu. To je u skladu s konstruktivističkom teorijom učenja prema kojoj učenici radije aktivno konstruiraju svoje znanje na temelju osobnog iskustva umjesto da ga samo pasivno primaju od strane nastavnika. Staro znanje utječe na novo znanje. Međutim, ono što je ispravno naučeno u prethodnom kontekstu može postati pogrešno u novom kontekstu pa predkoncepcije mogu voditi do miskoncepcija [44]. U tom slučaju, staro znanje ometa novo znanje. Zato je važno znati koje predkoncepcije učenici imaju i suočiti se s njima u cilju unapređenja procesa učenja i poučavanja [45] i to za svakog učenika posebno [46]. Dio predkoncepcija koje učenici imaju, a koje mogu ometati njihovo učenje programiranja te voditi do miskoncepcija, odnose se i na njihova matematička znanja. Više o njima istraživao je White u svom radu [44]. To se posebno odnosi na simbole + i = koji mogu imati različita značenja u matematici i računalnom programiranju. Na primjer, x=x+1 u matematici bi bila pogrešna izjava dok je u programiranju to ispravna izjava kojom povećavamo vrijednost varijable x na način da se njena trenutna vrijednost uveća za 1 i dobiveni rezultat pohrani natrag u x. Dakle, znak = u programiranju označava pridruživanje umjesto jednakosti. Osim toga, u programiranju znak = može također označavati i uspoređivanje kao u IF izjavama što je dodatan izvor zbunjenosti. Nadalje, u matematici znak + označava računsku operaciju zbrajanja, a u programiranju može značiti to isto, ali A+B može značiti i ulančavanje, tj. spajanje dva znaka A i B čiji je rezultat AB. Konačno, postoji prirodan način prebrojavanja jer nas se od prvog dana školovanja uči da ga započnemo od broja 1. U programiranju ga, međutim, često trebamo započeti nulom pa se događa da osoba ili propusti započeti nulom ili pogrešno izračuna broj članova niza. U [47], Green dodatno raspravlja složenost činjenice da u nekim programskim jezicima, kao što su C, C++ i Java, indeksi niza započinju nulom umjesto jedinicom kako je uobičajeno u matematici što učenicima predstavlja problem za zapamtiti i primijeniti u danim situacijama. Dakle, računalna matematika razlikuje se od algebarske matematike i to treba uzeti u obzir kod poučavanja programiranja. Osim matematičkih, i druga prethodna znanja učenika mogu voditi do miskoncepcija. Dio predkoncepcija povezanih uz poznavanje engleskog jezika te korištenje strategija svojstvenih prirodnom jeziku u programiranju opisana su ranije u ovom poglavlju. Problem može ležati i u prethodnim programerskim iskustvima učenika koja mogu negativno utjecati na usvajanje i primjenu novih koncepata [48]. Metodologija poučavanja na ranijim razinama obrazovanja, čak i više od samog sadržaja, može također značajno utjecati [49].

D. Analogije

Uzrok miskoncepcija mogu biti i analogije koje se koriste. U radu [36] Du Boulay navodi da je početnicima, u svrhu što boljeg usvajanja koncepata, potrebno prezentirati različite analogije, ali pri tome treba biti jako oprezan jer vrlo često analogija uvedena u nekom trenutku ne mora poslije biti odgovarajuća i zbog toga može dovesti do dodatne zbunjenosti kod učenika. Primjer takve analogije je često vjerovanje učenika da je varijabla kao kutija ili ladica s naljepnicom na sebi i da stoga može sadržavati više od jedne vrijednosti u isto vrijeme budući da i kutija ili ladica mogu sadržavati više od

jedne stvari. Sljedeća sporna analogija koju autor spominje je varijabla kao list papira na kojem su zapisane vrijednosti što može dovesti do pogrešnog razumijevanja i neshvaćanja da se u računalu svaki put piše preko postojećeg i da se pritom ne pamte sve prethodno pridružene vrijednosti dok na papiru istovremeno može biti zapisano više toga. Također ističe i slučaj da, kada se varijabla koristi za računanje ukupnog zbroja, činjenica da učenici često zaborave na početnu inicijalizaciju zbroja na nulu dolazi od pogrešne analogije koja varijablu predstavlja kao kutiju jer ako ništa nije stavljeno u kutiju, ona je prazna, a to je onda nešto poput nule. Isto tako, treba biti vrlo oprezan i s analogijom između programiranja kao niza instrukcija i niza radnji koje obavlja čovjek (npr. kod kuhanja čaja) jer se mogu izvesti netočni zaključci koji idu u dva smjera zbog toga jer je: 1) jezik za opis radnji koje obavlja ljudsko biće neusiljen za razliku od programskih jezika i 2) ne treba biti toliko detaljan u opisu budući da se mnoge stvari u svakodnevnim situacijama podrazumijevaju za razliku od situacija programiranja. Vrlo je važno znati da računalo nema normalne ljudske sposobnosti zaključivanja onoga što se htjelo iz onoga što se reklo te stoga treba misliti i na najmanji detalj i držati se vrlo strogih pravila korištenja jezika. Pea [20] također ističe problematičnost analogije između komuniciranja osobe s računalom prilikom pisanja programa i međusobnog komuniciranja ljudi koja bi mogla početnike navesti na pogrešan put budući da, za razliku od ljudi koji razumiju i neizgovoreno, programski jezici nisu inteligentni tumači razgovora. Poteškoće biranja prikladnih analogija spominju se i u [22]. Halasz i Moran [50] smatraju da analogije imaju korisnu ulogu u poučavanju početnika o računalnom sustavu, ali mogu biti vrlo opasne ako se koriste za objašnjavanja detalja. Taj je dio puno bolje odraditi pomoću apstraktnih konceptualnih modela jer uvijek postoje neki aspekti analognih modela koji su irelevantni za sustav koji se modelira pa je za njegovo učinkovito korištenje početnik suočen sa zbunjujućim zadatkom odvajanja relevantnih zaključaka od brojnih mogućih, nevažnih ili netočnih zaključaka koji su predloženi po analogiji.

IV. KAKO OTKRITI MISKONCEPCIJE I ZAMIJENITI IH KONCEPTIMA?

Miskoncepcije se javljaju u poučavanju gotovo svih područja znanosti. Mnogi nastavnici ostanu zatečeni kada shvate da ni uz njihov najveći trud učenici ne uspijevaju usvojiti osnovne koncepte, ponekad čak ni najbolji među njima [51]. Pogrešni odgovori učenika najbolji su način otkrivanja miskoncepcija. Međutim, vrlo je važno znati da se one mogu skrivati i iza njihovih točnih odgovora. Zato nastavnici trebaju biti svjesni učenikovog načina razmišljanja i njegovih mentalnih procesa, steći vještine za otkrivanje alternativnih koncepcija, razumjeti i alternativne koncepcije i njihove uzroke, a zatim koristiti različite pedagoške alate za njihovo uklanjanje i pomoć učenicima u poboljšanju razumijevanja [18].

Nameće se pitanje kako to ostvariti. Da bi se uspješno implementirao konceptualni pristup u nastavni proces koji će učenicima pomoći u boljem razumijevanju i ispravnom usvajanju novih sadržaja, potrebno je: 1) prepoznati miskoncepcije koje već postoje 2) dodatno ih istražiti 3) tražiti od učenika da objasne svoja pogrešna shvaćanja 4) raspraviti o kontradiktornim konceptima učenika uz pitanja, primjere i protuprimjere 5) poticati daljnju raspravu i tražiti od učenika da primijene svoje koncepte u konkretnim primjerima 6) zamijeniti miskoncepcije novim konceptima postavljajući učenicima pitanja i uvodeći pritom neku hipotetsku situaciju 7) ponovno analizirati usvajanje novih koncepata postavljajući konceptualna pitanja. Dakle, miskoncepcije je potrebno prepoznati, istražiti, tražiti pojašnjenja za njih, potaknuti raspravu o njima te ih na kraju zamijeniti novim, valjanim konceptima uz ponovnu analizu i provjeru.

Prema Strike i Posner [52], da bi se miskoncepcije uspješno zamijenile konceptima prethodno moraju biti ispunjeni sljedeći uvjeti: 1) mora postojati *nezadovoljstvo* postojećim konceptima jer učenici ne žele mijenjati već izgrađene koncepte ukoliko se ne uvjere da su postali nefunkcionalni 2) novi koncept mora biti *razumljiv* i učenici moraju biti u stanju sagledati njegov smisao 3) novi koncept mora biti *uvjerljiv* i omogućiti objašnjenje nekog pojma koje postojeći koncept nije mogao pružiti 4) novi koncept mora biti *plodonosniji* od postojećeg i otvarati nove putove razmišljanja.

Hazzan i suradnici u [18] predlažu nekoliko strategija koje omogućuju ne samo da se utvrde miskoncepcije nego i da se otkriju izvori pogrešnih odgovora te razumiju mentalni modeli koji leže u njihovoj pozadini. Jedna od njih odnosi se interakciju između učenika i nastavnika u kojoj se nastavnici trebaju oduprijeti svom očekivanju da učenik odmah odgovori na postavljeno pitanje i da oni potom odmah ispravljaju eventualno uočenu pogrešku u načinu njegovog razmišljanja. Druga strategija odnosi se na osmišljavanje dijagnostičkih vježbi koje trebaju pomoći u otkrivanju pogrešaka i iz perspektive učenika i iz perspektive nastavnika. Konkretno to znači da učenik mora prepoznati greške kako bi ih mogao ispraviti te modificirati svoje kognitivne modele, a nastavnik treba razotkriti učenikove greške u cilju provođenja pedagoške intervencije. Treća strategija greške i zablude navodi kao priliku za učenje i poboljšanje razumijevanja.

