Poučavanje objektno orijentiranog programiranja metodom didaktičkog skrivanja

Krpan, Divna

PRIRODOSLOVNO-MATEMATIČKI FAKULTET

Divna Krpan

POUČAVANJE OBJEKTNO ORIJENTIRANOG PROGRAMIRANJA METODOM DIDAKTIČKOG SKRIVANJA

DOKTORSKI RAD

Split, 2020.

PRIRODOSLOVNO-MATEMATIČKI FAKULTET

Divna Krpan

POUČAVANJE OBJEKTNO ORIJENTIRANOG PROGRAMIRANJA METODOM DIDAKTIČKOG SKRIVANJA

DOKTORSKI RAD

MENTOR: izv. prof. dr. sc. Saša Mladenović

Split, 2020.

FACULTY OF SCIENCE

Divna Krpan

TEACHING OBJECT ORIENTED PROGRAMMING USING DIDACTIC REDUCTION

DOCTORAL THESIS

SUPERVISOR: Associate professor Saša Mladenović, Ph. D.

Split, 2020.

ZAHVALE

Prvenstveno, zahvaljujem svom mentoru, izv. prof. dr .sc. Saši Mladenoviću koji me je svojim iskustvom i savjetima vodio i usmjeravao na putu od prvih istraživanja pa sve do konačne izrade doktorske disertacije.

Hvala svim članovima povjerenstva: prof. dr. sc. Andrini Granić, prof. dr. sc. Maji Štuli, doc. dr. sc. Goranki Nogo, izv. prof. dr. sc. Krešimiru Pavlini, doc. dr. sc. Nikoli Maranguniću na trudu uloženom u vrednovanje ovog rada.

Zahvaljujem kolegama i kolegicama na Fakultetu, Odjelu za informatiku i svim prijateljima na podršci, a posebno izv. prof. dr. sc. Ivici Boljatu na povjerenju i motivaciji te kolegi dr. sc. Goranu Zahariji na nesebičnoj suradnji, komentarima, pomoći u istraživanju i nastavi.

Veliko hvala mojoj obitelji na potpori i razumijevanju…

TEMELJNA DOKUMENTACIJSKA KARTICA

Sveučilište u Splitu Doktorska disertacija
Prirodoslovno-matematički fakultet
Poslijediplomski sveučilišni studij
„Istraživanje u edukaciji u području prirodnih i tehničkih znanosti“

POUČAVANJE OBJEKTNO ORIJENTIRANOG PROGRAMIRANJA METODOM DIDAKTIČKOG SKRIVANJA

Divna Krpan

Prirodoslovno-matematički fakultet

Ruđera Boškovića 33, 21 000 Split, Hrvatska

Sažetak

Učenje i poučavanje programiranja je teško. Poteškoće se pojavljuju kod svih uzrasta, a na fakultetima se odražavaju u obliku slabe prolaznosti. Studenti u kratkom vremenu moraju usvojiti apstraktne koncepte iz objektno orijentiranog programiranja (OOP) uz složenu sintaksu programskog jezika i okruženje. Zabrinjava činjenica da i nakon položenih početnih predmeta studenti i dalje imaju poteškoće. Obzirom na vremensko ograničenje potrebno je smanjiti irelevantno kognitivno opterećenje studenata. Jedan od načina je odabir početnog jezika s jednostavnijom sintaksom. Tijekom trogodišnjeg pilot istraživanja kod uvođenja vizualnih programskih jezika u visoko obrazovanje utvrđeno je da nema značajnog utjecaja na uspjeh u predmetima s tekstualnim jezicima, ali su određeni zahtjevi prema okviru za poučavanje objektno orijentiranog programiranja. Okvir OTTER temeljen na metodi didaktičkog skrivanja implementiran je u nastavi te su prikupljeni studentski projekti izrađeni pomoću OTTER-a tijekom pet godina. Postavljen je cilj istraživanja: utvrditi utjecaj primjene metode didaktičkog skrivanja pomoću okvira OTTER na usvajanje koncepata OOP-a. Razvijen je poseban alat za analizu projekata i izdvajanje objektno orijentiranih koncepata definiranih u radu. Faktorskom analizom izdvojeni su koncepti koji su vezani za grafičko korisničko sučelje i koncepti koji su isključivo vezani za OOP. Rezultati su pokazali da su studenti tijekom izrade projekata pomoću OTTER-a primjenjivali više koncepata OOP-a u odnosi na one koji nisu koristili okvir. Analiza podataka pomoću strukturalnog modeliranja je potvrdila rezultate te omogućila dodatno pojašnjenje odnosa među varijablama. Dokazana je opravdanost poučavanja OOP-a kod početnika u visokom obrazovanju primjenom metode didaktičkog skrivanja. Okvir omogućuje brži prijelaz iz početne faze usvajanja sintakse do izrade projekata.

(198 stranica, 65 slika 10 tablica, 277 literaturnih navoda, jezik izvornika: hrvatski). Rad je pohranjen u Sveučilišnoj knjižnici u Splitu, Ruđera Boškovića 31, Split te u Nacionalnoj i sveučilišnoj knjižnici, Ul. Hrvatske bratske zajednice 4, Zagreb.

Ključne riječi: poučavanje programiranja; objektno orijentirano programiranje; didaktičko skrivanje; projektno učenje

Mentor: izv. prof. dr. sc. Saša Mladenović

Ocjenjivači:
1. prof. dr. sc. Andrina Granić
2. prof. dr. sc. Maja Štula
3. doc. dr. sc. Goranka Nogo
4. izv. prof. dr. sc. Krešimir Pavlina
5. doc. dr. sc. Nikola Marangunić

Rad prihvaćen: 09. siječnja 2020.

BASIC DOCUMENTATION CARD

University of Split Doctoral Thesis

Faculty of Science
Doctoral programme
“Education Research in Natural and Technical Sciences”

TEACHING OBJECT ORIENTED PROGRAMMING USING DIDACTIC REDUCTION

Divna Krpan

Faculty of Science

Ruđera Boškovića 33, 21 000 Split, Hrvatska

Abstract

Learning and teaching programming is difficult. Difficulties are present with students of different ages, and evident at the universities through poor pass rate. Students have to acquire abstract concepts of object-oriented programming with complex syntax and environment. We are concerned with the fact that students seem to have the same problems even after they pass introductory programming courses. Based on time constraints, it is important to reduce irrelevant cognitive load. One way to tackle it is through the use of simple introductory languages. During a three-year pilot study in introducing visual programming languages in higher education, we found there was no significant influence on students' achievement in textual programming language courses. However, we have determined the requirements for the framework used in teaching object-oriented programming. OTTER framework is based on didactic reduction and implemented in teaching. We have collected students' projects for five years. The main goal of the research was set: determine the influence of didactic reduction using OTTER on acquiring OOP concepts. An analysis tool used to detect object-oriented concepts defined in this paper was developed for the research. Factor analysis separated concepts based on the graphical user interface and OOP. Results showed that students who used OTTER also applied more OOP concepts than students who didn't use the framework. Structured equation modeling confirmed results and enabled a better understanding of relationships among variables. Appropriateness of teaching OOP to higher education novice programmers using didactic reduction was proved. The framework enables faster transfer from the initial syntax learning stage to project development.

(198 pages, 65 figures 10 tables, 277 references, original in Croatian)
Thesis is deposited in the University Library of Split, Ruđera Boškovića 31, Split and National and University Library, Ul. Hrvatske bratske zajednice 4, Zagreb

Keywords: teaching programming; object oriented programming; didactic reduction; project based learning

Supervisor: associate professor Saša Mladenović, Ph. D.

Reviewers:
1. professor Andrina Granić, Ph. D.
2. professor Maja Štula, Ph. D.
3. assistant professor Goranka Nogo, Ph. D.
4. associate professor Krešimir Pavlina, Ph. D.
5. assistant professor Nikola Marangunić, Ph. D.

Thesis accepted: 9th January 2020.

SADRŽAJ

POPIS SLIKA I TABLICA

Popis slika

Popis tablica

1 UVOD

Učenje i poučavanje programiranja je općenito teško te predstavlja izazov za nastavnike i učenike, bez obzira na uzrast učenika (osnovna škola, srednja škola ili fakultet) (Turville, Meredith, & Smith, 2012). Studenti na fakultetima često imaju poteškoće pri polaganju ispita iz predmeta vezanih za programiranje što možemo potvrditi višegodišnjim promatranjem i iskustvom rada sa studentima. Uočili smo poteškoće pri usvajanju osnovnih koncepata i algoritama, ali također i kod rješavanja problema. Studenti na Prirodoslovno-matematičkom fakultetu u Splitu (PMFST) koji upisuju predmete početnog programiranja razlikuju se po interesima, odnosno smjerovima koje su upisali (matematika, informatika, tehnika, fizika), a većina studenata se obrazuje za buduće nastavnike odgovarajućih predmeta. Ono što je zajedničko većini studenata je nedostatak prethodnog iskustva u učenju programiranja, odnosno često se događa da studenti nisu uopće imali doticaj s programiranjem u prethodnom obrazovanju. U vrijeme kad smo analizirali što naši studenti uče prije upisa na fakultet (osnovna i srednja škola), informatika je bila izborni predmet u osnovnoj školi (Krpan & Bilobrk, 2011). U srednjoj školi situacija nije bila puno bolja, osim u prirodoslovno-matematičkim smjerovima. Primjerice, u nekim gimnazijama su učenici imali informatiku samo tijekom prve godine, a obzirom da informatika nije samo programiranje, količina sadržaja koja se odnosi na programiranje je minimalna. Osim navedenog, može se očekivati da učenici koji imaju više iskustva iz informatike kasnije biraju inženjerske smjerove, a ne nastavničke, kao što je PMFST.

Studenti koji su odabrali smjer Informatika nakon završetka studija bi trebali biti osposobljeni za poučavanje učenika osnovne i srednje škole konceptima iz informatike, a za ostale smjerove se očekuje primjena stečenih znanja iz područja informatike u drugim područjima npr. kao alata za poučavanje koncepata i sl. Trenutno je aktualno integriranje četiri područja koja su obuhvaćena akronimom STEM (Science, Technology, Engineering, and Mathematics) (William F. McComas, 2014), a obuhvaća znanost, tehnologiju, inženjerstvo i matematiku. STEM je akronim koji stječe popularnost i u Hrvatskom jeziku. Ako se bar dvije ili više navedenih područja na neki način integrira u nastavi, onda se radi o integriranom STEM-u (David R. Heil, Burger, & Burger, 2013; William F. McComas, 2014). Jednostavan primjer je upotreba simulacija za učenje pojmova iz fizike, kemije, biologije ili učenje matematike uz pomoć računala i sl. Prema tome, informatika se integrira i u ostala područja.

Djeca su danas u školi i kod kuće veoma rano izložena računalima i tehnologiji općenito (Bers i ostali, 2002; Haugland, 2000; Seybert, 2011). Djeca u dvadeset prvom stoljeću se razlikuju od prethodnih generacija, te se često nazivaju "digitalni urođenici" (eng. “digital natives”) (Prensky, 2001) ili Net generacija (eng. "millenials") (Strauss & Howe, 2000). Takvi učenici zahtijevaju i različite metode poučavanja koje su u skladu s njihovim sposobnostima. Učenici su danas upoznati s korištenjem tehnologije te očekuju da će tehnologija biti sastavni dio njihovog poučavanja (M. Lister, 2015; Prensky, 2005). Postavlja se pitanje kako se u cijelu priču uklapa programiranje. Guido von Rossum, autor programskog jezika Python, smatra da je poučavanje programiranja važno kao i poučavanje čitanja, pisanja i matematike (Rossum, 1999). Učenici bi prema tome trebali učiti programiranje iako svi neće postati vrhunski programeri, no isto tako neće svi koji nauče pisati postati vrhunski pisci. Napretkom tehnologije i njenom prisutnosti u našim životima, svakako bi trebalo osigurati i veću razinu računalne pismenosti.

Nastavnici moraju biti u stanju koristiti tehnologiju u nastavi kako bi razumjeli što je primjereno, ali isto tako u skladu s razvojnom fazom učenika (Bers i ostali, 2002). Primjerice, ako prema razvojnim fazama Piaget-a (Piaget & Inhelder, 2013) djeca u dobi od 7-11 godina razvijaju sposobnosti logičkog razmišljanja i rješavanja problema onda ne možemo očekivati da će se te sposobnosti razviti mnogo ranije te u skladu s tim treba pripremati sadržaje koje učenici mogu usvojiti. Može se pročitati kako su današnje učionice pune načina poučavanja koji su „zastarjeli, dosadni i puni stvari iz prošlosti“ (Prensky, 2005), a današnji učenici su spretni u korištenju moderne tehnologije te očekuju da ih se zainteresira i uključi.

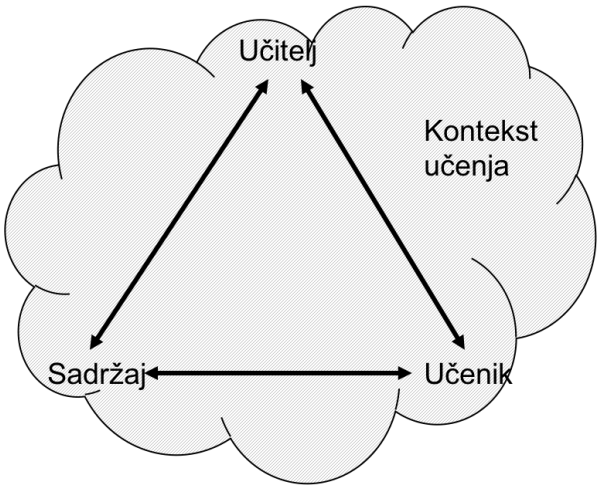
Programiranje se kao značajna stavka pojavljuje u sklopu informacijske ili računalne pismenosti (K. Williams, 2003), a računalna pismenost (ponekad: računalne vještine (eng. computational skills)) se po nekim autorima smatra bitnom za sve preddiplomske studije i cjeloživotno učenje (McAllister & Alexander, 2008).

2 TEORIJSKE OSNOVE

Prilikom pretraživanja literature, općenito se može steći dojam kako istraživači koriste vlastito iskustvo i intuiciju na temelju kojeg odlučuju uvesti inovacije za koje smatraju da bi mogle pomoći učenju i poučavanju programiranja premda nije uvijek jasno na kojoj teoriji se taj pristup temelji.

Postoji izreka da je "didaktika stara kao vrijeme" (Tchoshanov, 2013). Već je od razvoja društva postojala potreba za prijenosom znanja, vještina i iskustava na sljedeću generaciju. Međutim, tijekom vremena su se znanja, vještine i iskustva akumulirali te je postalo sve teže prenositi ih na iduće generacije pa su nastale škole (Poljak, 1990). S pojavom škola, definira se i uloga poučavatelja, odnosno učitelja kao nekoga čiji je posao prijenos znanja. U skladu s tim povijesnim razvojem, čini se da je najprije nastao sadržaj kojeg treba prenijeti budućim generacijama, a nakon toga je nastala potreba za osmišljavanjem učinkovitijih načina poučavanja. Od daleke 1613. godine ističe se potreba za vještinom poučavanja, a 1632. godine Komensky objavljuje svoje djelo: Velika didaktika.

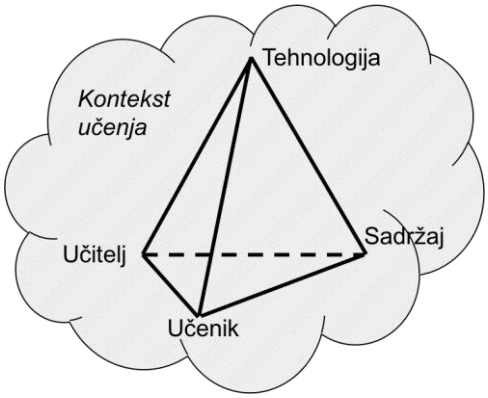
Trojka: učenik – sadržaj – učitelj je prisutna od početka prijenosa znanja, vještina i iskustava, ali se pojam didaktičkog trokuta pojavljuje tek 1980-tih godina (Kansanen & Meri, 1999). Premda bi se didaktički trokut trebao promatrati kao cjelina, to je u praksi nemoguće. Didaktički trokut se smješta u kontekst poučavanja (Slika 2.1).



Slika 2.1. Didaktički trokut

Informacijsko – komunikacijska tehnologija (IKT) u obrazovanje donosi revolucionarne promjene pa se u skladu s tim nastojalo istaknuti ulogu IKT-a kao posrednika između sadržaja, učitelja/nastavnika i učenika (Slika 2.2) stvarajući tetraedar. Svaka ploha tetraedra predstavlja poseban odnos pa je tako donja ploha tradicionalni didaktički trokut. Ploha "učenik - sadržaj -

tehnologija" može predstavljati uključivanje e-učenja, ali također i *obrnutu učionicu* (eng. flipped classroom) ili neki model samostalnog učenja u virtualnom okruženju. Pojam *e-poučavanje* (eng. e-teaching) predstavlja trojku "učitelj - sadržaj - tehnologija", a trojka "učitelj - učenik - tehnologija" prikazuje interakciju između učitelja i učenika van domene sadržaja. U ovom kontekstu IKT pripada sadržaju učenja i poučavanja.



Slika 2.2. Didaktički tetraedar (Tchoshanov, 2013)

Problemi kod poučavanja u svakom području su specifični te nije moguće odvojiti poučavanje od onog što se poučava. Nastavnik bez obzira na to što su neki principi poučavanja općeniti ne može poučavati bilo što tj. mora biti dobro upoznat sa sadržajem područja kojeg poučava i razumjeti specifične zahtjeve učenja i poučavanja u tom području. Nastavnici u visokom obrazovanju se godinama pripremaju i uče kako bi postali stručnjaci u svom području istraživanja, ali se pri tome često događa da nemaju iskustva u poučavanju istog iako im je poučavanje postalo dio svakodnevnih aktivnosti (Berthiaume, 2008).

Premda smo ranije spomenuli definiciju didaktičkog tetraedra koji uključuje tehnologiju kao zasebnu stavku, prema pretraživanju literature nije uočeno da se u području učenja i poučavanja programiranja tehnologija posebno izdvaja. Vjerojatno je razlog što su programiranje i tehnologija koja se koristi previše integrirani da ih nema smisla razdvajati. Prema (Bennedsen, 2008) sadržaj iz didaktičkog trokuta se odnosi na ciljeve učenja, vrednovanje postignuća i stvarni sadržaj koji se uči. Učitelj/nastavnik predstavlja proces učenja i poučavanja, te prilagodbu sadržaja (odabira i prikaza) u skladu s tim. Učenik predstavlja pretpostavke učenja i proces učenja (konkretno proces prijelaza iz jednog stanja u drugo).

Pedroni (2003) tvrdi da je za učenje i poučavanje programiranja važno definirati koga se poučava, što se poučava i kako (opet trokut). Međutim, stavljanje stvari u kontekst, odnosno predstavljanje problema u realnoj situaciji pomaže pri usvajanju koncepata iz programiranja. Zanimljivo je i da odgovarajući kontekst može omogućiti djeci usvajanje specifičnih znanja i

vještina čak ranije nego što bi prema trenutnoj razvojnoj fazi to trebali biti u stanju (Richardson, 1998). Važno je proučiti karakteristike programera početnika koje želimo poučavati kako bi za njih odabrali primjerene sadržaje i načine poučavanja.

2.1 Početnici u programiranju

Iskusni programeri imaju mnogo znanja, vještina, a logično i iskustva. Neke od tih vještina su više, a neke manje očite (Jenkins, 2002). Programer mora biti sposoban koristiti računalo, napisati program, testirati program, otkriti i ukloniti pogreške. To su očite vještine te također ono što često nastavnici traže od studenata početnika u programiranju. Međutim, programiranje predstavlja hijerarhiju vještina, a u situaciji kad učenik mora usvojiti nešto tako složeno, onda najprije kreće od najniže razine (Ng & Bereiter, 1991), a to u programiranju predstavlja dio koji se odnosi na kodiranje, odnosno učenje osnova sintakse kao prvog koraka te kasniji prijelaz na semantiku (Jenkins, 2002). Pojednostavljeni primjer iz prakse je kad početnici ne rade uvlake ili ne pišu komentare, ostavljajući to za kasnije. Ako to neće biti dio ocjene, onda to neće napraviti ni kasnije. Kad su studenti na PMFST učili QBASIC na početnom predmetu iz programiranja, često su izbjegavali raditi uvlake što je činilo programe teže čitljivima, a tek bi to počeli raditi na inzistiranje nastavnika i to nakon što bi provjerili da program radi. To su loše navike za programere, a na neki način se pokušavaju ispraviti jezicima kojima su uvlake bitne (Python) ili okruženjima u kojima se to radi automatski (npr. MS Visual Studio za C#).

Bennedsen & Caspersen (2007) su ispitivali prolaznost početnih predmeta iz programiranja širom svijeta. Premda mnogi tvrde kako je prolaznost općenito loša, jedni su od rijetkih koji su pokušali doći do kvantitativnih dokaza koji bi obuhvatili šire područje. Poslali su upitnike na brojne adrese, ali su dobili odgovor od samo 63 institucije u 15 država svijeta. Broj studenata upisanih na početne predmete je varirao od 8 do 645. Prema dostupnim podacima zaključili su kako trećina studenata ne prolazi početne predmete iz programiranja, manje grupe studenata imaju bolju prolaznost te da na prolaznost nije utjecao programski jezik koji se koristi za poučavanje.

Obzirom na ograničenja provedenog istraživanja nisu mogli generalizirati zaključke. Watson & Li (2014) su nastavili na njihovom tragu te su nakon analize nedostataka prethodnog istraživanja umjesto slanja upitnika proveli sistematsku analizu literature relevantnih znanstvenih radova vezanih uz temu. Rezultati iz izdvojenih radova su se odnosili na ukupno 161 predmet u razdoblju od 1979. do 2013. godine, odnosno 51 instituciju iz 15 različitih država. Prosječna prolaznost je bila 67% što je bilo u skladu s prethodnim istraživanjem.

Statističkom analizom prolaznosti tijekom vremena, zaključuju kako nije bilo značajnih promjena tijekom vremena, što znači da se zadnja tri desetljeća prolaznost nije bitno promijenila bez obzira na napredak IKT-a. Također, nije utvrđena statistički značajna razlika u odnosu na programski jezik koji se koristi na predmetima, ali je utvrđena pozitivna razlika u korist manjih grupa studenata. Watson & Li su se složili s prethodnim da prolaznost od 67% nije zabrinjavajuće niska razina uz napomenu da ipak treba promatrati širi kontekst kao što je opadanje broja upisanih studenata na predmete iz područja računarstva. Zanimljivo je napomenuti da je 49% kolegija iz istraživanja koje su proveli Bennedsen & Caspersen (2007) objektno orijentirano.

Perkins i ostali (1986) klasificiraju programere početnike u dvije kategorije:

- Zaustavljači (eng. stoppers) – ne žele istraživati te odustaju kod poteškoća,
- Pokretači (eng. movers) – ne staju kod poteškoća već stalno istražuju što mogu napraviti.

Podjela se temelji na akcijama početnika u trenutku kad ne znaju što treba dalje napraviti. Pri tome su ponekad *pokretači* učinkovitiji, ali i njima se može dogoditi da počnu lutati pri traženju rješenja. Tehnika koja je bliska pokretačima može se nazvati „petljanje“ ili „čeprkanje“ prema engleskom nazivu „tinkering“, a to zapravo znači da pokušavaju napisati dio koda u kojem rade sitne izmjene. Pokretači s takvim načinom rada mogu imati poteškoće pri praćenju kôda (eng. tracing) odnosno preciznom razumijevanju što program točno radi.

Tijekom 70-tih godina provedena su istraživanja u kojima su se uočile razlike između dva pristupa učenju: *dubinski* (eng. deep) i *površinski* (eng. surface) (Trigwell, Prosser, & Waterhouse, 1999). Pokazalo se da učenici koji uče površinski imaju slabije rezultate. Općenito, ako učenici procijene da će se od njih tražiti samo memoriranje ili smatraju da su sadržaji previše zahtjevni onda redovito pristupaju učenju površinski (Ng & Bereiter, 1991; Trigwell i ostali, 1999). Učenici se pri tome trude samo izvršiti zadatke koje je definirao učitelj. Površinsko učenje se češće javlja kod nastave orijentirane prema učitelju, dok se kod nastave orijentirane prema učeniku očekuje više aktivnosti učenika koji postaje aktivni subjekt u nastavi te se koristi dubinsko učenje. Proces učenja programiranja zahtjeva od studenata dodatne aktivnosti izvan učionice: čitanje dodatnih materijala, samostalnu izradu algoritama, otkrivanje i uklanjanje grešaka, rješavanje problema, traženje pomoći od svojih kolega i sl. Programiranje je vještina koja se ne može učiti površinski (Jenkins, 2002). Pamćenje sintakse ili pravila koje treba primijeniti te čak i uzoraka ili dijelova kôda koji se mogu upotrijebiti na prepoznatom

problemu mogu studenta dovesti do rješenja, ali to ipak spada u površinsko učenje. Za razvijanje stvarnih kompetencija ključni su elementi dubinskog učenja i razumijevanja.

Početnicima nedostaje vještina planiranja (Robins, Rountree, & Rountree, 2003), što se vidi u tome kad počnu odmah pisati programski kôd bez prethodno definiranog plana rješenja problema. Pri tome često dolazi do grešaka zbog nepoznavanja jezika, te studenti više nisu u stanju razlikovati iz kojeg razloga njihovo rješenje radi ili ne radi. Problem u ovom slučaju može biti i kad se nastavnik više koncentrira na probleme sintakse umjesto poučavanja složenijeg procesa izrade algoritma (Jenkins, 2002). Ponekad ni sami početnici nisu svjesni svojih nedostataka (Ala-Mutka, 2004; McCracken i ostali, 2001). Trebali bi naučiti i kako procijeniti svoju uspješnost, te u kojem trenutku i kako zatražiti pomoć od učitelja/nastavnika ili kolega.

Medeiros i ostali (2019) su napravili pregled literature kako bi identificirali poteškoće u poučavanju programiranja u visokom obrazovanju koje se spominju u radovima. Njihov rad se dosta oslanja na pregled kojeg su napravili Vihavainen i ostali (Vihavainen, Airaksinen, & Watson, 2014) iako se popis baza koje su istražili razlikuje. Na temelju 89 izdvojenih radova Medeiros i ostali izdvajaju dvije kategorije preduvjeta za bolje učenje programiranja u visokom obrazovanju: preduvjeti vezani za programiranje i opći obrazovni preduvjeti. U prvoj kategoriji se ističu matematičke sposobnosti i rješavanje problema, apstrakcija te predznanje iz programiranja. Pod kategoriju općih obrazovnih predznanja svrstavaju znanje engleskog jezika (jer programski jezici često imaju sličnosti s engleskim jezikom), sposobnosti kritičkog razmišljanja, kreativnost i upravljanje vremenom (eng. time management). Nadalje zaključuju kako studenti često imaju problema s formuliranjem problema, izražavanjem tog problema pomoću programskih struktura (kao što su kontrolne strukture, rekurzije, funkcije, klase i sl.), izvršavanje i vrednovanje rješenja (najteži dijelovi ovdje su uklanjanje pogrešaka i praćenje izvršavanja kôda) te karakteristike i ponašanje studenata (angažman, motivacija, samouvjerenost). U vezi studenata ističu kako se često spominje da studenti učenju programiranja posvećuju daleko manje vremena od preporučenog. J. Kramer (2007) u svojem osvrtu na apstrakciju u računarstvu ističe kako je apstrakcija zapravo temelj za matematiku i inženjerstvo. S druge strane, Devlin (2001) ističe kako je u računarstvu zapravo sve apstrakcija (svaki koncept, metoda i sl.) te smatra kako je matematika jedino „mjesto“ gdje se u školi i na fakultetima može steći i razvijati apstraktne sposobnosti te da su upravo te apstraktne sposobnosti koje učenici razvijaju na matematici ključne za pisanje dobrih programa.

Promatrajući aspekt nastavnika, Medeiros i ostali (2019) također zaključuju kako se nastavnici suočavaju s izazovima odabira strategije poučavanja, motivacijom i angažmanom studenata, problemima razvijanja vještina programiranja kod previše različitih profila studenata istovremeno, komunikacijom sa studentima te odabirom programskog jezika. Većina okruženja koja pomažu početnicima pri lakšoj formulaciji problema jesu alati koji su namijenjeni djeci (npr. Scratch, Alice i sl.) te kao takvi općenito nisu prenosivi na razinu visokog obrazovanja. Sami studenti takve alate smatraju djetinjastima. Medeiros i ostali zaključuju kako bi se trebalo razmisliti o izradi sličnih alata za visoko obrazovanje jer takvih stvari očito nedostaje. Vihavainen i ostali (2014) su uspoređivali prolaznost prije i poslije neke intervencije, a prema rezultatima u njihovom pregledu literature zaključuju kako se prolaznost s različitim intervencijama popravila za trećinu. Nedostatak analize je što su posebno izdvojili samo 32 rada tako da ne možemo po tome generalizirati. Možemo se složiti s autorima u primjedbi kako istraživači kojima neka intervencija nije uspjela rijetko objavljuju rezultate tako da ne znamo puno o neuspjelim intervencijama. Također vrijedi ponovo spomenuti zaključak od Watson & Li (2014) gdje se ističe kako se tijekom vremena prolaznost previše ne mijenja bez obzira na intervencije. Način ocjenjivanja kod različitih institucija i država nije usporediv. Ocjenjivanje u okviru istog predmeta na istoj instituciji može se razlikovati po akademskim godinama (npr. relativno ocjenjivanje). Obzirom na navedeno, nameće se ideja kako bi bilo poželjno ispitati broj koncepata koje studenti usvajaju. Možda nakon neke intervencije prolaznost ostaje ista, no je li i broj koncepata isti? Neki studenti se koncentriraju na prolaznu ocjenu, ali ako se ljestvica podigne, onda moraju naučiti više. Posljedica je da studenti moraju usvojiti više koncepata, no prema iskustvu sa studentima na PMFST-u, prolaznost će opet ostati u približno istim okvirima kao i prosječne ocjene.

Kori i ostali (2016) su ispitivali motivaciju studenata ovisno o tome jesu li prije upisa na fakultet imali iskustva u programiranju. Studenti koji su odlučili učiti programiranje bez prethodnog iskustva su bili više motivirani vanjskim utjecajima (moguće plaćom ili mogućnostima zaposlenja) u odnosu na studente koji su imali iskustva. Studenti s više prethodnog iskustva su kasnije ostvarili bolje rezultate, ali je kod njih također postojala opasnost zanemarivanja studija zbog zapošljavanja za vrijeme studiranja. Učenici koji su motivirani i koriste odgovarajuće strategije učenja, ostvaruju bolje rezultate. Kao posljedica toga, takvi učenici će steći navike koje će im koristiti kod cjeloživotnog učenja. Pomicanje fokusa s učitelja na učenika utječe na nastavu koja se više zasniva na izradi projekata i međusobnoj suradnji učenika. Smatra se da unutarnja motivacija za učenjem predstavlja instrument za optimalno učenje i poučavanje te

veću kreativnost. Jenkins (2002) ističe kako iz činjenice da studenti imaju poteškoće s programiranjem mora slijediti i razlog zašto je to tako te da bi trebalo obratiti pozornost na literaturu iz područja psihologije i kognitivne znanosti. Na primjer, ograničeno vrijeme u kojem studenti moraju naučiti kako formulirati problem u nekom programskom jeziku predstavlja veliko kognitivno opterećenje za studente (Medeiros i ostali, 2019).

2.1.1 Teorija kognitivnog opterećenja

Obzirom da je programiranje aktivnost koja uključuje kognitivne sposobnosti učenika, potrebno je uzeti u obzir i kognitivne faktore, a jedan od tih faktora koje ćemo ovdje promatrati je kognitivno opterećenje učenika.

Teorija kognitivnog opterećenja (eng. cognitive load theory, CLT) (Sweller, Ayres, & Kalyuga, 2011) je kognitivistička teorija učenja koju je 1980tih godina predstavio J. Sweller, a nastala je na temelju zapažanja kako ljudski mozak može istovremeno raditi s ograničenom količinom informacija. Prema CLT smatra se da višak informacija može ometati učenika, te da prisutnost elemenata koji se ne koriste za rješavanje trenutnog problem stvara poteškoće (Raina Mason & Cooper, 2013).

Jedna od važnih pretpostavki CLT je podjela kognitivne arhitekture čovjeka na dva dijela (Sweller i ostali, 2011):

1. radna memorija ili kratkotrajna memorija (eng. short-term memory),
2. dugoročna memorija ili dugoročno pamćenje.