I drugi autori također se slažu da su ispravni mentalni modeli osnova za učenje i ispravno razumijevanje programerskih koncepata [53], [54]. L. Ma u svojoj disertaciji [55] dokazuje da učenici koji imaju održive mentalne modele programerske zadatke izvode značajno bolje od onih koji ih nemaju, a za njihovo razvijanje predlažu model učenja baziran na konstruktivizmu koji integrira strategije kognitivnog konflikta u kombinaciji s tehnikama programskih vizualizacija. Vizualizaciju, kao način suočavanja učenika s njihovim mis/konceptima te kao dio rješenja, vide i drugi istraživači [56], [57]. Motivacija učenika također može biti značajan faktor [58], [59]. U svom radu [20] Pea se zalaže za više obrazovnih aktivnosti za učenike i više dijagnostičkih aktivnosti za nastavnike kako bi pogreške postale očigledne. Aktivno učenje zagovaraju i Götschi i suradnici [60]. Također su važni i materijali za poučavanje koji trebaju uvažavati sve poznate miskoncepcije kako bi se na taj način pomoglo nastavnicima da ih identificiraju u svojim razredima i potom

zamijene konceptima [61]. Jednako su važni i materijali za učenje jer informacije o miskoncepcijama i načinima njihovog prevladavanja svakako mogu biti od pomoći i učenicima [62].

Miskoncepcije se ne mogu lako detektirati konvencionalnim testovima i drugim tradicionalnim metodama procjene. Najčešći načini koje nastavnici primjenjuju u tu svrhu su istraživanja obrazovanja u području računalne znanosti [18], nedoumice i pitanja koja učenici postavljaju tijekom nastavnog procesa [63], učenikovo izgovaranje naglas prilikom objašnjavanja u razrednoj diskusiji njegovog viđenja tijeka i rezultata izvođenja programa [42], pitanja višestrukog izbora na zadanu temu [10], [64], pitanja otvorenog tipa [55], zadaci uparivanja, tj spajanja odgovarajućih pojmova i njihovih značenja s jednakim ili različitim brojem ponuđenih opcija na obje strane [65], praćenje kôda od strane učenika s ciljem njegovog razumijevanja [20], [66] ili predviđanja izlaza programa [18], pisanje vlastitih jednostavnih programa [20], [67], [68], prepravljanje već napisanih programa u svrhu obavljanja novih zadataka [64], završavanje započetih programa, nadopunjavanje programa s jednom ili više linija kôda koje nedostaju u cilju obavljanja postavljenog zadatka [69], [65] razmještanje ponuđenih linija programskog kôda u pravilan redoslijed u cilju obavljanja postavljenog zadatka [65], traženje pogrešaka u ponuđenim programima [20], obavljanje kratkih zadataka u jednoj liniji kôda [65], odgovori učenika na pitanja u kojima trebaju predvidjeti koji će od programa funkcionirati [70], ispiti znanja (koji ne trebaju biti samo sredstvo ocjenjivanja nego i istraživački instrument za otkrivanje ne/znanja učenika) [65], učenikovo prostoručno crtanje, skiciranje i opisivanje njegovog shvaćanja određene izjave tijekom izvršavanja programa ili određenog koncepta [71], interaktivne vizualne programske simulacije [56], popisi s prethodnim korisnim iskustvima kolega [25], [72] te opsežni intervjui [73] odnosno klinički razgovori s učenicima u kojima nude objašnjenja informacija vlastitim riječima [18], [38]. Navedene tehnike neizostavno treba implementirati i pojedinačno i kombinirano u nastavni proces jer je prisutnost te postojanost miskoncepcija izravno povezana s učestalošću suočavanja i učenika i nastavnika s njima.

V. MISKONCEPCIJE U UVODNOJ NASTAVI PROGRAMIRANJA

Osim osvijestiti činjenicu o postojanju miskoncepcija te važnosti njihovog prepoznavanja i uklanjanja kako bi se usvojili koncepti, jedan od glavnih ciljeva ovog rada je i dati pregled istraženosti područja miskoncepcija koje se javljaju u području programiranja. Velik broj niže navedenih radova pokazuje da se iste miskoncepcije pojavljuju već od prvih istraživanja na tu temu pa sve do današnjih dana što najbolje svjedoči o njihovoj stabilnosti.

A. *Miskoncepcije o osnovnim konceptima programiranja*

Među najranijim i najcitiranijim radovima je i onaj Baymana i Mayera [74] iz 1983. koji kategorizira kako se 30 studenata snašlo u samostalnom proučavanju BASIC-a. Istraživanje je pokazalo da samo 3% početnika razvija ispravno shvaćanje BASIC izjave INPUT A, 10% izjave 30 READ A, 27% izjava IF A<B GOTO 99, LET A=B+1, 20 DATA 80, 90, 99 i 60 GOTO 30, 33% izjave PRINT C, 43% izjave LET D=0, a 80% studenata razvija točno shvaćanje izjave PRINT "C". Tablica 1 prikazuje najčešće miskoncepcije povezane sa svakom od navedenih izjava.

Tablica 1: Najčešće miskoncepcije navedenih naredbi

Izjave	Najčešće miskoncepcije navedenih izjava
INPUT A	• upisati slovo A u memoriju • pričekati na određeni broj ili slovo • ispisati slovo A na ekranu
30 READ A	• ispisati vrijednost od A na ekranu • upisati slovo A u memoriju • čekati na unos broja s tipkovnice
IF A<B GOTO 99	• pomaknuti se na liniju 99 bez ispitivanja uvjeta • ispisati broj 99 ili liniju 99 na ekranu • upisati A ili B ili A<B ili broj u memoriju
LET A=B+1	• upisati jednadžbu u memoriju • upisati B+1 u memorijsku lokaciju A • ispisati A=B+1 ili A ili vrijednost od A na ekranu
20 DATA 80, 90, 99	• staviti brojeve u memoriju • ispisati brojeve na ekranu • staviti brojeve u memorijsku lokaciju A
60 GOTO 30	• pronaći broj 30 • pomaknuti se na liniju 30 ako A nije jednak određenoj vrijednosti • ispisati broj 30 na ekranu
PRINT C	• ispisati slovo C na ekranu • upisati slovo C u memoriju • ispisati na ekranu ili "greška" ili ništa
LET D=0	• upisati jednadžbu u memoriju • upisati D ili 0 u memoriju • ispisati jednadžbu na ekranu
PRINT "C"	• upisati slovo C u memoriju • ispisati vrijednost od C na ekranu • pronaći broj u memorijskoj lokaciji C

Putnam i suradnici [75] proveli su 1984. godine istraživanje s 96 srednjoškolskih učenika o konstruktima u BASIC programskom jeziku na način da su učenici pratili jednostavne programe i predviđali njihove izlaze i koje je pokazalo da su najčešće greške povezane s:

1) *izjavama pridruživanja*, npr. u izjavi LET A = B učenici radije dodjeljuju vrijednost od A u B nego vrijednost od B u A, to je tzv. okretanje izjave pridruživanja, ili smatraju da je izjava LET C=C+1 nemoguća zbog njene matematičke interpretacije;

2) *pogrešnim tumačenjem oznake navodnika u PRINT izjavama*, npr. u izjavi PRINT "Q: "; Q učenici ili jednostavno ignoriraju tekst unutar navodnika ispisujući jednom vrijednost od Q ili dva puta zaredom ispisuju istu vrijednost od Q ili u slučaju da je varijabla promijenila vrijednost tijekom izvršavanja programa ispisuju dvije različite vrijednosti od Q što ukazuje dodatno i na dublju miskoncepciju po kojoj bi varijabla mogla "zapamtiti" svoju prvotnu vrijednost;

3) *poteškoćama s READ izjavama* pogotovo ako se koriste smislena imena varijabli jer smatraju da program odabire vrijednost iz DATA izjave na temelju značenja imena varijable umjesto čitanja sljedeće vrijednosti u redu, npr. ako

su dane naredba 40 READ SMALLEST i naredba 200 DATA 99, 2, -3, -100, 6 učenici smatraju da će u SMALLEST biti učitan -100, to je tzv. semantičko čitanje;

4) *varijablama* jer često smatraju da varijabli može biti pridruženo više od jedne vrijednosti istovremeno ili im varijabla cijelo vrijeme pamti samo svoju inicijalnu vrijednost;

5) *petljama*, npr. ne shvaćaju da se u FOR/NEXT petlji vrijednost kontrolne varijable povećava svakom iteracijom nego smatraju da je treba mijenjati unutar tijela petlje;

6) *korištenjem IF izjave*, npr. smatraju da izvršenje programa prestaje kada uvjet u IF izjavi nije ispunjen ili da se kontrola u tom slučaju prebacuje na početak programa ili da se sve izjave u programu moraju izvršiti barem jednom pa čak i izjave koje bi trebale biti preskočene zbog grananja u programu.

Ovo je istraživanje, dakle, pokazalo prisutnost miskoncepcija koje proizlaze iz primjene znanja i zaključivanja iz neformalnih područja u programiranje, a koje sprječavaju učenike u razvijanju kognitivnih sposobnosti više razine koje bi nastava programiranja trebala poticati.

Perkins i Simmons [37], slično Putnamu i suradnicima, naglašavaju također kako početnici često imaju miskoncepcije vezane uz imena varijabli za koja očekuju da ih računalo razumije iako znaju da je izbor tih imena, u pravilu, njihov. Isto je, u svom radu [36], zaključio i Du Boulay. On ujedno predlaže povremenu upotrebu besmislenih imena za varijable kako bi se učenici sami uvjerili da računalo ne razumije engleski jezik na način na koji oni to očekuju.

Vezano uz miskoncepcije, Du Boulay je u istom istraživanju [36] uočio i sljedeće:

1) LET A=2 i LET 2=A se poistovjećuju.
2) TEMP:=A; A:=B; B:=TEMP ispravan je redoslijed naredbi kod zamjene vrijednosti dviju varijabli, međutim, mnogi početnici ova pridruživanja obave u pogrešnom redoslijedu što se vjerojatno događa zbog njihovog nalikovanja na tri jednadžbe koje su istovremene izjave o svojstvima A, B i TEMP, a ne način za postizanje određenog unutarnjeg stanja.
3) LET A=A+1 izaziva zbunjenost upravo zbog nerazumijevanja asimetrije i sekvencijalne prirode izvršavanja čak i ovog jednog pridruživanja. A-ovi sa svake strane jednakosti ne znače isto, desni je vrijednost s memorijske lokacije simbolički nazvane varijablom A, a lijevi je ta ista adresa na koju će se spremiti rezultat. Poteškoća nastaje zbog nedovoljnog razumijevanja i pojma varijable i načina interpretiranja izjave.
4) LET A=B neki početnici vide kao povezivanje druge varijable B na prvu varijablu A na način da svaka buduća promjena u A rezultira promjenom i u B što je posljedica ključnog nerazumijevanje privremenog dosega varijabli i neshvaćanja da vrijednost ostaje u varijabli sve dok se ona eksplicitno ne promijeni ili se ne izbriše sadržaj memorije ili se računalo ne isključi.
5) SUM se, kod računanja ukupnog zbroja, zaboravlja inicijalizirati na 0.
6) LET A=7+4 izaziva nerazumijevanje jer početnici misle da A sadrži 7+4 kao neevaluirani izraz umjesto vrijednosti 11.
7) LET A=2; LET B=A u slučaju pridruživanja vrijednosti jedne varijable drugoj navodi početnike vrlo često na ideju da je u konačnici varijabla A prazna, a da varijabla B sadrži 2.

Godine 2011. Simon [66] je, 25 godina nakon Du Boulaya, a 30 i više godina nakon nekih drugih istraživača, također utvrdio postojanje istih neshvaćanja kroz činjenicu da mnogi studenti prve, druge pa čak i treće godine kolegija programiranja ne razumiju semantiku izjave pridruživanja ili redoslijed izvršavanja izjava ili pak oboje, a samim tim ni algoritam zamjene vrijednosti dviju varijabli koji u sebi uključuje te osnovne koncepte. Velik dio studenata, uz neshvaćanje izjave pridruživanja, vjeruje da se skupina izjava izvršava istodobno umjesto sekvencijalno, tj. redoslijedom kojim se pojavljuje, i to mogu biti jasne smjernice na što treba obratiti posebnu pozornost u prvim tjednima uvodnog programiranja. Dodatno, Robins [76] zaključuje da će nepoznavanje osnovnih koncepata onemogućiti studente u daljnjem uspješnom svladavanju svih novih koncepata koji su usko povezani s njima. Primjerice, Corney i suradnici [64] smatraju da studenti koji ne razumiju zamjenu vrijednosti dviju varijabli nisu pripremljeni za usvajanje iterativnog kôda koji potom slijedi.

Soloway i suradnici [77] opisuju kako više od četvrtine njihovih studenata na kraju semestra programiranja netočno koristi istu varijablu za više od jedne uloge. Primjer takve situacije je kada varijablu X koriste i za pohranu vrijednosti koja je učitana (READ (X)) i za držanje ukupnog zbroja (X:=X+X). Pritom pretpostavljaju da će računalo prepoznati da ista varijabla igra dvije različite uloge i da sukladno tome može koristiti različite vrijednosti na odgovarajući način. To je usko povezano s pogreškom egocentrizma koju niže opisuje Pea.

Pea [20] je pronađene konceptualne pogreške, neovisne o programskom jeziku i zajedničke svim uzrastima od osnovnoškolske do studentske razine, podijelio u tri odvojene skupine:

1) *Paralelizam*

Paralelizam je pogreška koja se javlja u mnogim kontekstima, a u njenoj suštini je pretpostavka da različite linije programskog kôda mogu biti aktivne u isto vrijeme, tj. paralelno.

Dva su tipa programa u kojima je pogrešno razumijevanje često.

Jedan je onaj u kojima se uvjet pojavljuje izvan petlje, npr. kada se prvo javlja uvjet IF SIZE=10, THEN PRINT "HELLO, a kasnije u programu slijedi petlja FOR SIZE=1 to 10, PRINT "SIZE / NEXT SIZE. Učenicima koji razumiju kontrolu tijeka programa jasno je da se prvo evaluira izraz u uvjetu i da će se, ako je SIZE jednak 10, ispisati HELLO i zatim kontrola programa prijeći na sljedeću izjavu, a ako SIZE nije jednak 10, ništa se neće ispisati i kontrola programa će prijeći na sljedeću izjavu. To je zato što, kada je IF izvršen, on postaje neaktivan i nevažan za ostatak

programa jer se kontrola više ne vraća unatrag na njega. Međutim, više od polovine ispitanika smatra da će se, kada unutar petlje varijabla SIZE postane jednaka 10, ipak ispisati HELLO. Objašnjenje koje učenici nude za takvo razmišljanje je da IF izjava čeka sve dok u petlji varijabla SIZE ne postane jednaka 10 kako bi onda mogla obaviti zadani ispis. Ono je posljedica pogrešnog shvaćanja da su sve linije programa aktivne istovremeno, a koje brzina izvršavanja programa samo još dodatno naglašava.

Drugi primjer paralelizma ogleda se u programu u kojem se izjave pridruživanja varijablama pojavljuju nakon izjava koje se pozivaju na te varijable. Npr.:

```
AREA = Height x Width
Input Heigh
Input Width
PRINT "AREA
```

Mnogi studenti ne vide problem u ovom programu i predviđaju da će on ispisati umnožak vrijednosti *visine* i *širine* koje korisnik unese (u PROLOG-u bi to i napravio). To, međutim, nije slučaj u programskom jeziku kojeg su studenti koristili jer u trenutku kada se prva izjava (u kojoj se AREA definira kao umnožak *visine* i *širine*) izvršava, ona još uvije nije primila ulazne vrijednosti *visine* i *širine* nego ih tretira kao "defaultne" vrijednosti, tj. 0. Zato je vrijednost umnoška 0x0=0 i ona će biti ispisana, a ne umnožak unesenih vrijednosti *visine* i *širine* kako studenti pretpostavljaju.

Oba primjera posljedica su utjecaja strategija koje se koriste u konverzaciji prirodnim jezikom i koje se potom prenose u programerski jezik te koje dodatno odražavaju nepoznavanje sekvencijalnog redoslijeda izvršavanja izjava u programu.

2) *Intencionalizam*

Intencionalizam je pogreška u kojoj studenti programu pridjeljuju usmjerenost cilju koja nadilazi informacije dane u linijama programskog kôda pretpostavljajući da će napraviti ono što nije eksplicitno navedeno i dajući mu ljudske karakteristike.

Primjer intencionalizma koji je dodatno istražen i opisan u [78] tražio je od studenata da predvide što će sljedeći rekurzivni program u Logu:

```
TO SHAPE :SIDE
IF :SIDE = 10 STOP
REPEAT 4 [FORWARD :SIDE RIGHT 90]
SHAPE :SIDE/2
END
```

napraviti nakon poziva SHAPE 40.

Neki studenti pogrešno predviđaju da će se nakon pokretanja programa nacrtati kvadrat stranice duljine 10. Njihova obrazloženja otkrivaju razloge takvog razmišljanja. Naime, u trenutku kada dođu na drugu programsku liniju koja sadrži IF izjavu, studenti pogledaju unaprijed na sljedeću liniju kôda da vide što ona radi (crta kvadrat), a onda se vrate natrag na IF izjavu i protumače je na način da će se nacrtati kvadrat stranice duljine 10 jer ga "program želi nacrtati". Neki od studenata ipak prepoznaju da je vrijednost varijable u IF izjavi 40, ali zatim kažu da program vidi u sljedećoj programskoj liniji izjavu za crtanje kvadrata kojeg želi nacrtati s tim da se mora zaustaviti na 10.

I u ovom primjeru, kao i u slučaju paralelizma, studenti programu daju osobine bića koje ima namjere i ciljeve te koje zna i vidi unutar sebe što će se dogoditi.

3) *Egocentrizam*

Egocentrizam je zrcalna strana intencionalizma koja se odnosi na pisanje programa dok se intencionalizam odnosi na razumijevanje i praćenje onoga što program radi. Primjer takve situacije je zadatak u kojem je potrebno u Logu napisati proceduru koja će crtati kvadrat stranice duljine 30, a studenti ponude rješenje REPEAT 4 [FORWARD 30]. Oni pritom misle da će program ipak nacrtati kvadrat jer su uključili naredbu za pomicanje kornjače naprijed, a računalo će samo dalje znati što se od njega očekuje kako bi ispunilo zadani cilj.

Sve tri pogreške posljedica su pridjeljivanja računalu ljudskih karakteristika koje ono nema.

Ulazni podaci su problematika o kojoj je pisalo nekoliko prethodno spomenutih istraživača [74], [75]. Studenti se pitaju odakle dolazi ulazni podatak, kako i gdje je pohranjen te kako je dostupan programu koji ga koristi. Haberman i Kolikant 2001. u [79] pretpostavljaju dva netočna modela za ulaznu obradu koje studenti koriste: *povijesni mehanizam* prema kojem je varijabla dinamično uređen skup vrijednosti koji se mijenja svaki put kada je izjava unosa izvršena na način da se nova vrijednost dodaje u taj uređen skup pa program s nekoliko ulaznih podataka koje se odnose na istu varijablu čuva sve vrijednosti ikad povezane s tom varijablom ili pak prema kojem se vrijednost jednom inicijalizirane varijable ne može više mijenjati te *prioritetni mehanizam* koji aktivira naredbe ne po njihovom redoslijedu pojavljivanja nego po njihovoj važnosti i prema kojem npr. naredba unosa podataka ima veći prioritet od naredbe pridruživanja ili prema kojem se vrijednost varijable može koristiti samo jednom u programu, a nakon što je korištena automatski joj se pridružuje sljedeća ulazna vrijednost.