Radna memorija je mjesto gdje se odvija kognitivna obrada informacija kao što je na primjer razmišljanje. U radnoj memoriji se informacije privremeno zadržavaju, no glavni problem je što je ona ograničenog kapaciteta. Prema (Miller, 1956) količina informacija, odnosno broj „elemenata“ ili „komada“ (eng. chunks) informacija koje istovremeno možemo držati u radnoj memoriji je u rasponu od pet do devet. Posljedica toga je što se nove informacije u slučaju preopterećenja mogu lako izgubiti. Cilj je informacije iz radne memorije spremiti u dugoročnu. Dugoročna memorija ima praktički neograničen kapacitet, a u radnoj memoriji možemo neograničeno pozivati informacije koje su prethodno već bile spremljene u dugoročnoj. CLT se koristi pri analizi kognitivnog opterećenja pri učenju programiranja (Caspersen & Bennedsen, 2007; Raina Mason & Cooper, 2012).

Pojedini dijelovi sadržaja ili informacije koje treba usvojiti nazivaju se *elementi* (eng. elements) (Sweller, 1994). *Shema* (eng. schema) je kognitivna konstrukcija koja služi za organizaciju

elemenata u osnovnu jedinicu znanja. Proces stvaranja sheme podrazumijeva organizaciju elemenata informacija te njihovo povezivanje i spremanje u dugoročnu memoriju. Nakon što je shema spremljena u dugoročnu memoriju, možemo je po potrebi pozvati opet u radnu. Znanje koje učenik ima o nekom području organizirano je u sheme koje također utječu na rad s novim informacijama.

Kognitivno opterećenje možemo podijeliti u tri kategorije (Raina Mason & Cooper, 2012; Paas, Renkl, & Sweller, 2004):

1. intrinzično kognitivno opterećenje (eng. intrinsic cognitive load),
2. irelevantno kognitivno opterećenje (eng. extraneous cognitive load),
3. povezano kognitivno opterećenje (eng. germane).

Intrinzično kognitivno opterećenje karakterizira *interaktivnost* (eng. interactivity) pojedinih elemenata (Sweller, 1994). Na primjer, ako se elementi koje treba naučiti mogu razumjeti pojedinačno, odnosno neovisno o drugim elementima iz tog nastavnog sadržaja, onda se smatra da je interaktivnost sadržaja niska. Na primjer, ključne riječi iz programskog jezika se mogu naučiti neovisno kao i riječi stranog jezika. Međutim, ako se elementi nastavnog sadržaja ne mogu razumjeti dok se njihove interakcije ne obrade u radnoj memoriji istovremeno, onda smatra da je interaktivnost tog sadržaja visoka. Na primjer, ako želimo naučiti govoriti strani jezik, moramo voditi računa o gramatici, različitim značenjima riječi, izgovoru i sl. U programskom jeziku moramo voditi računa o značenju naredbi.

Irelevantno kognitivno opterećenje (Paas i ostali, 2004) je nepotrebno opterećenje jer ne pridonosi konstrukciji sheme, ne uzima u obzir ograničenja čovjekove memorije, a mogu ga uzrokovati neodgovarajuće metode poučavanja. Ako se interaktivnost nekog elementa može smanjiti bez promjene onog što je naučeno onda je opterećenje irelevantno (Sweller, 1994).

Povezano kognitivno opterećenje je izravno povezano s procesom konstrukcije sheme, odnosno izazvano je aktivnim naporom učenika pri stvaranju nove sheme (Paas i ostali, 2004). Ova vrsta opterećenja je korisna za učenje te je tipična strategija poučavanja smanjiti irelevantno opterećenje kako bi se resursi mogli preusmjeriti na povezivanje i stvaranje sheme (Raina Mason & Cooper, 2012).

Prema (van Merriënboer & Sweller, 2005) kod složenog učenja učenici moraju biti sposobni raditi s ogromnim brojem interaktivnih elemenata. U kognitivnim domenama to se odnosi na veliki broj interaktivnih struktura znanja koje se istovremeno moraju obraditi u radnoj memoriji

kako bi ih učenici mogli razumjeti, a u domenama koje se odnose na vještine radi se o velikom broju vještina koje se moraju koordinirati u radnoj memoriji kako bi se izvodile ispravno.

2.1.2 Odrasli početnici u programiranju

Sposobnosti i karakteristike učenika ovise o njihovoj dobi. Teorija učenja koja se bavi poučavanjem odraslih učenika postoji od 1968. godine, naziva se andragogija, a autor je M. S. Knowles (Knowles, 1980; Merriam, 2001; Smith, 2002). Prije toga se dosta vremena posvetilo isključivo poučavanju djece. Knowles je istaknuo kako se način učenja odraslih osoba prilično razlikuje u odnosu na djecu. Odrasli učenici su samostalniji (eng. self-directed) i motivirani iznutra. Prema (Knowles, 1980; Knowles, Holton, & Swanson, 2005; Merriam, 2001) postoji šest principa andragogije:

1. potreba za znanjem (eng. need to know),
2. spoznaja o sebi (kao učeniku) (eng. self-concept),
3. prethodno iskustvo,
4. spremnost za učenje (eng. readiness to learn),
5. usmjerenost na učenje (eng. orientation to learning),
6. motivacija.

Prije nego što odrasli učenici počnu nešto učiti, moraju znati što će dobiti ako nešto nauče, odnosno koje su negativne posljedice ako ne nauče. Ukratko, moraju znati zašto nešto uče.

Odrasli učenici imaju izgrađenu spoznaju o sebi (eng. self-concept), smatraju se odgovornima za svoj život i učenje, imaju potrebu da ih drugi vide takvima te ne podnose ili zamjeraju kad im netko drugi nameće svoju volju. Veliki problem tu predstavlja trenutak kad se studenti na prvim godinama studija susreću s aktivnostima koje spadaju pod „obrazovanje“ jer se pri tome često vraćaju u zavisnu fazu iz prethodnog obrazovanja (osnovna i srednja škola) gdje zauzimaju stav „nauči me“ (Knowles i ostali, 2005). Reakcija nastavnika na to je da će studente tretirati na odgovarajući način no stav „zavisnosti“ je tipičan za djecu, a tretiranje odraslih učenika kao djece ima za posljedicu stvaranje unutarnjeg konflikta s psihološkom potrebom odraslih učenika da budu nezavisni i *samostalni* (eng. self-directed) učenici. Tipičan mehanizam rješavanja takvog psihološkog konflikta je pokušaj bijega iz te situacije, a to je također jedan od razloga odustajanja od studija na prvim godinama.

Odrasli učenici imaju određeno iskustvo, jednostavno iz razloga što su imali priliku živjeli duže pa su mogli akumulirati više iskustva od djece. Studenti se po iskustvu mogu prilično razlikovati, tako da ova pretpostavka može značajno utjecati na primjenu individualizirane

nastave. Dodatno je važno napomenuti kako djeca svoj identitet razvijaju od vanjskih utjecaja npr. bitnih osoba u životu (roditelji, članovi obitelji i ostali) i okruženja u kojem žive dok odrastanjem počinju definirati sami sebe u odnosu na vlastita iskustva. Ako se iskustvo odraslih učenika ignorira ili omalovažava oni to mogu percipirati kao odbacivanje njih samih kao osoba što može stvoriti problem kod obrazovanja odraslih.

Spremnost na učenje (eng. readiness to learn) znači da odrasli učenici postaju spremni za učenje stvari koje moraju znati kako bi se mogli nositi sa stvarnim situacijama u životu. To može biti u izravnoj vezi s razvojnim fazama u životu. Na primjer, studenti nisu spremni za učenje metodike nastave informatike dok nisu sami savladali znanja i vještine koje će poučavati, a još više će biti spremni za to kad počnu raditi kao nastavnici u školi jer će im se tada javiti stvarna potreba. Međutim, nije potrebno čekati dok se dogodi neka razvojna faza već ih se može staviti u situaciju ili simulaciju stvarne situacije kako bi potakli spremnost, a vezano uz prethodni primjer, tome može poslužiti održavanje pripremnih satova s djecom u okviru predmeta vezanih za metodiku kako bi se potaklo razvijanje spremnosti za stvarni rad u školi.

Djeca su u školi usmjerena na učenje konkretnih predmeta, a odrasli više na zadatke, probleme, odnosno situacije iz stvarnog života u kojima bi im naučeni sadržaji mogli pomoći. Odrasli bolje uče kroz iskustvo i ako sadržaj učenja za njih ima neposrednu vrijednost. Važno je da su sadržaji koji uče u kontekstu situacija iz stvarnog života.

Odrasli učenici reagiraju na vanjsku motivaciju (dobar posao, napredovanje, plaća i sl.), ali unutarnja motivacija je učinkovitija (samopouzdanje, kvaliteta života i sl.). Odrasli imaju unutarnju motivaciju za rastom i razvojem, ali se to često blokira negativnom spoznajom o sebi kao studentu, vremenskim ograničenjima i nastavnim programima koji krše principe obrazovanja odraslih.

Pri izravnom radu sa studentima uočavamo kako je dio studenata teško motivirati, brzo gube interes i motivaciju, uče površno s ciljem zadovoljavanja kriterija polaganja ispita koje je postavio nastavnik, ne žele aktivno sudjelovati i sl.

Studenti koji pri upisu na fakultete vjeruju kako su njihove sposobnosti iznad prosjeka često nisu ni svjesni kako zapravo nisu dovoljno spremni za studiranje tako da im zapravo treba puno više pomoći ili podrške (Hesser & Gregory, 2015). Hesser & Gregory nakon 14 tjedana rada sa studentima zaključili su kako studenti početnici nisu u stanju procijeniti svoje sposobnosti, ali čak i u situacijama kad prepoznaju da im je potrebna dodatna pomoć nisu je skloni tražiti. Na našem fakultetu smo prepoznali studente koji nisu dovoljno pripremljeni za programiranje u

prethodnom obrazovanju, no prilikom omogućavanja upisa takvim studentima zapravo institucija na neki način preuzima i odgovornost da će im izaći u susret.

Programiranje je ključna vještina u računarstvu, a poučavanje programiranja teško (McAllister & Alexander, 2008). Teško je identificirati koje su to kognitivne poteškoće koje stvaraju probleme kod učenja, ali i vještine koje čine dobrog programera. Čini se kao da se za poučavanje programiranja ulaže više vremena nego kod ostalih područja, a učenicima je unatoč tome i dalje teško. Pojedini autori tvrde da neke osobe jednostavno ne mogu naučiti programirati dok drugi smatraju da bi osobe koje imaju poteškoće ipak mogli usvojiti osnovna znanja i vještine programiranja iako neće postati vrsni programeri ili stručnjaci (Caspersen, 2007). Činjenica je i da učenje programiranja zahtijeva puno vremena i puno vježbanja (McAllister & Alexander, 2008). Takvo što se ne uklapa u predavački način rada na fakultetu koji traje jedan semestar. *Predavački* (eng. lecture-based) način rada je često prisutan na fakultetima (Shreeve, 2008).

Obzirom da je predmet našeg istraživanja rad sa studentima, jasno je da studenti kao odrasli učenici ne smiju biti pasivni promatrači kojima se samo prenose informacije, te je od iznimne važnosti njihovo aktivno sudjelovanje. Aktivno sudjelovanje zahtijeva motivaciju. Unatoč velikom broju studenata koji odustaje od programerskih predmeta, takvi predmeti u principu i dalje imaju veliki broj studenata. Takav je slučaj i na našem fakultetu, pogotovo na predmetima početnog programiranja. Grupe su često velike zbog ograničenja opreme i prostora, posebno tijekom predavanja (100 studenata u predavaonici je bitno drugačije od 10 studenata). U takvim situacijama za vrijeme nastave nije moguće dovoljno aktivirati studente, ali bi se predavački način rada bar na vježbama trebao izbjegavati kad god je to moguće (Bennedsen, 2008).

Motivacija se spominje u raznim kontekstima, no kod projektne nastave je od iznimne važnosti (posebno unutarnja ili intrinzična). U skladu s konstruktivizmom nastala je *teorija uključivanja* (eng. engagement theory) koja nije izravna posljedica nekog teorijskog okvira za učenje već je nastala na temelju iskustva u poučavanju uz pomoć tehnologije, a predstavlja se kao model za učenje (iako može i bez tehnologije) (Kearsley & Shneiderman, 1998). Smatra se da aktivnosti učenika uključuju aktivne kognitivne procese (stvaranje, rješavanje problema, razmišljanje, donošenje odluka i vrednovanje). Naglasak je na suradničkom radu, projektnim zadacima i fokusu na stvarni svijet.

2.2 Programiranje kao sadržaj poučavanja

Svakodnevno dolaze novi alati i okruženja koji mijenjaju poimanje oblikovanja programa i kodiranja. Isto tako, definicija pojma „programiranje“ se tijekom povijesti mijenjala (Blackwell, 2002). U početku je pojam programiranja bio više vezan uz računanje, a kasnije ulazi i u kognitivnu domenu gdje se programiranje opisuje kao ljudska aktivnost koja uključuje oblikovanje ponašanja računala koje pomaže ljudima ili umjesto njih obavlja intelektualne zadatke (Hoc, Green, Samurcay, & Gilmore, 1990). Na pitanje što je program, možemo odgovoriti sintaksno: program je tekst koji je konstruiran prema određenim gramatičkim pravilima (Pair, 1990). Takva definicija kod poučavanja stavlja naglasak na sintaksu. To je najniža razina, ali nema smisla poznavati sintaksu jezika ako ne znamo značenje rečenica, odnosno semantiku. Myers (1990) definira računalni program kao skup naredbi kojima možemo upravljati ponašanjem računalnog sustava, a programiranje je stvaranje programa. Ormerod (1990) definira programiranje kao vještinu rješavanja problema gdje je početno stanje problem za koji je potrebno izraditi program, a ciljno stanje je program i rješenje kojeg program računa. Operatori koji vode od početnog stanja do cilja su sintaksna i semantička obilježja jezika, kao i kognitivne osobine programera.

Početnici i stručnjaci u programiranju razlikuju se u različitim aspektima rješavanja problema: kako prikazuju problem, koje strategije biraju za rješavanje problema te po načinu organizacije znanja o domeni problema (Ormerod, 1990). Početnici pišu programe liniju po liniju dok iskusni programeri rastavljaju cilj na manje dijelove. Iskusni programeri spremaju veće količine informacija koje su bolje organizirane, a organizacija pomaže bržem prepoznavanju tipa problema kao i odgovarajućih strategija rješavanja (Petre, 1990). To u programiranju znači da će iskusni programeri predstavljati program u terminima semantičke strukture dok ih početnici predstavljaju sintaktički. Također, iskusni programeri će razvrstati probleme prema apstraktnim osobinama, dok se početnici oslanjaju na površne osobine. Tako na primjer početnici grupiraju programe prema karakteristikama kao što je područje ili primjeri primjene konkretnog programa, dok će iskusni programeri grupirati programe prema algoritmima koji se koriste za rješavanje problema (Ormerod, 1990).

Kod poučavanja programiranja često se događa da sadržaj poučavanja poistovjećujemo s programskim jezikom koji se koristi za poučavanje. Na primjer, ako pitamo studenta što uči na nekom predmetu iz programiranja ili nastavnika što poučava, vjerojatno će odgovor s obje strane biti naziv programskog jezika koji se na tom predmetu koristi. Međutim, programski jezik ne bi trebao biti u fokusu odnosno učenje programiranja ne bi trebalo biti vođeno

tehničkim karakteristikama jezika već je potrebno usvojiti općenite koncepte programiranja (Bennedsen, 2008). Programski jezik koji se koristi za rješavanje problema ima slab utjecaj na konačno rješenje kod iskusnih programera jer oni stvaraju apstraktni model rješenja koji nije blisko vezan s programskim jezikom (Petre, 1990). Čak se događa da prilikom rješavanja problema iskusni programeri skiciraju svoje rješenje u nekom svom pseudojeziku bez opterećenja sa sintaksom u kojoj će to pisati. Pri tome koriste elemente i simbole iz poznatih programskih jezika, prirodnog jezika i različitih drugih notacija (npr. matematika). Nakon oblikovanja rješenja ili skice u vlastitoj notaciji, pisanje u odabranom programskom jeziku je na kraju samo stvar prijevoda. Također su skloniji promjeni jezika nego promjeni algoritma tako da algoritam mijenjaju tek kad naiđu na veće probleme u drugom jeziku.