Pogrešna shvaćanja često su vezana i uz rekurziju kao jedan od osnovnih koncepata programiranja [80]. Kahney [81] je otkrio da studenti imaju različite miskoncepcije, a najčešća je da rekurziju razumiju više kao iteraciju. Rekurziju je istraživala i Booth [82] te utvrdila postojanje triju različitih načina na koje se ona može doživjeti: kao programski konstrukt, kao način ponavljanja i kao samoreferenciranje. Studenti nisu uvijek u stanju ispravno shvatiti sve te aspekte. Scholtz i Sanders [83] smatraju da u poučavanju rekurzije više pažnje treba posvetiti objašnjavanju pasivnog dijela tijekom rekurzivnog poziva budući da studenti teško shvaćaju vraćanje unatrag koje se događa i vjeruju da rekurzivna funkcija završava na graničnom slučaju. Dodatan dokaz nerazumijevanja pasivnog dijela rekurzije kod njenog izvršavanja je i taj što mnogi studenti rekurziju vide slično petlji pa tu miskoncepciju prilikom poučavanja treba svakako dobro razjasniti. Do istog su zaključka prije 25 godina došli i

Kurland i Pea [42] koji također smatraju da u poistovjećivanju rekurzije s petljom leži glavni razlog neshvaćanju kontrole tijeka izvršavanja rekurzije, a time i same rekurzije. I drugi autori podupiru potpuno isto [54]. Götschi, Sanders i Galpin [60] dodaju da, pored navedenog, nerazumijevanje prosljeđivanja parametara i evaluacije povratne vrijednosti također može razvijati miskoncepcije budući su to predznanja nužna za shvaćanje izvršavanja rekurzivnih programa.

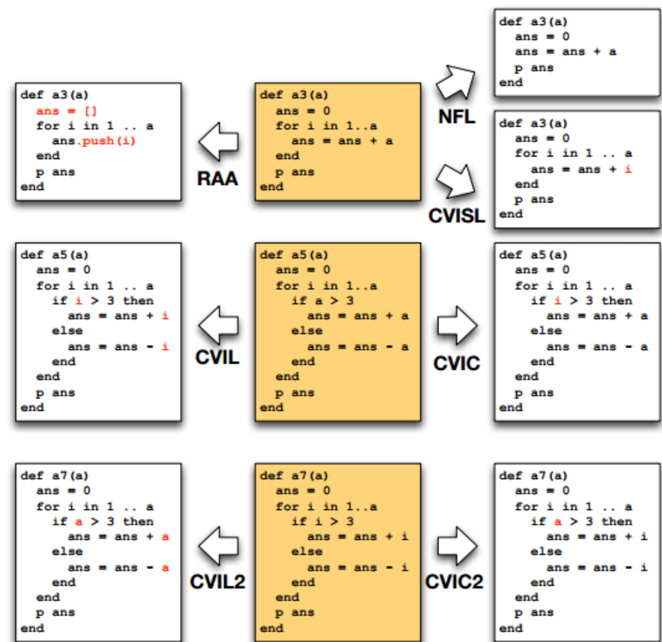
Soloway i suradnici [77] pronašli su da samo 38% početnika može ispravno napisati program za računanje prosjeka brojeva. Lopez i suradnici [65] utvrdili su da 60% studenata točno predviđa povratnu vrijednost metode koja u sebi uključuje *while* petlju, a 32% metode koja u sebi uključuje *for* petlju unutar koje je *if*.

Sekiya i Yamaguchi [84] također su se bavili razumijevanjem konstrukata uvjeta i petlje koje je često kod studenata nepotpuno. Njihovo istraživanje je pokazalo da programeri početnici razumiju uvjet i petlju zasebno, ali ne uspijevaju shvatiti njihovu kombinaciju. Autori predlažu postupak za pronalaženje miskoncepcija koje su u pozadini tog neuspjeha na način da su razvili niz kvizova praćenja kako bi locirali točna mjesta na kojima će studenti pogriješiti i iz njih izveli 7 mogućih miskoncepcija koje su, prema njihovom iskustvu, uzrokovale značajan dio tih grešaka. Eksperiment su proveli na 5 kolegija s oko 370 studenata, a Tablica 2 prikazuje popis dobivenih miskoncepcija s njihovim opisom i kraticom.

Tablica 2: Kratice, nazivi i opisi pronađenih miskoncepcija

Kratica misko-ncepije	Naziv miskoncepcije	Opis miskoncepcije
CVIC	Mijenja varijablu u uvjetu *(Change Variable in Condition)*	Interpretira varijable u dijelu IF-a u kojem se ispituju uvjeti kao kontrolne varijable
CVIC2	Mijenja varijablu u uvjetu 2 (*Change Variable in Condition 2*)	Interpretira varijable u dijelu IF-a u kojem se ispituju uvjeti kao krajnju vrijednost petlje
CVIL	Mijenja varijablu u petlji *(Change Variable in Loop)*	Interpretira varijable u svim dijelovima IF-a kao kontrolne varijable
CVIL2	Mijenja varijablu u petlji 2 (*Change Variable in Loop 2*)	Interpretira varijable u svim dijelovima IF-a kao krajnju vrijednost petlje
CVISL	Mijenja varijablu u jednostavnoj petlji (*Change Variable in Simple Loop*)	Interpretira varijable u tijelu IF-a kao kontrolne varijable
NFL	Zanemaruje, tj ignorira petlju (*Neglect, i.e. ignore For Loop*)	Ignorira petlju kada tijelo petlje nema kontrolne varijable
RAA	Smatra nizom (*Regards As Array*)	Interpretira zamjenu na način da zadržava prethodne vrijednosti

Grešku koja se može objasniti npr. NFL (*Neglect For Loop*) miskoncepcijom nazvali su NFL greškom i tako redom. Treba primijetiti da ista greška može biti i NFL greška i CVISL greška istovremeno jer se ista greška može objasniti pomoću više miskoncepcija. Za simuliranje miskoncepcija, izvorni Ruby kôd pretvoren je u drugi Ruby kôd koji odražava miskoncepciju. Ostali primjeri prikazani su na Slici 1.



Slika 1: Pretvorbe kôda koje odražavaju različite miskoncepcije

Provedeni eksperiment je pokazao da je, među 7 pronađenih miskoncepcija koje objašnjavaju značajan dio studentskih pogrešaka, NFL miskoncepcija najvažnija jer studenti s NFL miskoncepcijom zadatke rješavaju značajno lošije od drugih, stoga njeno rano otkrivanje može biti od pomoći.

Godine 2012. Sirkia i Sorva [56] završili su i objavili višegodišnje istraživanje o miskoncepcijama koje su utvrđene pomoću vizualnih programskih simulacija programskog jezika Python. Istraživanje su provodili 3 uzastopne godine, od 2010. do 2012., svake godine na uzorku od 600-700 studenata koliko ih je upisalo kolegij. Zaključili su da početnici rade mnogo različitih grešaka, a poznavanje koje su među njima najčešće omogućuje fokusiranje na njihovo uklanjanje. Dobivene pogreške povezane s miskoncepcijama podijelili su u 3 skupine i to:

- osnovni koncepti: pridruživanje, odabir i petlje
- funkcije
- OOP

Tablica 3 navodi 5 uobičajenih studentskih pogrešaka koje su povezane s miskoncepcijama o fundamentalnim konceptima programiranja (pogreške prve skupine):

Tablica 3: Pogreške vjerojatno povezane s miskoncepcijama o pridruživanju, kontrolnim strukturama i uvjetima

Pogreška	Postotak 2010	2011	2012
Misc 1: Obrnuto pridruživanje `first = second`	27%	26%	38%
Misc 2: Pogrešna grana `if divisor == 0:`	11%	17%	17%
Misc 3: Pogrešno *Netočno* `if not information_ok(place, distance):` `return False`	52%	35%	45%
Misc 4: Uvjet u varijablu `if divisor == 0:`	8%	8%	6%
Misc 5: Uvjet u kontrolnu varijablu petlje `while i < 7:`	8%	0%	3%

Prva pogreška, Misc 1 (*Obrnuto pridruživanje, tj. dodjeljivanje vrijednosti varijable s lijeve strane varijabli s desne strane, a ne obrnuto*), pokazuje isto što i mnoga druga istraživanja po tom pitanju u proteklih nekoliko desetljeća, a to je da se početnici u velikoj mjeri bore sa smjerom izjave pridruživanja. Ovo istraživanje dodatno pokazuje da tu vrstu greške rade samo u izravnom varijabla-u-varijablu pridruživanju dok u slučajevima kada je desna strana doslovan ili složen izraz nema analognih poteškoća. Zabrinjavajuće je što problem ne nestaje jednostavno i često se ponovo javlja u kasnijim fazama pisanja vlastitih programa. Ovim tipom pogreške bavili su se i mnogi drugi autori, počevši od prethodno navedenih [36], [74], [75], ali i brojni drugi. Sljedeće dvije pogreške, Misc 2 (*Pogrešna grana, tj. izvršavanje* then *dijela IF izjave iako uvjet nije ispunjen*) i Misc 3 (*Pogrešno Netočno, tj. funkcija vraća Netočno ako je uvjet evaluiran kao Netočan*), u kojima se ne reagira odgovarajuće na Boolean vrijednost izraza u uvjetu, mogu se vjerojatno pripisati onim početnicima koji smatraju da se *then* dio IF izjave uvijek izvršava te za Misc 3 dodatno njihovom relativnom neiskustvu s *not* operatorom i izrazima u uvjetu koji sadrže poziv funkcije. Pogreška Misc 4 (*Uvjet u varijablu, tj. dodjeljivanje vrijednosti izraza iz uvjeta varijabli nakon njegove evaluacije*) je lako objašnjiva i posljedica je miješanja == i = operatora te pokazuje kako početnici također mogu interpretirati == i kao pridruživanje. Unatoč površnoj sličnosti između Misc 4 i Misc 5 (*Uvjet u kontrolnu varijablu petlje, tj. dodjeljivanje vrijednosti izraza iz uvjeta nakon njegove evaluacije kontrolnoj varijabli petlje*), više iznenađuje podatak da do 8% učenika radi grešku Misc 5 koja izgleda kao prilično egzotičan pogrešan korak. Pogreška upućuje na miskoncepciju koja se odnosi na ulogu varijabli u *while* petljama.