Učenici u različitim kontekstima uče koncepte. Sposobnost prijenosa naučenih koncepata programiranja u drugi programski jezik pokazuje dublje razumijevanje (Dann, Cosgrove, Slater, Culyba, & Cooper, 2012; Mladenović, Krpan, & Mladenović, 2016). Konceptima općenito smatramo skupove objekata, simbola ili događaja koji imaju zajedničke karakteristike ili kritične atribute, a učenicima služe za organizaciju znanja (Hunt, Wiseman, & Touzel, 2009). Koncept se može opisati kao mentalna konstrukcija ili reprezentacija kategorije koja omogućava klasifikaciju odnosno prepoznavanje primjeraka koji pripadaju ili ne pripadaju toj kategoriji (J. Bruner, Goodnow, & Austin, 2017; Hunt i ostali, 2009; Schunk, 2012). *Učenje koncepata* (eng. concept learning) odnosi se na formiranje reprezentacija s identificiranim atributima, poopćavanje na nove primjerke i diskriminaciju primjeraka koji pripadaju ili ne pripadaju kategoriji (Schunk, 2012). Pri tome koncepti mogu biti neki konkretni objekti (npr. stol, računalo, pas, mačka, …) ili apstraktni (npr. sreća, smisao, …). Učenici stvaraju pravila o konceptima kako bi ih mogli razvrstati. Npr. „ako ima četiri noge, brkove, rep, krzno, mijauče“ onda je mačka. Učenici brže usvajaju koncepte ili stvaraju pravila o konceptima ako im se prezentiraju pozitivni primjeri (ono što stvarno pripada kategoriji). Usvajanje koncepata odvija se u četiri razine (Schunk, 2012): konkretna razina, razina identiteta, razina klasifikacije i formalna razina. Usvajanje na konkretnoj razini znači da učenik prepoznaje koncept kojeg već poznaje, ali u istom kontekstu. Kod razine identiteta, koncept se prepoznaje iz različitih perspektiva (poopćavanje). Razina klasifikacije zahtijeva da učenik prepozna najmanje dva koncepta kao ekvivalentne. Učenik će obično moći imenovati koncepte tek na formalnoj razini te će na toj razini moći pobrojati njegove atribute, dati definiciju i razlikovati primjerke. Nastavnik tijekom poučavanja koncepata najprije imenuje koncept, zatim ga definira, opisuje bitne atribute, daje primjere što jest koncept te što nije. Na sličan način su organizirane i vježbe

na predmetu Objektno orijentirano programiranje gdje na primjer nastavnik imenuje koncept klase, definira što je to klasa, što čini klasu, pokazuje primjere u okruženju u kojem će studenti raditi vježbu te što nije klasa u zadacima koje rješavaju. Kako vježbe postaju složenije, uvode se složeniji koncepti, uključujući i zdvajanje primjera projekata i karakteristika dobrih i loših projekata.

Primjeri koncepata iz programiranja su osnovne algoritamske strukture ili koncepti kontrolnih struktura: slijed, grananje i ponavljanje, a s te tri strukture se mogu izraziti ostali algoritmi (Böhm & Jacopini, 1966). Osnovna karakteristika kontrolnih struktura grananja i ponavljanja je što remete linearnost programskog teksta (Rogalski & Samurcay, 1990). Početnici imaju problema s razumijevanjem i usvajanjem takvih koncepata. Jedan dio problema odnosi se i na logičke uvjete koji su sastavni dio takvih naredbi, a tu će početnici s boljim matematičkim znanjem imati manje problema. Problem s ponavljanjem ili petljom uključuje identificiranje akcija koje se moraju ponoviti i uvjeta koji utječe na ponavljanje ili zaustavljanje. Tu dodatnu komplikaciju donosi i koncept varijabli koje se mijenjaju unutar petlje. Varijable predstavljaju unos korisnika (vanjske), ali i međurezultate (unutarnje varijable) koje kontrolira programer. Tu je glavni izvor pogrešaka kod početnika činjenica da često te unutarnje varijable ili njihova stanja nisu dio rješenja „na papiru“. Na primjer, ako se dio naredbi mora ponoviti *n* puta, a koristi se petlja *while* onda korisnici često unutar tijela petlje zaborave povećati vrijednost varijable koja služi kao brojač koraka. U vizualnom programskom jeziku Scratch postoji petlja „ponavljaj *n* puta“ no takve petlje nema u Pythonu ili C#. U tim jezicima je potrebno voditi računa o varijabli koja broji korake petlje. Rogalski & Samurcay (1990) ističu kako je koncept varijable koji se pojavljuje u petlji zapravo koncept više razine.

Bayman & Mayer (1988) su predložili tri kategorije znanja iz programiranja:

1. Znanje sintakse – znanje o elementima programskog jezika i pravilima upotrebe,
2. Konceptualno znanje – razumijevanje konstrukata i principa programiranja,
3. Strateško znanje – upotreba znanja sintakse i konceptualnog znanja za rješavanje problema (ili vještine rješavanja problema koje su specifične za jezik).

McGill & Volet (1997) su u skladu s kognitivnom psihologijom na gore navedeno dodali dimenziju znanja (deklarativno, proceduralno i kondicionalno (metakognitivno)) i definirali konceptualni okvir. Na primjer, strateško i kondicionalno znanje znači da osoba može koristiti sintaksno i konceptualno znanje na primjeren način u novim situacijama i zadacima.

Podjela prema (McGill & Volet, 1997) uz napomenu da su se fokusirali na integraciju dviju nižih kategorija u četiri nove kategorije:

1. Deklarativno – sintaksno (sintaksa jezika),
2. Deklarativno – konceptualno (objašnjenje semantike i dijelova pseudokoda),
3. Proceduralno – sintaksno (sposobnost primjene pravila),
4. Proceduralno – konceptualno (sposobnost izrade rješenja).

Ovo pomalo podsjeća na popularnu Bloom-ovu taksonomiju (Bloom, 1956): znanje, razumijevanje, primjena, analiza, sinteza, vrednovanje. Revidirana taksonomija (Anderson i ostali, 2001) je zapravo dvodimenzionalna matrica koja se sastoji od kognitivne dimenzije i dimenzije znanja. Kognitivna dimenzija odnosi se na: prisjećanje, razumijevanje, primjenu, analizu, vrednovanje i stvaranje.

Dimenzija znanja (Anderson i ostali, 2001):

1. Činjenično znanje – deklarativno, npr. simboli, sintaksa i sl.,
2. Konceptualno znanje – principi, npr. semantika, pravila, koncepti,
3. Proceduralno znanje – znanje kako obaviti zadatak, npr. algoritmi, procesi,
4. Metakognitivno znanje – strategija visoke razine ili kognitivni proces učenja.

Robins i ostali (2003) su kategorizirali praktične probleme programera početnika u dvije kategorije: problemi vezani za proces rješavanja problema (oblikovanje, generiranje, vrednovanje) i problemi vezani za karakteristike programera početnika (znanje, strategije, mentalni modeli).

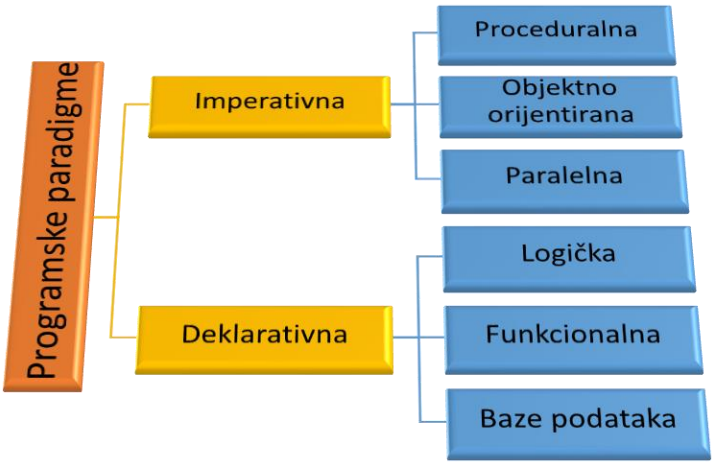
Pojednostavljeno, sadržaj kod učenja i poučavanja početnog programiranja odnosi se na usvajanje: koncepata programiranja (neovisno o jeziku, ali u skladu s odabranom programskom paradigmom), elemenata programskog jezika (sintaksa i primjena) i razvojnog okruženja. Osnovni koncepti programiranja kao što su na primjer slijed, grananje i ponavljanje mogu biti problematični za početnike. Događa se da početnici kažu kako su shvatili primjer koji su obradili na nastavi, ali ipak ne uspijevaju primijeniti koncepte u novoj situaciji. Poučavanje koje se fokusira isključivo na sintaksu programskog jezika neće omogućiti usvajanje vještina koje čine dobrog programera, a često udžbenici promoviraju upravo takav način (Linn & Clancy, 1992). Razvojna okruženja profesionalnih programskih jezika su prilagođeni iskusnim programerima tako da čak i alati za otkrivanje i uklanjanje pogrešaka koji bi trebali pomoći zapravo dodatno zbunjuju početnike (Ziegler & Crews, 1999).

Postoji više programskih jezika nego paradigmi tako da je bi najprije trebalo obratiti pozornost na paradigmu. Na primjer, ako gledamo sa strane paradigmi onda su programski jezici Java i C# u principu jednaki jer oba implementiraju objektno orijentiranu paradigmu. Prema tome možemo zaključiti da bi bilo jednostavnije promijeniti programski jezik unutar jedne paradigme nego promijeniti paradigmu. Svaki programski jezik temelji se na jednoj ili više programskih paradigmi (Samuel, 2017).

2.2.1 Programske paradigme

Pojam paradigmi u programiranju spominje se još od Floyd-ovog predavanja 1978. godine (Floyd, 1979). Za Floyd-a je u programiranju fokus više na rješavanju problema nego na programskom jeziku. Slično tome Shriver (1986) ističe kako nema precizne definicije paradigme te je za njega paradigma u programiranju model ili pristup rješavanju problema. Van Roy (2009) definira programsku paradigmu kao pristup programiranju koji se temelji na matematičkoj teoriji ili skupu principa. Nešto jednostavnijim riječima, Bobrow (1985) definira programsku paradigmu kao stil programiranja koji omogućuje programeru izraziti ono što želi. Također Van Roy (2009) smatra kako bi programski jezici morali podržavati više paradigmi jer je za različite stvari potrebno koristiti različite skupove koncepata, a Stroustrup (1988) ističe kako jezik *podržava* (eng. supports) paradigmu ili stil programiranja ako sadrži elemente koji primjenu paradigme čine relativno jednostavnom, sigurnom i učinkovitom. S druge strane, jezik ne podržava paradigmu ako je potrebno prilično napora ili vještine za primjenu paradigme, ali ako je to ipak moguće napraviti smatra se da *omogućuje* (eng. enables) primjenu paradigme.

Van Roy (2009) navodi kako postoji 27 različitih paradigmi programiranja. Programske paradigme možemo podijeliti na dvije velike skupine: **imperativna paradigma** i **deklarativna paradigma**. (Gallage, 2019).



Slika 2.3. Programske paradigme

Svaka programska paradigma ima svoj skup koncepata iz programiranja (Samuel, 2017), a paradigme se mogu i preklapati u nekim elementima (Smyth, 2018). Shriver (1986) smatra kako programske paradigme iniciraju nove vrste programskih jezika ili su inicirane novim programskim jezicima i pripadajućim okruženjima za razvoj. U nastavku ćemo samo kratko opisati nekoliko zanimljivih paradigmi.

Proceduralna paradigma temelji se na „von Neumann-Eckert“ modelu računanja u kojem su program i varijable spremljeni zajedno, a program sadrži niz naredbi kojima se izvodi računanje, pridruživanje vrijednosti varijablama, unos, ispis, preusmjeravanje kontrole i sl. (Tucker, 2007). Pridruživanje vrijednosti varijablama, algoritamske strukture (linijska, razgranata i ciklička), rad s iznimkama, izdvajanje procedura, biblioteke s podrškom za strukture podataka su temeljni elementi ove paradigme (Samuel, 2017; Tucker, 2007). Snyder (1986) jednostavno definira proceduralnu paradigmu rečenicom: potrebno je odlučiti koje procedure želimo te odabrati najbolje algoritme koje možemo naći. Primjeri jezika su Fortran, Ada i C. Većina proceduralnih jezika podržava rekurzije i iteracije, a rekurzija je moćan alat za rješavanje problema posebno ako je problem rješiv metodom „podijeli pa vladaj“ (Samuel, 2017). Redoslijed naredbi ili koraka pri rješavanju problema je bitan što ne odgovara svakom problemu. Osnovne algoritamske strukture ili kontrola tijeka izvođenja programa (eng. control structures) su: *slijed, grananje i ponavljanje* (Meerbaum-Salant, Armoni, & Ben-Ari, 2013).

Objektno orijentirana (OO) paradigma temelji se na modelu u kojem je program kolekcija objekata koji su u međusobnoj interakciji slanjem poruka (Tucker, 2007). Porukama se može

mijenjati stanje objekata. Primjeri jezika su: Smalltalk, Python, C# i Java. Prednosti OO paradigme uključuju smanjenje složenosti, ponovnu upotrebu objekata, modularnost, enkapsulaciju i održavanje (Samuel, 2017).

Funkcijska paradigma omogućuje modeliranje problema kao kolekcije matematičkih funkcija tako da svaka ima područje ulaznih vrijednosti (domenu) i područje rezultata (Tucker, 2007). Naredbe pridruživanja vrijednosti varijablama u ovoj paradigmi nemaju smisla. Primjeri jezika su: Lisp, Haskell, ML i Scheme. Samuel (2017) smatra da su funkcijski jezici izražajniji jer programeri ne moraju voditi računa o složenosti izračunljivosti jer je to prepušteno stroju.

Logička paradigma omogućuje modeliranje problema deklariranjem rezultata kojeg bi program trebao ostvariti umjesto načina na koji će to ostvariti (Tucker, 2007). Deklaracije u programu podsjećaju na skup pravila ili ograničenja (eng. constraints) više nego na naredbe. Primjer jezika u logičkoj paradigmi je Prolog.

Ključno je pitanje, koju programsku paradigmu odabrati kao prvu za poučavanje programera početnika? Ovisno o autorima, kao prvu paradigmu treba odabrati proceduralnu, objektnu, funkcijsku, … ili započeti isključivo s algoritmima ili varijablama i sl. (Cooper, Dann, Pausch, & Pausch, 2003; Sajaniemi & Hu, 2006). Radna skupina sastavljena od organizacija IEEE Computer Society (IEEE-CS) i Association for Computing Machinery (ACM) pri izradi kurikuluma za područje računarstva (2001, str. 22) zaključuje kako ne žele preporučiti niti jedan pristup ni paradigmu za početne predmete iz programiranja obzirom da svaki ima svoje prednosti i nedostatke te preporučuju nastavak istraživanja u području. Bolshakova (2005) zaključuje kako bi studenti trebali naučiti tehnike glavnih programskih paradigmi, odnosno učenje modernih programskih jezika bi trebalo nadopuniti učenjem paradigmi. Preporuku pojašnjava činjenicom da ne možemo predvidjeti budućnost aktualnih programskih jezika jer tijekom vremena se stvaraju novi programski jezici te postojeći mogu izgubiti popularnost.

Dominantni pristup programiranju 1970tih je bilo strukturirano programiranje (Floyd, 1979). 1960tih godina nastao je BASIC koji je bio iznimno popularan kao jezik za poučavanje, a 1970tih nastaje Pascal koji je razvijen s ciljem poučavanja programiranja te je neko vrijeme bio jedan od glavnih jezika na fakultetima, no 1990tih su spomenute jezike većinom zamijenili C, C++ i Java zbog zahtjeva industrije (Tucker, 2007). Prema tome, može se reći da je proceduralno programiranje na fakultetima bilo dominantno do pojave objektno orijentiranih programskih jezika 1990tih godina. U istraživanju koje su proveli Bennedsen & Caspersen

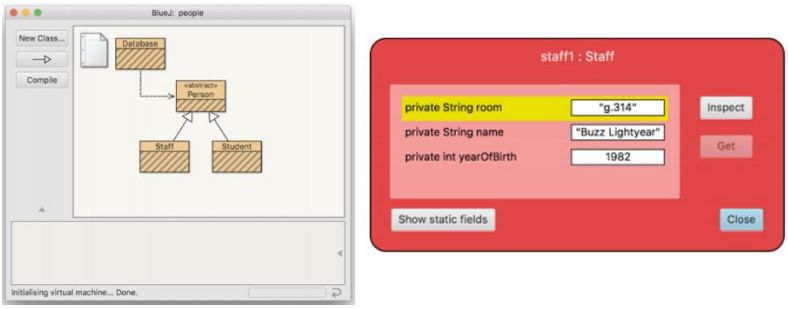
(2007) većina početnih predmeta iz programiranja je koristila objektno orijentiranu paradigmu, ali podaci u istraživanju su prema upisu studenata 1999. godine.