B. *Miskoncepcije o pozivima funkcija*

Tablica 4 navodi najčešće pogreške povezane s miskoncepcijama o pozivima funkcija koje su u svom istraživanju pronašli Sirkia i Sorva (pogreške druge skupine):

Tablica 4: Pogreške vjerojatno povezane s miskoncepcijama o funkcijama

Pogreška	Postotak		
	2010	2011	2012
Func 1: Izvršavanje funkcije umjesto njenog definiranja `def ask_euros():`	17%	15%	16%
Func 2: Neevaluirani parametri `result = calculate(result, result+1)`	7%	N/A	10%
Func 3: Parametar u pogrešnom okviru stoga `def calculate(first, second):`	6%	7%	11%
Func 4: Zagubljena vraćena vrijednost `return second * 2 + first` `#...` `result = calculate (3,2)`	17%	27%	24%
Func 5: Vraćanje vrijednosti u varijablu `return intermediate * intermediate`	5%	9%	8%
Func 6: Nepohranjivanje vraćene vrijednosti `text = ask_euros()`	20%	16%	29%

Pogreška Func 1 (*Izvršavanje funkcije umjesto njenog definiranja*) odgovara skupini miskoncepcija u kojima se smatra da se potprogram izvršava čim započne izvršavanje programa. Naime, naredba *def* definira novu funkciju i početnici bi to trebali raditi, dakle, definirati funkciju, a umjesto toga, oni započinju s izvršavanjem naredbi tijela funkcije. Pogreška Func 2 (*Neevaluirani parametri, tj. pozivanje funkcije prije evaluiranja izraza*) puno govori o miskoncepciji u kojoj studenti misle da se neevaluirani izrazi prosljeđuju dalje kao parametri umjesto njihovih vrijednosti što može biti posljedica slabog razumijevanja koncepta evaluacije izraza. To tumačenje analogno je onima iz izvještaja u kojima se za objekte misli da su dio kôda [73], a za izjave pridruživanja da su dijelovi jednadžbe [74]. Pogreška Func 3 (*Parametar u pogrešnom okviru stoga*) pokazuje da neki početnici nisu razumjeli svrhu stoga i kako je on povezan s varijablama koje su dostupne tijekom poziva funkcije. To je u suglasju s miskoncepcijom o prosljeđivanju parametara u kojoj su studenti zbunjeni oko toga koje su varijable dostupne tijekom kojeg poziva funkcije i oko toga gdje su vrijednosti varijabli pohranjene u memoriji. Pogreška Func 4 (*Zagubljena vraćena vrijednost, tj. kreiranje varijable u okviru stoga funkcije umjesto vraćanja vrijednosti*) je slična, ali uključuje vraćenu vrijednost umjesto parametra. Tako studenti, umjesto da vrate vrijednost funkcije putem *return* naredbe, spremaju rezultat u varijablu unutar okvira stoga funkcije. Pogreška Func 5 (*Vraćanje vrijednosti u varijablu*) govori o miskoncepciji pridruživanja rezultata obavljene računske radnje natrag u varijablu. Nakon što vrate vrijednost, početnici često ne znaju što bi s njom, kao što pokazuje pogreška Func 6 (*Nepohranjivanje vraćene vrijednosti*). Ta je pogreška vjerojatno povezana s nerazumijevanjem da vraćena vrijednost treba biti sačuvana u slučaju da kasnije bude potrebna. Studenti često misle da mogu koristiti lokalne varijable pozvane funkcije čak i nakon vraćanja vrijednosti. Ova je miskoncepcija opisana i u radu [85].

C. *Miskoncepcije o objektno-orijentiranim konceptima*

Isti autori pronašli su i pogreške povezane s miskoncepcijama o objektno-orijentiranim konceptima (pogreške treće skupine). One su navedene u Tablici 5.

Tablica 5: Pogreške vjerojatno povezane s miskoncepcijama o objektno-orijentiranim temama

Pogreška	Postotak		
	2010	2011	2012
OO 1: Pridruživanje kopira objekt `car3 = car1`	12%	6%	14%
OO 2: Neuspjelo kreiranje drugog objekta klase `car2 = Car(60)`	16%	7%	18%
OO 3: Lokalna varijabla postaje varijabla instance `car1 = Car(45)`	7%	10%	11%
OO 4: Varijabla instance postaje lokalna varijabla `self.__name = firstname`	8%	7%	3%
OO 5: Poziv metode bez primatelja `car1.fuel(40)`	18%	38%	33%
OO 6: Dodjeljivanje reference neinicijaliziranom objektu `car1 = Car(45)`	6%	12%	9%
OO 7: Vraćanje reference *self* umjesto njenog dereferenciranja `return self.__profession`	6%	15%	15%

Pogreška OO 1 (*Pridruživanje kopira objekt, tj kreiranje novog objekta umjesto kopiranja reference*) odgovara miskoncepciji prema kojoj pridruživanje objekata varijablama stvara njihove kopije (čak i tamo gdje su zaista korištene reference na objekte). Ova je pogreška vrlo česta. Pogreške OO 2 (*Neuspjelo kreiranje drugog objekta klase, tj. umjesto kreiranja novog objekt, pravi se kopija reference na postojeći objekt*) i OO 3 (*Lokalna varijabla postaje varijabla instance, tj. umjesto kreiranja nove lokalne varijable za pohranjivanje reference stvara se varijabla instance za novi objekt*) vjerojatno su povezane s dobro poznatom i učestalom poteškoćom koju početnici imaju s razumijevanjem odnosa između klasa i objekata. Neki studenti koji rade pogrešku OO 2 možda vjeruju da može postojati samo jedna instanca svake klase. Pogreška OO 3 vjerojatno je povezana s razmišljanjem da je varijabla koja referencira objekt dio tog istog objekta. Pogreška OO 4 (*Varijabla instance postaje lokalna varijabla, tj. umjesto kreiranja nove varijable instance za objekt, stvara se nova lokalna varijabla*) također puno govori o konceptualnoj konfuziji između lokalnih varijabli i varijabli instance. Pogreška OO 5 (*Poziv metode bez primatelja, tj. pokušaj poziva metode klase prije pristupanja objektu ili referenci na objekt*) uobičajena je pogreška koja se djelomično može objasniti nepoznavanjem pojma objekata: zbog ograničenog izlaganja primjerima u kojima je redoslijed evaluacije važan, početnici ne uspijevaju razumjeti zašto je referenca objekta nužna prije nego se poziv može napraviti. Oni možda intuitivno osjećaju da je "logičnije" prvo dohvatiti metodu, a zatim ciljani objekt. Pythonovi eksplicitni *self* parametri u definicijama metoda možda dodatno doprinose učestalosti ove pogreške. Pogreška OO 6 (*Dodjeljivanje reference neinicijaliziranom objektu*) odnosi se na dodjeljivanje reference na objekt prije pozivanja *_ _init_ _* metode, a pogreška OO 7 (*Vraćanje reference* self *umjesto njenog dereferenciranja*) odnosi se na vraćanje *self* umjesto vrijednosti varijable instance.

I drugi su se autori bavili pogrešnim konceptima u objektno-orijentiranoj paradigmi programiranja. 2012. godine Chen i suradnici [86] su, među 84 studenta, odabrali 22 s najlošijim uspjehom na kraju semestra i kroz intervjue pokušali doznati razloge njihovog neuspjeha. Na taj su način otkrili koje su miskoncepcije u pozadini njihovog neuspjeha. Dobiveni rezultati prikazani su u Tablici 6:

Tablica 6: Miskoncepcije u objektno-orijentiranom programiranju

	Miskoncepcija	**Postotak javljanja misko-ncepcije**
1.	*Statički podatkovni članovi vs. konstantni podatkovni članovi* Miješanje svojstava statičkih podatkovnih članova i konstantnih podatkovnih članova	14 %
2.	*Klase vs. objekti* Neuspjelo prepoznavanje da su klase zapravo korisnički definirani tipovi podataka koji se mogu koristiti za definiranje varijabli baš kao ugrađeni tipovi podataka, npr. int ili boolean	36 %
3.	*Konstruktori* 1) Pogrešno razmišljanje da su konstruktori svih klasa definiranih u projektu pozivani automatski kada je projekt otvoren 2) Definiranje konstruktora klase kao njenog privatnog člana iako bi trebao biti javni član 3) Neshvaćanje da konstruktori, kao posebna vrsta metode, nemaju povratni tip	14 % 47 % 50 %
4.	*Prosljeđivanje argumenata u pozivu metoda* Miješanje formalnih i stvarnih parametara	31 %
5.	*Polimorfizam* 1) Pogrešno shvaćanje da deklariranje jedne varijable nadklase umjesto dviju varijabli podklase znači deklariranje dviju varijabli u istoj liniji 2) Pogrešno shvaćanje da smještanje dvaju objekata podklasa sukcesivno u istu varijablu nadklase znači zamjenjivanje dvaju objekata nadklase dvama objektima podklase	14 % 45 %
6.	*Pristup modifikatorima metoda* Pogrešno shvaćanje da privatna metoda ne može pozvati metode definirane u drugim klasama	12 %

Holland i suradnici [72] tvrde da poučavanje objektno-orijentiranog programiranja nudi velike mogućnosti za razvijanje miskoncepcija koje poslije više nije lako ispraviti. Također su uočili da početnici teško prave razliku između klasa i njihovih instanci, odnosno objekata, koje vide samo kao zapise podataka ne obazirući se na njihove aspekte ponašanja i činjenicu da se ponašanja objekta mogu suštinski mijenjati ovisno o njegovom stanju. Sanders i Thomas 2007. u svom su radu [25] utvrdile da studenti imaju poteškoća u razlikovanju pojmova klasa i objekata, ali su u nekim slučajevima ti problemi uočeni samo u situacijama kad ih definiraju sami korisnici, a ne i onih iz biblioteka. Klase se tretiraju kao kolekcije objekata umjesto kao apstrakcije ili predlošci za njihovo kreiranje. Do istog su zaključka došli i Ragonis i Ben-Ari [87] 2005. tvrdeći uz to da studenti imaju poteškoća u razlikovanju klasa i objekata kada je atribut inicijaliziran konstantom unutar deklaracije konstruktora. Teif i Hazzan [88] utvrdile su da početnici miješaju vezu između klase i objekta s vezom između skupa i podskupa koja je u objektno-orijentiranoj paradigmi, u stvari, veza između nadklase i podklase te vezu između klase i objekta s vezom između cjeline i njenih dijelova, npr. pogrešno zaključivanje da je pas-rep klasa-objekt veza. Eckerdal i Thuné [73] su kroz opsežne intervjue sa studentima uočili miskoncepcije u kojima objekt vide ili kao vrstu varijable ili kao neku vrstu omotača za varijable te miskoncepciju koja se pojavljuje kada je podatkovni aspekt objekta prenaglašen u odnosu na njegov aspekt ponašanja. Nerazumijevanje razlike između klase i objekata pripisuju nizu loše odabranih uvodnih primjera kojima su studenti izloženi, a u kojima svaka klasa ima samo jednu instancu pa u cilju izbjegavanja tog problema preporučuju dobru praksu korištenje primjera s nekoliko

instanci svake klase. Isti bi pristup pomogao i u izbjegavanju pojavljivanja miskoncepcije u kojoj se na objekte gleda kao na omotače varijabli dok bi se miskoncepcija nastala kao posljedica prenaglašenog podatkovnog aspekta mogla izbjeći preko dobro osmišljenih primjera objekata u kojima se odgovor na poruku značajno mijenja ovisno o stanju objekta. U radu [89], Xinogalos je 2015. pokazala da početnici ipak bolje razumiju koncept klase nego koncept objekta. Kaczmarczyk i suradnici [90] pak ukazali su na nedostatak čak i najosnovnijeg koncepta o objektima. Oni tvrde da neki studenti u njihovom istraživanju nisu imali formulirane miskoncepcije o objektima jer nisu imali ama baš nikakve koncepcije o njima. Jedna važna implikacija nedostatka tog znanja, prema zamislima autora, mogla bi biti da se nastavnici vrate korak unatrag u svom poučavanju i ponovo razmisle o tome kako uvesti pojam objekta fokusirajući se pritom eksplicitno na ono što smatraju najkritičnijim u tom konceptu. Dodatno, u svom su istraživanju kroz intervjue sa studentima pokušali grupirati sve pronađene miskoncepcije tematski i pritom su identificirali 4 teme: veza između jezičnih elemenata i memorije, *while* petlje, koncept objekta i sposobnost praćenja kôda. Sejaniemi i suradnice [71] otkrile su 2008. dva glavna uzroka problema: 1) odnos između objekta i metode i 2) uloga glavne metode *main* obzirom na stanje programa (eng. *program state*). Ti problemi manifestiraju se kroz različite miskoncepcije od pridruživanja vrijednosti podatkovnim članovima do referenci na objekt smještenih unutar samog objekta. Sorva [91] nije iznenađen što studenti imaju problema sa složenošću objektno-orijentiranog programiranjem kada je njihovo znanje osnovnih konstrukata klimavo ili netočno. Njegovo istraživanje ukazalo je na problematičnost odnosa između primitivnih i varijabli objekata jer neki od studenata ove dvije vrste varijabli vide ili kao potpuno odvojene konstrukte unatoč površnoj sličnosti u sintaksi ili smatraju da sve varijable funkcioniraju na isti način i imaju istu vrstu odnosa s podacima s kojima su povezane ili ih pogrešno razumijevaju zbog izloženosti matematičkim varijablama i jednadžbama ili konceptualno stapaju objekte i varijable koje ih referenciraju. Carter i Fowler [92] utvrdile su da je studentima zahtjevno pisanje programa koji imaju više od jedne klase jer ih zbunjuje nedoumica trebaju li koristiti odvojene datoteke za svaku klasu ili ih mogu smjestiti u istu datoteku. Nastavak tog problema pisanje je programa koji uključuju kompoziciju klasa [87]. Definiranje klase koja sadrži atribute drugih klasa vrlo je izazovno te povlači za sobom nekoliko pogrešnih shvaćanja povezanih s razumijevanjem enkapsulacije, modularnosti i skrivanja informacija. Enkapsulaciju problematizira i Fluery [93], a ona se očituje kroz izbjegavanje davanja istih imena metodama u različitim klasama. Osim tvrdnje da dvije klase mogu imati samo metode čija se imena ne podudaraju, utvrdila je postojanje i drugih pogrešnih pravila koje formiraju studenti kao što su: jedina svrha pozivanja konstruktora inicijalizacija je varijabli instance, parametri metoda mogu biti samo brojevi te *dot* operator može biti primijenjen samo na metode. Vrlo je zanimljivo uočiti da uklanjanjem riječi "samo" ili "jedino" u svim tim pravilima, u stvari, dobivamo ispravne tvrdnje što navodi na zaključak da su studenti stvorili netočna pravila pogrešnim primjenjivanjem točnih pravila.

Sanders i Thomas [25] proučavale su rezultate istraživanja u različitim područjima objektno-orijentiranog programiranju i na temelju toga 2007. objavile zajedničku listu uočenih poteškoća koje se pritom javljaju :

1) Pogreške u osnovnim mehanizmima: definiranje klasa, varijabli instanci i metoda
2) Instanca/klasa poistovjećivanje: klase i instance su isto
3) Problemi s povezivanjem i interakcijom
4) Klasa/kolekcija poistovjećivanje: klasa je kolekcija instanci, a ne apstrakcija svojih instanci
5) Problemi s apstrakcijama i hijerarhijama (uključujući nasljeđivanje, apstraktne metode, preopterećenje metoda, sučelja)
6) Problemi modeliranja: neshvaćanje da klasa oblikuje nešto u domeni problema
7) Klasa samo tekst (ne pridajući mu značenje)
8) Klasa/varijabla poistovjećivanje: klase su samo omotači za varijable instance
9) Identitet/atribut zbunjenost: varijabla koja referencira objekt dio je identiteta objekta isto kao i njegovo ime
10) Uvjerenje da su objekti jednostavni zapisi za pohranjivanje i dohvaćanje podataka
11) Problemi s enkapsulacijom
12) Rad u metodama je završen pridruživanjem, a ne prosljeđivanjem poruke
13) Problemi s akcesorima/mutatorima (geteri/seteri)
14) Problemi s konstruktorima

Nakon istraživanja ovih miskoncepcija među svojim studentima, sastavile su i dvije liste koje bi trebale pomoći drugim nastavnicima u poučavanju objektno-orijentiranog programiranja te osmišljavanju zadataka kojima će testirati određene koncepte i miskoncepcije.

VI. ZAKLJUČAK

Programiranje je poseban način razmišljanja kojim se od postavljenog problema dolazi do rješenja izvedivog na računalu. Zato je učenje i poučavanje programiranja danas tako važno jer kod učenika razvija strategije rješavanja problema i sposobnosti snalaženja u novim situacijama te mnoge druge temeljne kompetencije koje će im biti od koristi ne samo tijekom njihovog obrazovanja nego i šire.

Poučavanje programiranja bit će uspješnije ukoliko nastavnici steknu uvid u ono što učenici znaju, ali također i u ono što učenici ne znaju. Postupci provjeravanja učeničkog razumijevanja nastavnog gradiva često otkrivaju da su tumačenja učenika u neskladu sa znanstvenim spoznajama. Zbog toga je važno upoznati najčešće poteškoće i pogrešna shvaćanja učenika te razloge koji ih uzrokuju. Miskoncepcije se često javljaju kod učenika jer su to, iako netočna, jednostavna objašnjenja koja učenici lakše i brže usvajaju. One su sastavni dio procesa učenja i pravi je izazov prevladati ih. Kako bi se ispravio što veći broj pogrešno percipiranih pojmova, potrebno ih je prvo utvrditi. U tu svrhu najbolje mogu poslužiti pogrešni odgovori učenika iako se miskoncepcije ponekad skrivaju i iza njihovih točnih

odgovora. Veliki problem nakon utvrđivanja miskoncepcija predstavlja proces njihovog ispravljanja i objašnjavanja učenicima u čemu su griješili kako bi se omogućilo usvajanje valjanih koncepata. Pritom je potrebno razvijati ispravne mentalne modele kod učenika jer netočni konceptualni modeli vode do nerazumijevanja, a nerazumijevanje vodi do novih miskoncepcija. Miskoncepcije se u manjoj mjeri ispravljaju kod tradicionalnih oblika nastave jer učenik nije dovoljno zainteresiran da promjeni svoje mišljenje o nekom pojmu, dok mu aktivan oblik rada omogućava samostalno donošenje ispravnih zaključaka i sustavno organiziranje znanja.