S povećanjem popularnosti objektno orijentirane paradigme, počeli su se javljati argumenti za i protiv započinjanja s objektima tako da su mišljenja podijeljena (R. Lister i ostali, 2006). Primjerice, Cooper i ostali (2003) smatraju kako je primjereniji proceduralni pristup u kojem se započinje s jednostavnijim programima koji s vremenom postaju složeniji i prerastaju u projekte. Na taj se način znanje gradi inkrementalno ili spiralno što je poželjno (Schunk, 2012), dok kod objektno orijentiranog pristupa moraju odmah krenuti s puno većim skupom koncepata isključivo vezanih za objektno orijentiranu paradigmu uz istovremeno usvajanje „uobičajenih" koncepata programiranja. Također se smatra da studenti koji uče objekte prije imaju više problema s razvojem algoritamskog načina rješavanja problema (R. Lister i ostali, 2006).

S druge strane Kölling (1999a) tvrdi kako je iznimno teško programirati u objektno orijentiranom stilu nakon što se netko navikne na proceduralni, što je jedan od razloga zašto bi neki smatraju da bi trebalo započeti s objektima prije. Fowler (2003) to objašnjava primjerom u kojem kaže kako u objektno orijentiranom programiranju ima puno malih objekata s puno malih metoda koji se koriste za premošćivanje (eng. override) što prilično zbunjuje osobe navikle na velike procedure.

Vilner i ostali (2007) su utvrdili kako nije bilo značajne razlike u ukupnom postignuću između studenata koji su započeli s objektno orijentiranom paradigmom kao prvom u odnosu na one koji nisu. Sajaniemi & Hu (2006) su smatrali kako bi pristupe trebalo kombinirati, ali tako da se najprije započne s varijablama (eng. variable-first) u kombinaciji s kontrolama tijeka te se naknadno prelazi na objekte.

Kunkle (2010) ističe tri konkretna primjera pristupa poučavanju objektno orijentiranih koncepata. Prvi primjer odnosi se na pristup „objekti prvi" uz primjenu okruženja kao što je BlueJ u kojem se radi s klasama i objektima bez kodiranja (Slika 2.4).



Slika 2.4. Izgled BlueJ okruženja, preuzeto iz (Kölling, 2019)

U drugom primjeru se koristi vizualni programski jezik Alice, također je ideja „objekti prvi“, ali razlika je u tome što se u Alice može programirati. Treći primjer odnosi se na „tradicionalniji“ pristup gdje se prvo uči proceduralna paradigma. Ragonis & Ben-Ari (2005) su nakon godine dana rada s BlueJ okruženjem zaključili kako BlueJ pomaže početnicima pri usvajanju objektno orijentiranih koncepata, ali ti početnici imaju problema s tijekom izvršavanja programa. Slično tome Kölling i ostali (2003) su kod početnika koji su koristili vizualno okruženje utvrdili bolje usvajanje objektno orijentiranih koncepata no lošije usvajanje struktura podataka i algoritama u odnosu na one koji nisu koristili vizualno okruženje.

Ne može se očekivati neko jedinstveno rješenje niti kao pristup poučavanju niti alat za poučavanje koji može riješiti sve probleme u programiranju (Ragonis & Ben-Ari, 2005). Alat ili pristup koji rješava dio problema može pri tome stvoriti potpuno novi skup drugih problema. Potrebno je proučiti različita okruženja i jezike kako bi odabrali primjereno vlastitom kontekstu poučavanja (karakteristikama učenika i obrazovnom okruženju u kojem se trenutno nalaze).

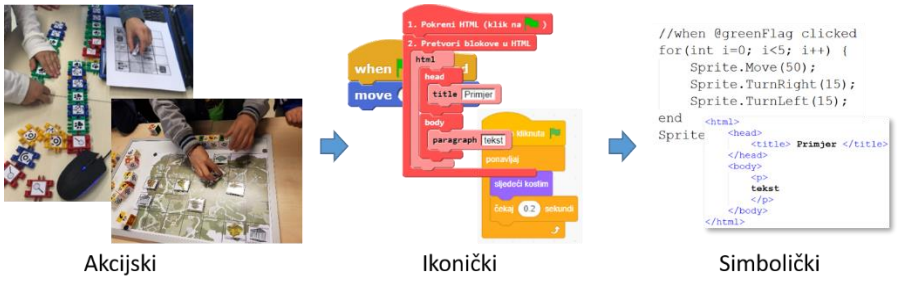
2.2.2 Jezici i okruženja za poučavanje programiranja

Programski jezici slično kao i prirodni jezici omogućuju izražavanje i prijenos ideja no programski jezici su manje „izražajni“ te omogućuju komunikaciju ili prijenos računalnih (eng. computational) ideja (Tucker, 2007), a to je također i razlog što ne mogu biti tako bogati i izražajni kao prirodni. Razina složenosti programskog jezika utječe na razinu truda ili vremena kojeg će učenici uložiti u učenje programiranja, a nastavnik također mora odabrati i stil poučavanja.

Učenje sintakse programskog jezika je donekle slično učenju stranog jezika, ali ipak složenije jer je programski jezik apstraktan te učenici moraju dodatno razumjeti problem i tehnike rješavanja problema u odabranoj programskoj paradigmi, oblikovati konačno rješenje (program) i testirati ga (Futschek, 2013). Odabir programskog jezika za poučavanje početnika nije ni malo jednostavan zadatak, posebno ako uzmemo u obzir posljedice koje taj odabir može imati na učenike, odnosno njihovo daljnje napredovanje u području programiranja. Previše složen jezik će brzo demotivirati učenike, a posljedica toga je odustajanje od učenja programiranja. Takva situacija je nepoželjna, posebno kad se radi o informatici kao izbornom predmetu, a na fakultetima to može značiti odustajanje od polaganja predmeta iz početnog programiranja te konačno od samog studija. Predmeti s početnim programiranjem imaju najviše razine odustajanja (eng. dropout rates) (McCracken i ostali, 2001; Robins i ostali, 2003).

Principi oblikovanja programskog jezika su (Tucker, 2007): sintaksa, nazivi i tipovi te semantika. Sintaksa jezika opisuje ono što čini ispravan program, a odnosi se na osnovni skup riječi i simbola te gramatiku. *Rječnik* (eng. vocabulary) programskog jezika uključuje skup pravila za imenovanje entiteta (varijable, funkcije, klase, …) ili pravila za definiranje naziva. Tipovi određuju vrste vrijednosti kojima program može manipulirati. Semantika se odnosi na značenje programa, odnosno opisuje efekt svake naredbe na vrijednosti varijabli kad se program izvrši. To su općenite stvari koje promatramo i kod odabira programskog jezika, bilo za programiranje ili poučavanje programiranja.

Prema Bruneru (J. S. Bruner, 1966; Schunk, 2012) ljudi imaju tri oblika predstavljanja znanja: *akcijski* (eng. enactive), *ikonički* (eng. iconic) i *simbolički* (eng. symbolic). Akcijski oblik predstavlja fizičke akcije ili manipulaciju fizičkim objektima, a to se u programiranju može povezati s opipljivim programskim jezicima gdje djeca stvaraju programe pomoću fizičkih predmeta kao što su kocke, kartice i sl. (Slika 2.5).



Slika 2.5. Brunerovi oblici predstavljanja znanja u programskim jezicima

Ikonički oblik predstavlja mentalne slike o objektima ili akcijama koji nisu fizički prisutni, a u programiranju bi se mogli povezati s vizualnim programskim jezicima gdje blokovi predstavljaju naredbe programskog jezika, a likovi objekte. Simbolički oblik odnosi se na sustav simbola kojim se kodira znanje, na primjer riječi, matematička notacija i tekstualni programski jezik. Pomoću simboličkog prikaza omogućuje se prikaz apstraktnih koncepata. Riječ „Split“ nam može značiti grad, funkciju programskog jezika, a može jednostavno biti niz besmislenih znakova. Simbolički oblik predstavljanja znanja se razvija zadnji te se kasnije češće koristi iako ljudi zadržavaju i prethodna dva (Schunk, 2012). Učenici nižih uzrasta mogu koristiti opipljive programske jezike (predškolski i početni razredi osnovne škole), no kasnije im više odgovara ikonički te simbolički. Studenti su odrasli učenici, no primijetili smo kako mogu koristiti vizualne jezike (ikonički prikaz), ali kako im jačaju sposobnosti u tekstualnom programskom jeziku imaju sve veći otpor prema vizualnim jezicima.

Prema tome, programske jezike ćemo podijeliti na tri kategorije:

- opipljivi programski jezici,
- vizualni programski jezici,
- tekstualni programski jezici.

Programski jezik smatramo opipljivim (eng. tangible) ako se pri izradi programa koriste naredbe u obliku fizičkih predmeta. Opipljivi programski jezici su primjereni za malu djecu koja još nisu dobro ovladala motoričkim vještinama za rad s mišem i tipkovnicom ili vještinom čitanja. Vizualni i tekstualni se mogu koristiti za sve ostale.

2.2.2.1 Tekstualni programski jezici

Tekstualni programski jezici sadrže naredbe isključivo u tekstualnom obliku, a programer piše naredbe ili programski kôd u odgovarajućem uređivaču kôda ili razvojnom okruženju. Moguće je pisati tekstualne naredbe i u najjednostavnijem uređivaču teksta kao što je Notepad u operacijskom sustavu MS Windows, ali alati tog tipa ne omogućavaju razlikovanje naredbi programskog jezika od običnog teksta jer je sve iste boje, a također nije moguće izvršiti napisani program bez upotrebe nekog drugog alata. Primjeri tekstualnih programskih jezika su: C, C++, Python, Java, C#, Pascal i sl.

Tekstualni programski jezici koji se koriste na fakultetima najčešće ovise o zahtjevima tržišta rada, odnosno ponudi poslova u industriji (posebno ako tvrtke imaju praksu interesirati se na fakultetu za potencijalne zaposlenike). Prilikom istraživanja koji se konkretni jezici koriste za poučavanje na fakultetima 2011. godine izdvojili smo nekoliko najbolje rangiranih sveučilišta koji su u sljedećoj tablici poredani abecedno (Krpan & Bilobrk, 2011):

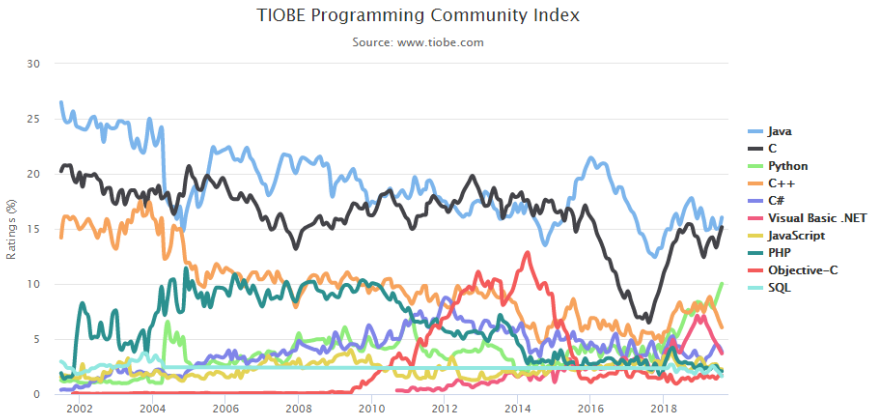
Tablica 2.1. Programski jezici za poučavanje programiranja na sveučilištima (Krpan & Bilobrk, 2011)

Sveučilište	**Programski jezik**
CalTech	Python
Cambridge	Java
Harvard	C, PHP, JavaScript
MIT	Python
Oxford	Haskel
Princeton	JavaScript, Java
University of Chicago	C/C++

U to vrijeme na fakultetima u Republici Hrvatskoj, prema podacima koji su bili dostupni online, na većim fakultetima za početne predmete iz programiranja koristili su se C ili C++ uz iznimku PMFST na kojem su dva predmeta koji se smatraju početnima (P1 i P2) te se pored C-a koristio i QBASIC. Danas se na početnim predmetima iz programiranja na PMFST koriste Python (za poučavanje proceduralne paradigme) i C# (za objektno orijentiranu paradigmu).

Prema pregledu literature iz 2018 godine kojeg su napravili Noone & Mooney (Noone & Mooney, 2018), kao prvi tekstualni programski jezik najčešće se pojavljuje Python. Kod pozitivnih iskustava ističe se kako je Python lagan i jednostavan za učenje, a kod negativnih se ističu poteškoće u prijelazu iz Pythona u druge jezike (npr. Java).

Farooq i ostali (2014) su predložili okvir za analizu programskih jezika koji se temelji na tehničkim osobinama jezika (npr. provjera tipova, čitljivost, lakše pisanje manjih programa, …) i okruženja (npr. zahtjevi industrije, čitljiva sintaksa, …). Analizirali su programske jezike koji se koriste u poučavanju početnika u uvodnim predmetima programiranja od 1994-2011. godine, a jezici koji su bili među najboljima u ukupnom zbroju bodova su Java i Python. Java i Python prema TIOBE indeksu i danas uživaju popularnost među programerima (Slika 2.6).



Slika 2.6. Popularnost programskih jezika (TIOBE, 2019)

Programski jezik Java je također bio popularan na fakultetima 1990tih (Hadjerrouit, 1998), no Hadjerrouit zaključuje kako je ipak previše težak za poučavanje početnika koji nemaju nikakvo predznanje. Složeni programski jezik će utjecati na gubitak motivacije i odustajanje studenata jer u nekom trenutku više neće biti u stanju pratiti nastavu. Početni programski jezik za početnike bi morao imati jednostavnu sintaksu, brzu povratnu informaciju i strukturirani način pisanja kako bi programeri početnici stekli navike urednog pisanja. Java je i dalje popularan jezik tako da će se očito često pojavljivati kao prvi programski jezik (Noone & Mooney, 2018).

Jezici koji se koriste za poučavanje programiranja na fakultetima često nisu namijenjeni poučavanju, a s podsmijehom se gleda na korištenje jezika koji jesu namijenjeni poučavanju, pogotovo ako se inače koriste za niže uzraste (kao npr. Logo) (Jenkins, 2002). Također se izbjegava korištenje „ozbiljnijih" jezika koji nisu zastupljeni u industriji (npr. Pascal). Profesionalni programski jezici su prilagođeni profesionalcima pa tako koriste različite elemente koji su početnicima prilično teški i nerazumljivi. Na primjer, kod korištenja logičkih operacija I/ILI u programskom jeziku Python koristimo operatore *and* i *or* koji su bliski prirodnom engleskom jeziku, no u programskom jeziku C i C# umjesto toga se koriste kombinacije znakova: && za *and*, odnosno || za *or*. Treba također voditi računa i o tome da upotreba jednog znaka umjesto dva u tim primjerima nije sintaksna pogreška već rezultira različitim izvršavanjem programa.

Profesionalni tekstualni jezici, odnosno oni koji se koriste u tvrtkama imaju velike mogućnosti koje početnici nisu u stanju svladati ni koristiti (Grandell, Peltomäki, Back, & Salakoski, 2006). Koncepti iz programiranja općenito su sami po sebi dovoljno apstraktni i složeni da učenicima (bez obzira na dob) ne bi trebalo dodatno otežavati sa složenom sintaksom jezika.

2.2.2.2 Vizualni programski jezici

Vizualno programiranje (eng. visual programming, VP) odnosi se na bilo koji sustav koji omogućuje korisniku izradu ili specifikaciju programa u dvije ili više dimenzija (Myers, 1990). Tekstualni programski jezici ne pripadaju ovoj definiciji jer ih prevoditelji obrađuju kao dugačke, jednodimenzionalne tokove (eng. streams). Myers (1990) posebno ističe kako okruženja koja služe za crtanje ili izradu slika ne pripadaju u kategoriju VP jer se njima ne mogu izraditi računalni programi. Kod *vizualizacija programa* (eng. program visualisation, PV), grafički elementi se koriste za ilustraciju nekog dijela programa te izvršavanja programa, a kod vizualnog programiranja se grafički elementi koriste za stvaranje programa. Pojam *vizualni programski jezici* odnosi se na sve sustave koji koriste grafičke elemente uključujući vizualno programiranje i vizualizaciju programa.

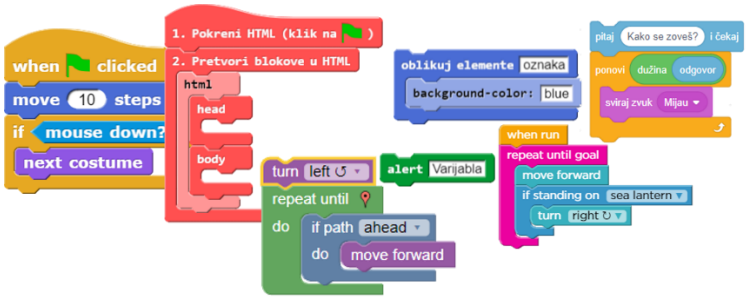
Vizualni programski jezici potječu još iz 1960-tih (Boshernitsan & Downes, 2004). Globalno ih možemo podijeliti na dvije velike kategorije:

- čiste vizualne programske jezike,
- hibridne programske jezike.

Kod čistih vizualnih programskih jezika programer stvara ili slaže program od grafičkih elemenata koji predstavljaju elemente programskog jezika (naredbe, varijable i sl.), a program

se izvršava u istom vizualnom okruženju. U ovoj kategoriji ne postoji prijevod ili prijelaz u tekstualni oblik i obrnuto. Hibridni jezici se sastoje od kombinacije vizualnih i tekstualnih elemenata (Weintrop, 2015). Grafički elementi kojima se predstavljaju naredbe programa mogu biti elementi slagalice (eng. puzzle, jigsaw), dijelovi dijagrama toka, ikone i sl. (Myers, 1990).

Programski jezici koji koriste naredbe u obliku dijelova slagalice nazivaju se *blokovski programski jezici* (eng. block-based programming languages) (Weintrop & Wilensky, 2015). U takvim okruženjima korisnici pomoću miša dovlače i slažu naredbe u programe. Svaka naredba ili blok ima oblik prema kojem korisnik može lako znati može li se spojiti ili ne s nekim drugim blokom. Blokovi su često obojani različitim bojama koje služe jednostavnijem prepoznavanju kojoj vrsti blok pripada, ali svaki jezik ima svoju paletu boja (Slika 2.7).



Slika 2.7. Boje blokova

Papert je 1970tih godina razvio programski jezik Logo i uveo *kornjačinu grafiku* ili *kornjačinu geometriju* (Papert, 1980). Tijekom poučavanja djece programiranju u Logu Papert i njegov tim su uočili kako djeca uče bolje ako se „užive“. Na primjer, ako dijete prošeta po prostoriji u obliku kvadrata (eng. walk the square), lakše će razumjeti kako usmjeriti kornjaču na zaslonu računala kako bi nacrtala kvadrat ili kako programirati robotsku kornjaču da se prošeta u stvarnom svijetu na isti način (McNerney, 2004). Programski jezik Logo je imao ogroman utjecaj na vizualne programske jezike uvođenjem brojnih karakteristika koje su danas ključne u vizualnim jezicima, kao na primjer: upotreba naredbi za kretanje (npr. idi naprijed, okreni se), prisutnost aktera na zaslonu računala koji izvršava te naredbe (npr. Logo kornjača) i sintaksa koja je oblikovana upravo za početnike. Primjer vizualnog jezika koji je nastao na temelju karakteristika Logo jezika je Scratch koji je pokrenut 2007. godine (Maloney, Resnick, & Rusk, 2010). Scratch podržava „tinkering“ pristup u kojem početnici izvršavaju male dijelove koda, omogućuje prikaz izvršavanja naredbi (označavanje aktivne naredbe i prikaz rezultata na prostoru pozornice) i prikaz vrijednosti varijabli. Logo omogućuje vizualni prikaz što kornjača radi nakon što se pokrene program (što olakšava početnicima jer im daje brzu i jasnu povratnu

informaciju). Premda Scratch i Logo dijele zajedničku karakteristiku vizualnog prikaza rezultata izvršavanja, Scratch nema problem sintakse jer se koriste blokovi, a tekst na blokovima ima prijevod na mnoge prirodne jezike. Aivaloglou & Hermans (Aivaloglou & Hermans, 2016) su u svojem istraživanju analizirali 250.000 Scratch projekata. Zaključuju kako se Scratch većinom koristi kao prvi susret s programiranjem i za stvaranje jednostavnih programa i interaktivnih animacija jer su većinom programi mali i interaktivni bez uvjetnih naredbi (eng. conditional statements) odnosno odluka.

Upotrebom blokovskog jezika zaobilazi se ili bolje rečeno minimizira problem sintakse. Neka jednostavna pravila postoje, ovisno o oblicima kako se što može i smije slagati, ali je to prilično jednostavno i intuitivno. Uklanjanjem problema sintakse ostaje više vremena za sve ostalo što bi početnici trebali usvojiti tijekom učenja programiranja. Učenici u Scratch-u mogu započeti učenje sa složenim konceptima (npr. ponavljanje ili petlja) znatno ranije nego što se to radi s klasičnim programskim jezicima. Berglund i Lister (2010) ističu kako postoji povezanost između okruženja za učenje i učenja programiranja. Složena profesionalna okruženja koja se koriste za rad s programskim jezicima C#, Java i sl. često su sama po sebi zastrašujuća početnicima jer imaju jako puno mogućnosti te se početnici teže snalaze. Na primjeru Scratch-a možemo vidjeti da je okruženje za rad prilično jednostavno, može se koristiti u web pregledniku, ne zahtijeva puno niti od računala niti od korisnika. Sve naredbe su dostupne i vidljive u paletama te će učenici često sami pronaći i iskoristiti nove naredbe istražujući dok to u tekstualnim jezicima nije tako jednostavno. Na primjer, ako želimo dodati element u neku kolekciju i pri tome sami istražiti kako pronaći odgovarajuću funkciju, u tekstualnom programskom jeziku se moramo sjetiti kako bi riječ *dodaj* mogla zvučati na engleskom jeziku, na primjer: *add*, no u konkretnom programskom jeziku ili ovisno o samoj vrsti kolekcije to isto tako može biti *append* ili *push* ili *enqueue*. Kod vizualnih programskih jezika, učenik će pronaći blok naredbu na kojoj piše „*dodaj … u listu“*.

Premda bi programski jezik u slučaju učenja i poučavanja programiranja trebao više biti sredstvo za učenje općih koncepata iz programiranja, ipak koncepti ovise o samom jeziku tako da učenik usvaja i koncepte specifične upravo za odabrani jezik i njegovo okruženje.

Meerbaum-Salant i ostali (Meerbaum-Salant i ostali, 2013) su istraživali poučavanje koncepata iz računarstva pomoću Scratch-a u srednjoj školi te su utvrdili kako su učenici razumjeli sve koncepte koje su koristili osim koncepta petlje i varijabli. Učenici su uspješno izrađivali stvari u Scratch-u, ali su slabo usvajali koncepte programiranja. Nedostatak prvih verzija Scratch-a je bio ograničen skup naredbi koje se su se mogle koristiti, tako da su napredniji učenici često bili

frustrirani jer jezik nije bio dovoljno izražajan. Iz tog razloga su razvijani dijalekti Scratch-a, kao što je BYOB (akronim od: Build Your Own Blocks) iz kojeg kasnije nastaje "Snap!“. Osnovna prednost BYOB-a je bila izrada vlastitih blokova čime okruženje više nije bilo ograničeno, te izrada klonova postojećih likova koji su se stvarali tijekom izvođenja programa. Scratch je u međuvremenu s verzijom 2.0 proširen mogućnostima izrade vlastitih blokova kao i klonovima. Klonovi mogu poslužiti kao primjer stvaranja objekata tijekom izvođenja programa što bi eventualno moglo pomoći početnom učenju objektno orijentiranog programiranja.

Ima mnogo različitih vizualnih jezika kao na primjer: Alice (http://www.alice.org), Greenfoot (http://www.greenfoot.org), Kodu (https://www.kodugamelab.com/), EToys (http://www.squeakland.org), App Inventor (http://appinventor.mit.edu), GameMaker (https://www.yoyogames.com/gamemaker), NXTG (Lego Mindstorms), Google Blockly, Pencil Code i sl. Premda blokovski programski jezici dijele neke slične karakteristike, Alice se razlikuje od ostalih jer omogućuje manipuliranje objektima u tri dimenzije (3D). Možda je zbog treće dimenzije nešto složeniji jezik u odnosu na Scratch i slične 2D jezike. Najčešće se pojavljuje u početnim predmetima iz programiranja u visokom obrazovanju (Noone & Mooney, 2018). Također treba napomenuti da je Alice jezik koji omogućuje kasniji prijelaz na tekstualni programski jezik Java tako da i to uvjetuje odabir Alice kao prvog programskog jezika. Google Blockly omogućuje prijevod blokova u tekstualne jezike kao što je na primjer Python ili JavaScript, ali nije moguće pisati program u tekstualnom jeziku. Pencil Code omogućuje stalno prebacivanje iz blokova u tekst, moguće je izrađivati programe na oba načina, a od tekstualnih jezika trenutno podržava JavaScript i CoffeeScript, uz dodatak HTML-a i CSS-a. Cilj Pencil Code jezika je stvoriti samopouzdanje kod početnika kako bi u nekom trenutku mogli početi pisati programe bez blokova i Pencil Code jezika (Bau, Bau, Dawson, & Pickens, 2015).

Upotreba vizualnog programiranja pozitivno utječe na računalnu pismenost (Werner, Denner, Bliesner, & Rex, 2009). Gibbs & Coady (2010) su koristili Scratch za uvodni predmet iz programiranja za prvu godinu preddiplomskog studija i zaključili kako je većina studenta uspješno napravila programe u Scratch-u koji rade te su čak više istraživali od samih zahtjeva zadatka. No, pokazalo se da nisu dublje razumjeli koncepte i da imaju probleme s apstrakcijom jer su npr. stvarali previše kompliciranа rješenja. Iskusniji i stariji početnici smatraju blokovske jezike dječjim okruženjem te im je naporno slagati programe pomoću blokova kao i tražiti blokove po paletama (Monig, Ohshima, & Maloney, 2015). Jedna od tehnika koja bi tu mogla pomoći je omogućavanje pisanja kôda u obliku teksta, kad to sami početnici žele ili uređivati

blokove pomoću uređivača teksta. Neka okruženja to i omogućuju, na primjer novije verzije Snap!-a omogućuju traženje blokova koje treba dodati pomoću tipkovnice te isto tako dodavanje novih blokova pomoću tipkovnice.

Grafički elementi kojima zamjenjujemo tekstualne naredbe u vizualnim programskim jezicima često zauzimaju više prostora nego tekst tako da na zaslon računala zapravo stane manje. To može biti problem kod velikih programa jer se smanjuje preglednost. Današnji vizualni programski jezici kao na primjer Scratch ili „Snap!“ omogućuju automatsko raspoređivanje blokova u prostoru za uređivanje, ali i dalje je to nepregledno ili automatsko uređivanje neće rasporediti onako kako bi korisnik želio. Uklanjanje pogrešaka ili uređivanje velikih programa je naporno jer često nema mogućnosti traženja neke naredbe kao u tekstualnom okruženju. Myers (1990) navodi kako je vizualne programe teško prenositi ili slati, za razliku od tekstualnih. Danas to nije toliko teško jer se u različitim okruženjima mogu spremiti u obliku datoteke koja se može vrlo jednostavno spremiti na lokalno računalo, prenijeti ili poslati nekome (npr. Snap! programi se zapisuju u obliku XML datoteke). Okruženja kao što su Tynker (http://tynker.com) ili Hour of code (https://hourofcode.com/) omogućavaju spremanje programa online, ali nije moguće spremiti rješenje na lokalno računalo ili ga poslati nekome.

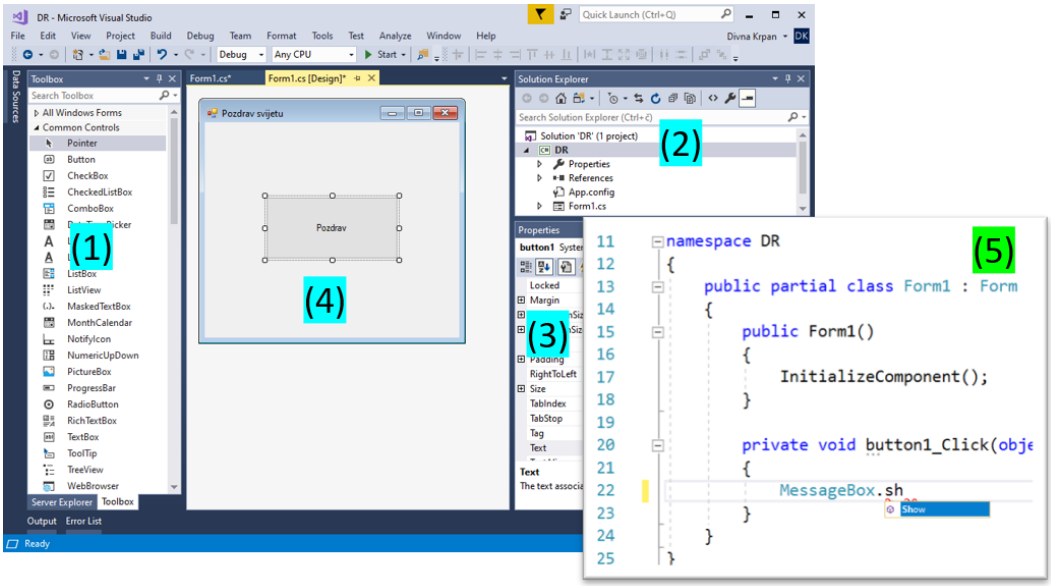
Iskusni programeri mogu prelaziti iz jednog okruženja u drugo te mijenjati svoj način predstavljanja problema kako bi se prilagodili novom okruženju jer se njihovo znanje odnosi na cijeli sustav (uređivač kôda, operacijski sustav, uređaji za unos i ispis podataka i sl.) (Rogalski & Samurcay, 1990). Početnici obično ne mogu sagledati cijelu sliku. Na primjer, u programskom jeziku C# se koristi decimalna točka, ali prema regionalnim postavkama računala na hrvatskom jeziku operacijski sustav koristi zarez dok točka služi za razdvajanje tisućica. Prema tome, početnici će upisati 3.123 no to će se interpretirati kao 3123. Početnicima neće biti jasno zašto njihov program ne ispisuje ispravan rezultat.

2.2.2.3 Razvojna okruženja

Razvojno okruženje ili *okolina* (eng. environment) predstavlja programsku podršku u kojoj se može pisati programski kôd tj. naredbe podržanog programskog jezika. Na primjer, možemo pisati C# programski kôd u običnom uređivaču teksta, ali nam jednostavni uređivači teksta neće bojati ključne riječi niti će biti moguće pokrenuti program. Razvojno okruženje bi trebalo imati: prilagođeni uređivač kôda, prevoditelj, alat za uklanjanje pogrešaka i sustav pomoći. Prilagođeni uređivač kôda znači da je prilagođen programskom jeziku, odnosno da će obojati naredbe programskog jezika posebnom bojom kako bi korisniku olakšao pisanje. Dodatne mogućnosti su automatsko dovršavanje naredbi (nakon što korisnik počne tipkati), automatsko

uvlačenje blokova kôda podržavajući strukturirano pisanje i sl. Ako je programski jezik namijenjen za izradu grafičkih aplikacija onda bi bilo poželjno i da ima grafički dizajner (eng. designer) za oblikovanje grafičkih sučelja. Za profesionalne programere ključna je i dostupnost različitih proširenja ili dodataka kojima mogu nadopuniti ono što njima konkretno treba u okruženju.

Primjer profesionalnog razvojnog okruženja je Microsoft Visual Studio (MSVS). Prva verzija MSVS objavljena je 1997. godine pod nazivom *Visual Studio 97* (Otey, 1997). To je bio jedan od prvih pokušaja integracije Microsoftovih alata za programiranje u jedno okruženje, vođeno na neki način idejom Microsoft Office paketa. Popularna izdanja koja se pojavljuju za većinu generacija MSVS su: *Enterprise, Professional* i *Community*. *Enterprise* ima najviše mogućnosti, a od 2010. godine uveden je *Professional* kao ograničena komercijalna verzija. Prije pojave besplatnog *Community* izdanja 2014. godine programeri su uz svaku novu verziju MSVS mogli besplatno koristiti *Express* izdanje. Čak i besplatna izdanja MSVS sadrže mnoštvo mogućnosti s kojima je moguće napraviti profesionalne aplikacije za različite platforme (Slika 2.8).