Brojna istraživanja u području računalne znanosti pokazuju da su kod programiranja općenito najčešće miskoncepcije vezane uz osnovne strukture kao što su varijable, grananja i ponavljanja te kontrolu tijeka programa, a kod objektno-orijentirane paradigme dodatno uz pojam klasa i objekata. Velik dio tih istraživanja pokazuje također da se iste miskoncepcije pojavljuju još od ranih 80-tih godina prošlog stoljeća pa sve do današnjih dana i da su vrlo otporne na protok vremena. Nastavnici stoga neizostavno trebaju osvijestiti postojanje tih pogrešnih shvaćanja kako bi ih uključili u planiranje svog poučavanja i time ga učinili uspješnijim.

LITERATURA

[1] P. McKenna, "Gender and Black Boxes in the Programming," *Journal on Educational Resources in Computing (JERIC),* vol. 4, no. 1, 2004.

[2] A. J. Perlis, "The computer in the university," in *Computers and the World of the Future*, Cambridge, 1962.

[3] M. Felleisen, R. B. Findler, M. Flatt and S. Krishnamurthi, How to Design Programs: An Introduction to Programming and Computing, Second edition, London, England: The MIT Press, 2001.

[4] M. Guzdial, Learner-Centered Design of Computing Education: Research on Computing for Everyone, Morgan & Claypool, 2016.

[5] M. C. Lewis, Introduction to the Art of Programming Using Scala, Boca Raton: CRC Press, 2013.

[6] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman and Y. Kafai, "Scratch: programming for all," *Communications of the ACM,* vol. 52, no. 11, pp. 60-67, 2009.

[7] M. Hassinen and H. Mäyrä, "Learning programming by programming: a case study," in *Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling 2006*, Koli, Finland, 2006.

[8] M. Prensky, "The True 21st Century Literacy Is Programming," in *From Digital Natives to Digital Wisdom: Hopeful Essays for 21st Century Learning*, Thousand Oaks, California, Corwin, 2012, pp. 192-199.

[9] A. Gomes and A. J. Mendes, "Learning to program - difficulties and solutions," in *International conference on Engineering Education*, Coimbra, Portugal, 2007.

[10] S. Dehnadi, "A Cognitive Study of Learning to Program in Introductory Programming Courses," Middlesex University, London, 2009.

[11] J. Bonar and E. Soloway, "Uncovering principles of novice programming," in *POPL '83 Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1983.

[12] A. Robins, J. Rountree and N. Rountree, "Learning and Teaching Programming: A Review and Discussion," *Computer Science Education,* vol. 13, no. 2, pp. 137-172, 2003.

[13] E. Lahtinen, K. Ala-Mutka and H.-M. Järvinen, "A Study of the Difficulties of Novice Programmers," *ACM SIGCSE Bulletin,* vol. 37, no. 3, pp. 14-18, 2005.

[14] M. Guzdial, "Programming Environments for Novices," *Computer Science Education Research,* vol. 111, pp. 127-155, 2003.

[15] I. Milne and G. Rowe, "Difficulties in Learning and Teaching Programming—Views of Students and Tutors," *Education and Information Technologies,* vol. 7, no. 1, pp. 55-66, 2002.

[16] L. Ma, J. Ferguson, M. Roper and M. Wood, "Investigating and Improving the Models of Programming Concepts Held by Novice Programmers," *Computer Science Education,* vol. 21, no. 1, pp. 57-80, 2011.

[17] K. M. Fisher, "A misconception in biology: Amino acids and translation," *Journal of Research in Science Teaching,* vol. 22, no. 1, p. 53–62, 1985.

[18] O. Hazzan, T. Lapidot and N. Ragonis, Guide to Teaching Computer Science: An Activity-Based Approach, London: Springer Publishing Company, 2011.

[19] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B.-D. Kolikant, C. Laxer, L. Thomas, I. Utting and T. Wilusz, "A multi-national, multi-institutional study of assessment of programming skills of first-year CS students," *ACM SIGCSE Bulletin,* vol. 33, no. 4, pp. 125-180, 2001.

[20] R. D. Pea, "Language-independent conceptual ”bugs” in novice programming," *Journal educational computing research,* vol. 2, no. 1, pp. 25-36, 1986.

[21] K. D. Sloane and M. C. Linn, "Instructional Conditions in Pascal Programming Classes," in *Teaching and Learning Computer Programming: Multiple Research Perspectives*, New Jersey, Lawrence Erlbaum Associates, 1998, pp. 137-152.

[22] L. E. Winslow, "Programming pedagogy—a psychological overview," *ACM SIGCSE Bulletin,* vol. 28, no. 3, pp. 17-22, September 1993.

[23] G. Bain and I. Barnes, "Why Is Programming So Hard to Learn?," in *ITiCSE '14 Proceedings of the 2014 conference on Innovation & technology in computer science education*, Uppsala, Sweden, 2014.

[24] R. Lister, E. S. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, R. McCartney, J. E. Moström, K. Sanders, O. Seppälä, B. Simon and L. Thomas, "A multi-national study of reading and tracing skills in novice programmers," *ACM SIGCSE Bulletin,* vol. 36, no. 4, pp. 119-150, 2004.

[25] K. Sanders and L. Thomas, "Checklists for grading object-oriented CS1 programs: concepts and misconceptions," in *ITiCSE '07 Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education*, Dundee, Scotland, UK, 2007.

[26] J. Piaget, The origins of intelligence in children, New York: International Universities Press, 1952.

[27] E. Fusco, "Matching Curriculum to Students' Cognitive Levels," *Educational Leadership,* vol. 39, no. 1, p. 47, October 1981.

[28] B. S. Bloom, Taxonomy of Educational Objectives: The Classification of Educational Goals, Volume 1, New York: Longmans, Green, 1956.

[29] M. Kölling, "The Greenfoot Programming Environment," *ACM Transactions on Computing Education (TOCE),* 2010.

[30] M. C. Linn and J. Dalbey, "Cognitive consequences of Programming Instruction: Instruction, Access, and Ability," *Educational Researcher,* vol. 14, no. 5, pp. 14-29, 1985.

[31] B. C. Wilson and S. Shrock, "Contributing to success in an introductory computer science course: a study of twelve factors," *ACM SIGCSE Bulletin,* vol. 33, no. 1, pp. 184-188, March 2001.

[32] L. Lambert, "Factors that predict success in CS1," *Journal of Computing Sciences in Colleges,* vol. 31, no. 2, pp. 165-171, 2015.

[33] P. Byrne and G. Lyons, "The effect of student attributes on success in programming," in *ITiCSE '01 Proceedings of the 6th annual conference on Innovation and technology in computer science education*, Canterbury, UK, 2001.

[34] N. Rountree, J. Rountree, A. Robins and R. Hannah, "Interacting factors that predict success and failure in a CS1 course," *ACM SIGCSE Bulletin,* vol. 36, no. 4, pp. 101-104, 2004.

[35] Simon, S. Fincher, A. Robins, B. Baker, I. Box, Q. Cutts, M. de Raadt, P. Haden, J. Hamer, M. Hamilton, R. Lister, M. Petre, K. Sutton, D. Tolhurst and J. Tutty, "Predictors of Success in a First Programming

Course," in *ACE '06 Proceedings of the 8th Australasian Conference on Computing Education*, Darlinghurst, Australia, 2006.

[36] B. Du Boulay, "Some Difficulties of Learning to Program," *Journal of Educational Computing Research,* vol. 2, no. 1, pp. 57-73, 1986.

[37] D. N. Perkins and R. Simmons, "Patterns of Misunderstanding: An Integrative Model for Science, Math, and Programming," *Review of Educational Research,* vol. 58, no. 3, pp. 303-326, 1988.

[38] R. E. Mayer, "The Psychology of How Novices Learn Computer Programming," *ACM Computing Surveys (CSUR),* vol. 13, no. 1, pp. 121-141, 1981.

[39] T. Green, "Cognitive Approaches to Software Comprehension: Results, Gaps and Limitations," University of Limerick, Limerick, Ireland, 1997.

[40] J. C. Spohrer and E. Soloway, "Novice mistakes: are the folk wisdoms correct?," *Communications of the ACM,* vol. 29, no. 7, pp. 624-632, 1986.

[41] J. Bonar and E. Soloway, "Preprogramming knowledge: a major source of misconceptions in novice programmers," *Human-Computer Interaction,* vol. 1, no. 2, pp. 133-161, 1985.

[42] D. M. Kurland and R. D. Pea, "Children's mental models of recursive logo programs," *Journal educational computing research,* vol. 1, no. 2, pp. 235-243, 1985.

[43] N. C. Brown and A. Altadmri, "Investigating novice programming mistakes: educator beliefs vs. student data," in *ICER '14 Proceedings of the tenth annual conference on International computing education research*, Glasgow, United Kingdom, 2014.

[44] G. White, "Misconceptions in CIS education," *Journal of Computing Sciences in Colleges,* vol. 16, no. 3, pp. 149-152, 2001.

[45] K. D. Powers and D. T. Powers, "Constructivist Implications of Preconceptions in Computing," in *ISECON 2000 The Proceedings of the Information Systems Education Conference*, Philadelphia, PA, USA, 2000.

[46] M. Ben-Ari, "Constructivism in computer science education," in *SIGCSE '98 Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education*, Atlanta, GA, USA, 1998.