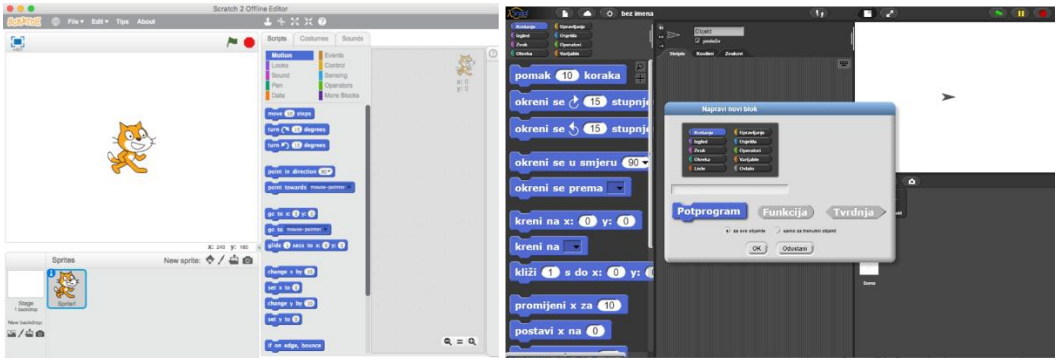


Slika 2.8. Microsoft Visual Studio 2017

Na slici možemo vidjeti glavne dijelove okruženja koje studenti koriste: (1) traka s alatima (eng. toolbox), (2) prozor s rješenjem (eng. Solution explorer), (3) svojstva (eng. properties), te u središnjem dijelu (4) alat za oblikovanje sučelja grafičkih aplikacija. U istom središnjem dijelu otvara se (5) uređivač kôda. MSVS također ima dobar alat za otkrivanje i otklanjanje pogrešaka (eng. debugger) koji omogućava praćenje stanja varijabli, zaustavljanje izvršavanja,

izvršavanje korak po korak i sl. Dodatna prednost je i *kontekstualna pomoć* (eng. context help) što znači da korisnik u bilo kojem trenutku može pritisnuti tipku F1 na tipkovnici. Pri tome se otvara tekst pomoći koji je točno vezan za kontekst u kojem se korisnik nalazio kad je pritisnuo tipku. Studenti prve godine koji su odslušali Programiranje II i položili pismeni dio ispita nakon 15 tjedana korištenja MSVS ponekad nisu u stanju opisati najvažnije dijelove okruženja te imaju poteškoće pri snalaženju čim im netko zatvori neki od glavnih prozora.

Razvojna okruženja za vizualne programske jezike su često online pa nije potrebna nikakva instalacija na računalu već samo web preglednik s pristupom Internetu. Obzirom na namjenu, okruženja su jednostavna te sadrže sve dostupne naredbe raspoređene po paletama (Slika 2.9).



Slika 2.9. Razvojna okruženja za Scratch 2.0 i Snap!

Scratch je dostupan online i kao posebna instalacija, a na gornjoj slici lijevo je prikazana instalirana verzija za Scratch 2.0. „Snap!“ je dostupan samo kao web aplikacija iako se cijeli po potrebi može spremiti lokalno. Tynker je dostupan kao web aplikacija, ali se može instalirati kao aplikacija za tablet.

Kölling (1999b) ističe kako je okruženje koje početnici koriste za programiranje ključno za uspjeh početnog programiranja te napominje kako su početnici u objektno orijentiranom programiranju imali ozbiljne probleme upravo s okruženjem. Okruženja s dobrom podrškom za objekte su se pokazala presloženima za početnike na prvoj godini studija.

Guzdial (2004) je analizirao nekoliko okruženja za programiranje te zagovara okruženja koja podržavaju izradu igara, multimediju i simulacije. Bilo bi idealno da studenti kroz početno obrazovanje (osnovnu i srednju školu) uče jezike tipa Logo i sl. namijenjene upravo poučavanju programiranja za niže uzraste, no studenti se s programiranjem i programskim jezicima često prvi put susreću upravo na fakultetu, a onda se od njih u relativno kratkom vremenu očekuje usvajanje koncepata iz programiranja i iskustvo u radu s profesionalnim programskim jezikom. Studenti bi nakon studija trebali biti osposobljeni za posao u struci. Nastavnici na PMFST su

suočeni sa situacijama kad studenti koji završe neki od informatičkih smjerova imaju želju raditi u industriji, a ne u obrazovanju tako da je problem kako uskladiti ta dva svijeta još izraženiji.

Odabir programskog jezika uvjetuje i posljedični odabir programskog razvojnog okruženja. Ako na primjer odaberemo Scratch onda se može birati samo između web okruženja ili lokalne verzije, dok primjerice odabir programskog jezika Python ili C# nude mnogo širi raspon okruženja. Vrijeme koje početnici provode u okruženjima za početnike samostalno istražujući nije dovoljno da bi naučili programirati već ih treba kombinirati s vođenjem i praćenjem nastavnika te povratnim informacijama. Važno je razmotriti različite metoda poučavanja te odabrati ono što je u skladu s karakteristikama učenika i okruženjem, odnosno programskim jezikom koji je odabran za rad.

2.3 Metode poučavanja programiranja

Hadjerrouit (1999) smatra kako je učenje objektno orijentiranog programiranja većini početnika posebno teško jer je apstraktnije od proceduralnog pristupa, no također zahtijeva i drugačiji način razmišljanja kod analize i oblikovanja, odnosno prije faze kodiranja.

Kod učenja početnog programiranja može se koristiti *objektivistički model* (eng. objectivistic model) koji podrazumijeva pasivni prijenos znanja (pasivno slušanje predavanja, čitanje udžbenika, programiranje bez aktivnog razmišljanja i sl.), odnosno pretpostavka da nema početnog znanja (Hadjerrouit, 1999). Rezultat takvog učenja je nedostatak konceptualnog znanja (čak i nakon godine dana učenja), loše navike u programiranju i miskoncepcije o načinu razvoja programa.

2.3.1 Konstruktivistički pristup u programiranju

S druge strane, konstruktivistički pristup se temelji na procesu aktivne izgradnje ili konstrukcije znanja (Ben-Ari, 2001). Konstruktivizam je teorija učenja u kojoj je ključna ideja da se znanje ne može prenijeti na učenika pasivnim učenjem iz udžbenika i predavanja već je potrebno stvoriti pedagoške uvjete za konstrukciju znanja i razumijevanje (Tchoshanov, 2013). To ne znači da učenici mogu raditi bilo što već se treba stvoriti okruženje za učenje. U tradicionalnoj učionici naglašavaju se osnovne vještine, nastavni sadržaj se predstavlja u malim dijelovima pomoću udžbenika, a nastavnik prenosi informacije učenicima te traži točne odgovore na pitanja. Ideja konstruktivističkog pristupa je preusmjeriti fokus na velike koncepte, a vrednovanje usvojenog znanja je autentično (Schunk, 2012). Učenici u konstruktivističkom pristupu aktivno stvaraju novo znanje temeljeno na postojećem. Važno je voditi računa o eventualnim konfliktima s prethodnim znanjem kako bi se prilagodilo poučavanje novih

koncepata inače se neće usvojiti na odgovarajući način (Hadjerrouit, 1999). Znanje se u konstruktivističkom pristupu usvaja rekurzivno kombinacijom novih senzornih informacija s postojećim znanjem stvarajući nove kognitivne strukture (Ben-Ari, 2001). Pri tome učenje mora biti aktivno uz vođenje nastavnika i povratne informacije učenika. Obzirom da svaki učenik dolazi s različitim predznanjem, odnosno kognitivnim strukturama koje je stvorio ranije, svaki će učenik također nastaviti graditi novo znanje na svoj način. Stoga je uloga nastavnika kao mentora ili voditelja pri učenju posebno složena jer mora razumjeti postojeće kognitivne strukture učenika, odnosno znati što učenici trenutno razumiju. J. Mason (J. Mason, 1994) ističe kako inače nije toliko problem u klasičnom predavanju već u pretpostavci da učenik stvarno zna ono što nastavnik kaže ili prenese tijekom predavanja. Također je teško očekivati da će učenik konstruirati novo znanje na temelju riječi ili predavanja koje čuje od nastavnika. Ben-Ari ističe kako pasivno učenje ne može uspjeti upravo zbog spomenute činjenice da svaki učenik donosi sa sobom svoje početno znanje na kojem će dalje graditi svoj poseban način.

Na prvi pogled aktivnosti kao što su *učenje otkrivanjem* (eng. discovery learning), izrada projekata u grupi i sl. izgledaju kao da baš odgovaraju konstruktivističkoj teoriji, no ključna pretpostavka je aktivno sudjelovanje učenika jer nema smisla ako učenik nešto radi, a mislima nije u tome (Resnick, 1997) pa prema tome samo uvođenje novog pristupa bez osiguravanja sudjelovanja neće imati očekivan učinak.

Schunk (2012) iznosi nekoliko principa konstruktivizma:

- postavljanje problema
- fokus na primarnim konceptima učenja
- praćenje mišljenja učenika
- prilagodba kurikuluma učenicima
- vrednovanje u kontekstu poučavanja

Prema tome bi nastavnici najprije trebali postaviti problem koji je učenicima važan, potičući njihov interes umjesto da im se „prijeti“ ispitima znanja. Nastava se ne mora organizirati doslovnim praćenjem priručnika za programiranje već na način koji ima više smisla ovisno o samim učenicima i promjenama fokusa poučavanja. Na primjer, često se u udžbenicima ili priručnicima za programiranje kreće od koncepata programskog jezika, tipova podataka, ulaznih i izlaznih naredbi, linijskih algoritamskih struktura, grananja, petlji itd. U Scratch-u čak možemo rano započeti s konceptom ponavljanja ili petlje (Slika 2.10).

Slika 2.10. Animacija hoda

Animacija kretanja lika znači da moramo brzo i neprestano izmjenjivati slike (kostime) tj. imamo potrebu koristiti neku naredbu kojom možemo stalno ponavljati izmjenu slike, a to je u Scratch-u beskonačna petlja „ponavljaj“. Petljom smo krenuli prirodno, prema potrebi zadatka.

Problem kod konstruktivističkog pristupa je i vrednovanje. Provjerama znanja kojima se inače ispituje činjenično znanje mogu se previdjeti miskoncepcije učenika (Ben-Ari, 2001). Možda je lakše pratiti postignuća učenika prema točnim odgovorima na pitanja u ispitima znanja, ali bi trebalo razmišljati i o načinu na koji su učenici došli do odgovora. Kurikulum treba prilagoditi predznanju učenika jer postavljanje ljestvice (zahtjeva) malo iznad trenutnih sposobnosti učenika (odnosno unutar zone približnog razvoja (eng. proximal develompent)) stvara izazov za učenike i potiče učenje. S druge strane, ako učenicima nedostaje smisao ili značenje onog što uče ili su zahtjevi postavljeni previsoko, onda će im učenje predstavljati problem. Konstruktivistički pristup također zahtijeva vrednovanje učenja u kontekstu u kojem učenici uče. Ispiti kojima se inače provjerava znanje oblikuju se u skladu s ciljevima učenja nekog predmeta, ali su odvojeni od samog procesa poučavanja. Ovdje se naravno ne želi reći da ispiti znanja i testovi nisu primjereni već da bi trebalo uskladiti načine vrednovanja s načinom poučavanja. To znači, ako uvodimo elemente konstruktivističkog pristupa onda tome treba prilagoditi vrednovanje. Konstruktivističko poučavanje u kombinaciji s klasičnim ispitima dovodi do konflikta, a učenici će eventualno shvatiti kako na kraju moraju točno odgovoriti na pitanja u ispitu te da im je vjerojatno lakše memorirati odgovore koji se očekuju nego ulagati trud u izradu projekata ili aktivno sudjelovanje u nastavi („recite mi što trebam naučiti za prolaznu ocjenu i gdje to mogu naći“). Pitanja tipa istina/laž ili pitanja višestrukog odgovora koje je lako ocijeniti ne mogu pokazati dublje razumijevanje koncepata.

Jedna od pojava kod učenja programiranja kod početnika je *bricolage* ili improvizacija od postojećih dijelova koja se manifestira kao neprestano traženje i uklanjanje pogrešaka sve dok

se ne dobije rješenje koje radi. Premda ponekad i eksperti koriste tu tehniku, metoda ipak nije učinkovita za početnike jer im ne pomaže pri apstrakciji i organizaciji znanja. Prema (Ben-Ari, 2001) studenti koji su izvrsni u tehnici *bricolage* ne snalaze se u sustavima koji zahtijevaju konkurentnost ili izvršavanje u realnom vremenu te komunikaciju, odnosno u rješavanju karakterističnih problema koje susreću programski inženjeri. S tim mišljenjem se slaže i Michaelson (2018) koji ističe kako je na primjer programski jezik Logo napravljen s ciljem olakšavanja prijelaza iz *bricolage* načina rada u apstraktni način razmišljanja, ali uzevši u obzir upravo djecu koja u konkretnoj fazi razvoja koriste *bricolage* te smatra kako se nije baš razmišljalo o upotrebi Logo jezika kod poučavanja odraslih početnika (srednja škola i studij).

Bez obzira na odabranu teoriju, pristup ili metode poučavanja i sl. nemoguće je promatrati dijelove didaktičkog trokuta potpuno izolirano jer primjerice karakteristike učenika utječu na odabir metoda poučavanja, ali isto tako i na sadržaj koji je primjeren za te učenike. U sadržaj programiranja spadaju i koncepti odabranog programskog jezika, odnosno programski jezik. Prema tome, ne možemo potpuno odvojiti programske jezike i metode poučavanja programiranja obzirom da su neki od jezika nastali upravo u svrhu učenja i poučavanja programiranja.

2.3.2 Uloga programskog jezika u poučavanju programiranja

Postoji više pristupa poučavanju programera početnika koji se razlikuju prema odlukama vezanima za programski jezik: upotreba smanjenog programskog jezika, izvršavanje programa neovisno o računalu (ručno ili mentalno) te izbacivanje programskog jezika u potpunosti (upotreba pseudokôda ili simboličkog jezika).

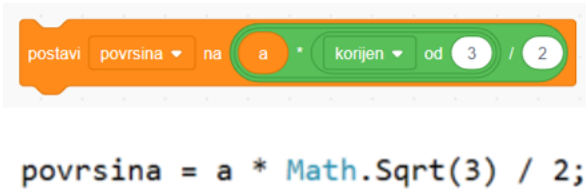
Kelleher & Pausch (2003) su analizirali načine na koje bi programiranje moglo biti pristupačnije početnicima te su izdvojili tri: pojednostaviti programiranje, pružiti podršku učenicima i motivirati učenike. Obzirom da pamćenje i prisjećanje sintakse predstavlja problem za početnike Kelleher & Pausch smatraju kako bi sintaksa trebala biti jednostavnija, intuitivnija te samim tim lakša za pamćenje. Jedan od načina uklanjanja problema sintakse je oblikovanje grafičkih elemenata koji zamjenjuju naredbe programskog jezika.

Brusilovsky i ostali (1994) predlažu tri različita pristupa u poučavanju programera početnika:

1. Inkrementalni pristup,
2. Pristup mini-jezika,
3. Pristup *podjezika* (eng. sub-language).

U inkrementalnom pristupu programski jezik se dijeli na niz podskupova te se spiralno napreduje. Pristup mini-jezika podrazumijeva upotrebu malog i jednostavnog programskog jezika. Brusilovsky i ostali (1994) spominju Logo i „Karel the Robot“ kao tipične primjere mini-jezika. Mini-jezici sadrže mali skup naredbi, a naredbama se obično kontrolira kretanje *aktera* (eng. actor) ili *lika* (eng. sprite) npr. kornjača, robot i sl. u ograničenom *mikrosvijetu* (eng. microworld). Mikrosvijet predstavlja malu domenu objekata i aktivnosti koji su implementirani kao računalni programi, a učenik je izravno u interakciji s njima. Mini-jezici su mali, jednostavni i intuitivni, a ograničeni skup naredbi ih čini primjerenim za početnike. Istraživanje ograničenih mikrosvjetova kroz igru pojavilo se kao alternativa klasičnom poučavanju temeljenom na jezicima i industrijskim tehnikama razvoja programske podrške (Michaelson, 2018). Ideja je krenula negdje od Papertovog Logo jezika (Papert, 1980) koji se smatra jednim od prvih mini-jezika. Napravljen je po principu razmišljanja od konkretnog do apstraktnog. Kad učenici upišu naredbe programskog jezika, mogu odmah vidjeti rezultat u obliku vizualnog prikaza izvršavanja, odnosno kretanja kornjače u mikrosvijetu. A. Kay (1993) je posjetio Paperta i njegov tim 1968. godine te je bio impresioniran kako školska djeca uče programirati u Logo programskom jeziku. To ga je inspiriralo da počne poučavati i Smalltalk na sličan način, prateći ideju mikrosvijeta i vizualnog okruženja s kornjačinom grafikom.