[47] T. R. G. Green, "Instructions and descriptions: some cognitive aspects of programming and similar activities," in *AVI '00 Proceedings of the working conference on Advanced visual interfaces*, Palermo, Italy, 2000.

[48] G. Alexandron, M. Armoni, M. Gordon and D. Harel, "The effect of previous programming experience on the learning of scenario-based programming," in *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*, Koli, Finland, 2012.

[49] H. Taylor and L. Mounfield, "An analysis of success factors in college computer science: High school methodology is a key element," *Journal of Research on Computing in Education,* vol. 24, no. 2, pp. 240-245, 1991.

[50] F. Halasz and T. P. Moran, "Analogy considered harmful," in *CHI '82 Proceedings of the 1982 Conference on Human Factors in Computing Systems*, 1982.

[51] C. o. U. Science, "Misconceptions as Barriers to Understanding Science," in *Science Teaching Reconsidered: A Handbook*, Washington D.C., National Academy Press, 1997.

[52] K. Strike and G. Posner, "Conceptual change and science teaching," *European Journal of Science Education,* vol. 4, no. 3, pp. 231-240, 1982.

[53] R. E. Mayer, Thinking, Problem Solving, Cognition, 2nd edition, New York: W.H. Freeman and Company, 1992.

[54] I. Sanders, V. Galpin and T. Götschi, "Mental models of recursion revisited," in *ITICSE '06 Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*, Bologna, Italy, 2006.

[55] L. Ma, "Investigating and Improving Novice Programmers' Mental Models of Programming Concepts," University of Strathclyde, Glasgow, 2007.

[56] T. Sirkiä and J. Sorva, "Exploring programming misconceptions: an analysis of student mistakes in visual program simulation exercises," in *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*, Koli, Finland, 2012.

[57] S. Denier and H. Sahraoui, "Understanding the use of inheritance with visual patterns," in *2009 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM)*, FL, USA, 2009.

[58] N. F. A. Zainal, S. Shahrani, N. F. M. Yatim, R. A. Rahman, M. Rahmat and R. Latih, "Students' perception and motivation towards programming," *Procedia - Social and Behavioral Sciences,* vol. 59, p. 277 – 286, 2012.

[59] A. Settle, A. Vihavainen and J. Sorva, "Three Views on Motivation and Programming," in *ITiCSE '14 Proceedings of the 2014 conference on Innovation & technology in computer science education*, Uppsala, Sweden, 2014.

[60] T. Götschi, I. Sanders and V. Galpin, "Mental models of recursion," in *SIGCSE '03 Proceedings of the 34th SIGCSE technical symposium on Computer science education*, Reno, NV, USA, 2003.

[61] L. Ohrndorf, "Measuring Knowledge of Misconceptions in Computer Science Education," in *ICER '15 Proceedings of the eleventh annual International Conference on International Computing Education Research*, Omaha, Nebraska, USA, 2015.

[62] R. Johan and S. Bull, "Consultation of Misconceptions Representations by Students in Education-Related Courses," in *Proceedings of the 2009 conference on Artificial Intelligence in Education: Building Learning Systems that Care: From Knowledge Representation to Affective Modelling*, Amsterdam, The Netherlands, 2009.

[63] S. Garner, P. Haden and A. Robins, "My program is correct but it doesn't run: a preliminary investigation of novice programmers' problems," in *ACE '05 Proceedings of the 7th Australasian conference on Computing education - Volume 42*, Darlinghurst, Australia, 2005.

[64] M. Corney, R. Lister and D. Teague, "Early relational reasoning and the novice programmer: swapping as the "hello world" of relational reasoning," in *ACE '11 Proceedings of the Thirteenth Australasian Computing Education Conference - Volume 114*, Darlinghurst, Australia, 2011.

[65] M. Lopez, J. Whalley, P. Robbins and R. Lister, "Relationships between reading, tracing and writing skills in introductory programming," in *ICER '08 Proceedings of the Fourth international Workshop on Computing Education Research*, Sydney, Australia, 2008.

[66] Simon, "Assignment and sequence: why some students can't recognise a simple swap," in *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*, Koli, Finland, 2011.

[67] R. Or-Bach and I. Lavy, "Cognitive activities of abstraction in object orientation: an empirical study," *ACM SIGCSE Bulletin,* vol. 36, no. 2, pp. 82-86, 2004.

[68] B. Thomasson, M. Ratcliffe and L. Thomas, "Identifying novice difficulties in object oriented design," in *ITICSE '06 Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*, Bologna, Italy, 2006.

[69] R. Samurçay, "The concept of variable in programming: its meaning and use in problem solving by novice programmers," in *Studying the Novice Programmer*, New Jersey, Lawrence Erlbaum Associates, 1989, p. 161–178.

[70] A. E. Fleury, "Parameter passing: the rules the students construct," in *SIGCSE '91 Proceedings of the twenty-second SIGCSE technical symposium on Computer science education*, San Antonio, TX, USA, 1991.

[71] J. Sajaniemi, M. Kuittinen and T. Tikansalo, "A study of the development of students' visualizations of program state during an elementary object-oriented programming course," *Journal on Educational Resources in Computing (JERIC),* vol. 7, no. 4, p. Article No. 3, 2008.

[72] S. Holland, R. Griffiths and M. Woodman, "Avoiding object misconceptions," in *SIGCSE '97 Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education*, San Jose, CA, USA, 1997.

[73] A. Eckerdal and M. Thuné, "Novice Java programmers' conceptions of "object" and "class", and variation theory," in *ITiCSE '05 Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, Caparica, Portugal, 2005.

[74] P. Bayman and R. E. Mayer, "A diagnosis of beginning programmers' misconceptions of BASIC programming statements," *Magazine Communications of the ACM,* vol. 26, no. 9, pp. 677-679, 1983.

[75] R. Putnam, D. Sleeman, J. Baxter and L. Kuspa, "A Summary of Misconceptions of High School Basic Programmers," *Journal of Educational Computing Research,* vol. 2, no. 4, pp. 459-472, 1986.

[76] A. Robins, "Learning edge momentum: a new account of outcomes in CS1," *Computer Science Education,* vol. 20, no. 1, pp. 37-71, 2011.

[77] E. Soloway, K. Ehrlieh, J. Boaar and J. Greeaspan, "What do novices know about programming?," in *Human-Computer Interaction - HCI*, 1982.

[78] R. D. Pea and D. M. Kurland, "On the Cognitive Effects of Learning Computer Programming," *New Ideas in Psychology,* vol. 2, no. 2, pp. 137-168, 1984.

[79] B. Haberman and Y. B.-D. Kolikant, "Activating “black boxes” instead of opening “zipper” - a method of teaching novices basic CS concepts," in *ITiCSE '01 Proceedings of the 6th annual conference on Innovation and technology in computer science education*, Canterbury, Kent, United Kingdom, 2001.

[80] M. Raynal, "Concurrency-Related Distributed Recursion," in *Stabilization, Safety and Security of Distributed Systems*, Osaka, Japan, 2013.

[81] H. Kahney, "What do novice programmers know about recursion," in *CHI '83 Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 1983.

[82] S. Booth, Learning to program : a phenomenographic perspective, Göteborg: Acta Universitatis Gothoburgensis, 1992.

[83] T. L. Scholtz and I. Sanders, "Mental models of recursion: investigating students' understanding of recursion," in *ITiCSE '10 Proceedings of the fifteenth annual conference on Innovation and technology in computer science education*, Ankara, Turkey, 2010.

[84] T. Sekiya and K. Yamaguchi, "Tracing quiz set to identify novices' programming misconceptions," in *Proceedings of the 13th Koli Calling International Conference on Computing Education Research*, Koli, Finland, 2013.

[85] M. Hristova, A. Misra, M. Rutter and R. Mercuri, "Identifying and correcting Java programming errors for introductory computer science students," in *SIGCSE '03 Proceedings of the 34th SIGCSE technical symposium on Computer science education*, Reno, NV, USA, 2003.

[86] C.-L. Chen, S.-Y. Cheng and J. M.-C. Lin, "A Study of Misconceptions and Missing Conceptions of Novice Java Programmer," in *Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS)*, Las Vegas, Nevada, USA, 2012.

[87] N. Ragonis and M. Ben-Ari, "A long-term investigation of the comprehension of OOP concepts by novices," *Computer Science Education,* vol. 15, no. 3, pp. 203-221, 2005.

[88] M. Teif and O. Hazzan, "Partonomy and taxonomy in object-oriented thinking: junior high school students' perceptions of object-oriented basic concepts," in *ITiCSE-WGR '06 Working group reports on ITiCSE on Innovation and technology in computer science education*, Bologna, Italy, 2006.

[89] S. Xinogalos, "Object-Oriented Design and Programming: An Investigation of Novices’ Conceptions on Objects and Classes," *ACM Transactions on Computing Education,* vol. 15, no. 3, 2015.

[90] L. C. Kaczmarczyk, E. R. Petrick, J. P. East and G. L. Herman, "Identifying Student Misconceptions of Programming," in *SIGCSE '10 Proceedings of the 41st ACM technical symposium on Computer science education*, Milwaukee, WI, USA, 2010.

[91] J. Sorva, "The same but different students' understandings of primitive and object variables," in *Koli '08 Proceedings of the 8th International Conference on Computing Education Research*, Koli, Finland, 2008.

[92] J. Carter and A. Fowler, "Object oriented students?," in *ITiCSE '98 Proceedings of the 6th annual conference on the teaching of computing and the 3rd annual conference on Integrating technology into computer science education: Changing the delivery of computer science education*, Dublin, Ireland, 1998.

[93] A. E. Fleury, "Programming in Java: student-constructed rules," in *SIGCSE '00 Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, Austin, TX, USA, 2000.