Razvojem računalne podrške, ideja se nastavlja u vizualnim programskim jezicima kao što su na primjer Scratch i Alice. Logo je imao kornjaču, dok Scratch i slični jezici imaju cijelu paletu različitih likova. Naravno, postoje stvari koje nije moguće jednostavno napraviti u nekom mikrosvijetu (Michaelson, 2018) ili nije uopće moguće napraviti. Uzmimo za primjer jednostavnu formulu za računanje površine jednakostraničnog trokuta napisanu u vizualnom jeziku Scratch i tekstualnom jeziku C# (Slika 2.11).



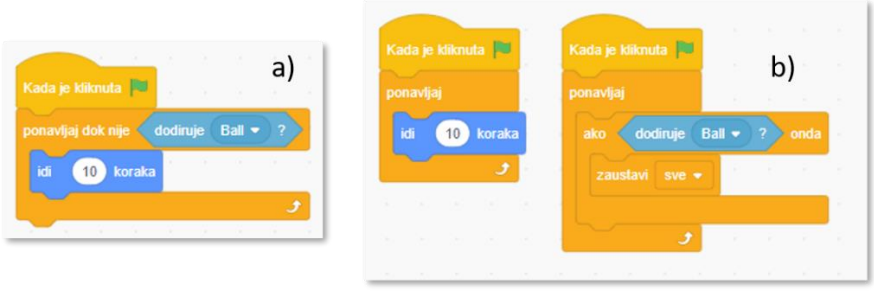

```
povrsina = a * Math.Sqrt(3) / 2;
```

Slika 2.11. Površina jednakostraničnog trokuta u Scratch-u i C#-u

Oba oblika formule su zapravo složenija i manje pregledna od onog što bi napisali na papiru kad rješavamo matematički zadatak, ali ako učenik nema problema s tipkanjem teksta, onda je zapravo brže napisati formulu u obliku teksta nego je posložiti u vizualnom programskom jeziku. Isto tako, učenik koji dobro vlada simboličkim prikazom preferira tekstualni oblik.

Premda upotreba mini-jezika olakšava učenje i poučavanje programiranja, problem je što ograničava učenje rješavanja problema u posebnom okruženju s jezikom specifične namjene umjesto nekog općenitog jezika (eng. general purpose language) ili profesionalnog jezika koji se koristi u industriji. Studenti koji su na PMFST koristili vizualni programski jezik Scratch za izradu jednostavnih projekata su ponekad imali dobre ideje ili algoritme koje nisu mogli implementirati zbog ograničenih mogućnosti jezika pa su se morali tome prilagođavati što im je stvaralo frustracije. Jasno je da bi početnici trebali povezati koncepte i konstrukte koje su učili u jeziku namijenjenom poučavanju programiranja s istima u jeziku opće namjene jer ih želimo naučiti programirati, ali nije u potpunosti jasno na koji način to postići i koliko bi zapravo jezici za poučavanje trebali sličiti „ozbiljnim“ programskim jezicima (Kelleher & Pausch, 2003).

Potrebno je istaknuti kako nastavnici moraju biti oprezni pri primjeni mini-jezika kao što su na primjer blokovski jezici jer oni s prednostima nose i određene nedostatke. Moors i ostali (2018) ističu kako početnici u Scratch-u i sličnim blokovskim jezicima mogu razviti loše navike. Jedna od tih loših navika je izrada programa počevši od najsitnijih dijelova koje spajaju u cjelinu – ekstremna primjena pristupa od dna prema gore (eng. bottom up). Takav način rada je u konfliktu s idejom oblikovanja algoritma za rješavanje problema. Druga loša navika je upotreba pristupa od vrha prema dolje (eng. top down) na ekstreman način, rastavljajući zadatke na podzadatke koji u nekom trenutku postaju besmisleni. Na primjer, ako treba napisati program u kojem lik treba stati kad dotakne drugog lika, onda možemo primijeniti petlju „ponavljaj dok nije“ (eng. repeat until) (Slika 2.12.a), no stil jezika koji omogućuje dijelove kôda koji se paralelno izvršavaju omogućit će također i daljnje rastavljanje zadatka (Slika 2.12.b). Ovakvo „usitnjavanje“ rješenja vodi do kasnijih problema kod prelaska na tekstualne jezike jer se u spomenutom primjeru izbjegla petlja s uvjetom. Obzirom da se takav koncept teško usvaja, zapravo se u takvom okruženju propušta prilika za poučavanje (Meerbaum-Salant, Armoni, & Ben-Ari, 2011).



Slika 2.12. a) Ispravan program, b) Ekstremno rastavljanje zadatka

Ovaj primjer je zapravo u skladu s rezultatima istraživanja koje su proveli Aivaloglou & Hermans (Aivaloglou & Hermans, 2016) analizirajući 250.000 Scratch projekata. Naime, 77% projekata je sadržavalo petlje, ali samo 14% je sadržavalo petlje s uvjetom što znači da su većinom koristili petlju *ponavljaj* (eng. forever) kao u gornjem primjeru. Zaključak spomenute analize je kako je prevelika upotreba petlje *ponavljaj* zapravo utjecaj dizajna samog jezika (jer se tako može) no upitno je jesu li korisnici razumjeli koncept petlje.

Kod pristupa *podjezika* kombiniraju se oba prethodna pristupa jer uključuje podskup osobina „stvarnog" jezika tako da se podskup može tretirati kao mini-jezik. Na ovaj način bi početnicima dodatno mogli olakšati prijelaz u jezik opće namjene.

Učenje sintakse programskog jezika se na prvi pogled čini slično učenju stranog jezika, ali je ipak drugačije jer programski jezik ima dodatne apstrakcije koje ne povezuju sa stvarnim svijetom. Pored apstraktnog jezika učenici moraju svladati i okolinu u kojoj programiraju, razumjeti problem i tehnike rješavanja problema, oblikovati konačno rješenje (program) i testirati ga (Futschek, 2013). Trebalo bi započeti s malim i jednostavnim podskupom jezika koji ima vizualne elemente (Brusilovsky i ostali, 1994; Hu, 2004). Zadatak odabira sadržaja poučavanja programiranja, bilo da se odnosi na paradigmu ili konkretni programski jezik odmah za sobom povlači i odgovarajući pristup poučavanju programiranja.

2.3.3 Pristupi poučavanju programiranja

Fincher (1999) smatra da se klasični načini poučavanja previše oslanjaju na izvršavanje programa te predlaže razdvajanje programiranja i kôdiranja u obliku četiri pristupa koji se temelje na konceptualnim modelima i metodologiji poučavanja programiranja:

1. pristup bez sintakse (eng. syntax-free approach),
2. pristup pismenosti (eng. literacy approach),
3. pristup rješavanja problema (eng. problem solving),
4. računarstvo kao interakcija (eng. computation as interaction).

U *pristupu bez sintakse* koristi se pseudokod, olovka i papir te se na taj način smanjuje kognitivno opterećenje početnika jer ne moraju učiti programski jezik niti koristiti računalo sa složenim okruženjem za izradu programa.

Pristup pismenosti podrazumijeva upotrebu programa pisanih u stvarnom programskom jeziku koje treba znati čitati i razumjeti. Početnici čitaju i proučavaju već napisane programe od jednostavnijih do složenijih razmišljajući na koji način te programe mogu proširiti. Problem

ovog pristupa je što za neke početnike može poći puno vremena dok uspiju razumjeti program pa bi se moglo dogoditi da neki od njih ne mogu niti objasniti primjere, a pogotovo napisati novi program.

Pristup rješavanja problema temelji se na pedagogiji, odnosno ciklusu rješavanja problema u fazama: razumijevanje (analiza problema), oblikovanje rješenja, pisanje programa (implementacija) i provjera (testiranje i uklanjanje pogrešaka).

Pristup računarstva kao interakcije temelji se na ideji objektno orijentiranog pristupa gdje se svijet promatra kao kolekcija objekata koji „znaju“ nešto jedan o drugome te međusobno komuniciraju. Smatra se da se učenici rano upoznaju s karakteristikama računala kao višenitnim uređajima s grafičkim sučeljem te da je pogrešno takvim učenicima predstavljati model rješavanja u obliku jednonitne (eng. single thread) procedure ili kao niz izračuna.

Neki nastavnici biraju poučavanje programiranja pomoću računala, ali bez korištenja složenog razvojnog okruženja, odnosno rade u običnom uređivaču teksta, smatrajući da samo okruženje skriva ili komplicira proces programiranja (Raadt, Watson, & Toleman, 2002). Takvo što nije moguće ako želimo koristiti vizualne programske jezike jer su oni integrirani u okruženje, ali njihova okruženja u pravilu nisu složena tako da ne predstavljaju problem. S druge strane postoji mišljenje da stariji nastavnici biraju "jednostavniju varijantu" jer su oni također učili programirati bez složenih *integriranih razvojnih okruženja* (eng. integrated development environment, IDE) kao što je npr. MS Visual Studio.

Pattis (1993) tvrdi kako bi početnici odmah na početku trebali krenuti s učenjem osobina jezika (tipovi, osnovne naredbe i sl.), a tek onda učiti primjenu, odnosno oblikovanje rješenja, što je čest pristup u udžbenicima iz programiranja. Nakon toga bi trebali krenuti od programa koje je pripremio nastavnik, redom: čitanje, mijenjanje i pisanje. Razlog za ovakav pristup je što se smatra kako početnici ne mogu u potpunosti razumjeti sve koncepte programa ako ne razumiju kako funkcionira programski jezik u kojem je taj program napisan. U skladu s navedenim Pattis (1993) predlaže upotrebu biblioteke potprograma tako da bi početnici najprije naučili kako koristiti potprograme iz te biblioteke (upoznavanje s ulaznim parametrima i sl.), nakon toga bi učili kako od različitih potprograma složiti svoj program, a na kraju kako napisati vlastiti potprogram. Sve navedeno implicira smanjenje opterećenja programera početnika.

2.3.4 Kognitivno opterećenje u programiranju

Pri učenju općih koncepata programiranja rješavanjem problema intrinzično opterećenje povezano je s konceptima i interpretacijom, a irelevantno opterećenje određeno je programskim

jezikom i okruženjem, uključujući sva ograničenja koja tu pripadaju (npr. sintaksa jezika) (Raina Mason & Cooper, 2012). McCracken-ovo izvješće (McCracken i ostali, 2001) o neuspješnosti programera početnika na prvoj godini studija je često citirano, a deset godina nakon tog izvješća McCartney i ostali (2013) su ponovili eksperiment smatrajući kako su pitanja iz originalnog istraživanja bila teška za početnike te da se nije uzelo u obzir kognitivno opterećenje ispitanika. U ponovljenom istraživanju koristili su integriranu razvojnu okolinu (eng. Integrated Development Environment, IDE) te su dali studentima djelomično započete programe dozvoljavajući im pristup online dokumentaciji. Zaključili su kako su time bitno smanjili kognitivno opterećenje studenata. Vizualni programski jezici polaze od ideje snižavanja početnog praga kako bi početnicima olakšali prve korake u programiranju, a pripremom nekih početnih materijala u profesionalnim programskim jezicima želi se na neki način postići isti efekt.

Pokazalo se da je primjena *obrađenih primjera* (eng. worked examples) korisna za konstrukciju shema (Sweller, van Merrienboer, & Paas, 1998). Obrađeni primjer je demonstracija kako treba riješiti problem korak po korak (Hesser & Gregory, 2015). Cilj je pokazati učeniku pravilan postupak rješavanja. Obrađeni primjeri uključuju stanje problema (ulaz i uvjeti), ciljno stanje (izlaz) te korake rješenja (proces koji povezuje ulaz i izlaz), a pomažu učeniku pri razvoju općeg rješenja, odnosno stvaranja sheme i smanjivanja irelevantnog opterećenja. Obrađeni primjeri ne moraju sadržavati detaljne korake rješenja, a koriste se do trenutka kad se smatra da su učenici usvojili osnove novog materijala tako da mogu samostalno rješavati probleme (Hesser & Gregory, 2015). Gray i ostali (2007) također predlažu primjenu obrađenih primjera kod poučavanja programiranja kako bi se smanjilo kognitivno opterećenje, a van Merriënboer & De Croock (1992) su potvrdili hipotezu da djelomično dovršeni problemi smanjuju kognitivno opterećenje pri rješavanju matematičkih i programerskih problema. S druge strane, Sweller i ostali (1998) smatraju kako obrađeni primjeri omogućuju stvaranje stereotipova rješenja ili uzoraka (eng. pattern) što smanjuje kreativnost.

Mason & Cooper (2012) zaključili su kako 10% najslabijih učenika ulažu ogroman napor kod učenja (ekvivalentno velikom kognitivnom opterećenju), ali i dalje ne uspijevaju naučiti. Odabirom odgovarajućih jezika razvojnih okruženja nastoji se pomoći početnicima pri učenju smanjenjem irelevantnog kognitivnog opterećenja, a Mason & Cooper predlažu sljedeće:

- mentalni napor potreban za razumijevanje problema odnosi se na intrinzično opterećenje,

- mentalni napor potreban za upotrebu okruženja i sintakse programskog jezika odnosi se na irelevantno opterećenje,
- mentalni napor pojačavanja prethodno naučenih koncepata programiranja ili učenje iz problema odnosi se na povezano opterećenje.

Učenje programiranja smatra se složenim učenjem gdje se veliki broj informacija mora obraditi istovremeno (sintaksa, semantika, uklanjanje pogrešaka, strategije programiranja, rješavanje problema, okruženje, ...). Robins (2010) tvrdi kako je svaki koncept iz programiranja u bliskoj vezi s mnogo ostalih te prema tome učenje programiranja ima veliku interaktivnost elemenata. R. Mason & Cooper (2012) tvrde kako su početnici trošili ogromne količine kognitivnih resursa na irelevantno opterećenje kao što je učenje sintakse jezika te snalaženje u okruženju. Dvije visoke razine kognitivnog opterećenja znače da ne ostaje dovoljno resursa za preostalu razinu koja je ključna za stvaranje shema i spremanje u dugoročnu memoriju.

Van Merriënboer & Sweller (2005) ističu kako studenti koji koriste kognitivne strategije objašnjavanja rješenja ulažu više mentalnog napora te bolje rješavaju testove, a to povlači zaključak kako objašnjavanje utječe na višu razinu povezanog kognitivnog opterećenja.

Procesno-orijentirani (eng. process-oriented) obrađeni primjeri fokusirani su na proces kojim se dolazi do traženog rješenja (Van Merriënboer & Kirschner, 2018). Nastavnik obavlja složeni zadatak te istovremeno pojašnjava kako i zašto se tako rješava. Van Merriënboer & Kirschner takve primjere nazivaju *modelirajući primjeri* (eng. modeling examples).

Caspersen & Bennedsen (2007) predlažu inkrementalni pristup poučavanja programera početnika u objektno orijentiranom programiranju gdje započinju s jednostavnim zadacima prema složenijima uz obrađene primjere:

1. upotreba pripremljenih metoda u klasama,
2. izmjene postojećih metoda,
3. proširenje postojećih metoda,
4. stvaranje novih metoda,
5. stvaranje novih klasa,
6. stvaranje novih modela.

Također, Caspersen & Bennedsen integriraju različite teorije i tehnike kao što je CLT, poduprto učenje, obrađeni primjeri, inkrementalni pristup, pristup temeljen na modelu. Međutim, nisu

proveli formalnu evaluaciju i nisu koristili vizualnu sintaksu za smanjivanje kognitivnog opterećenja vezanog za sintaksu programskog jezika.

Margulieux i ostali (2012) su tijekom istraživanja primjene CLT u programiranju došli do ideje u kojoj predlažu odvajanje računalnog razmišljanja (eng. computational thinking) od sintakse programskog jezika upotrebom vizualnog programskog jezika *Android App Inventor* ili *Scratch*. Predložili su tri tehnike za smanjenje irelevantnog kognitivnog opterećenja:

1. obrađeni primjeri (eng. worked examples),
2. oznake podciljeva (eng. sub-goal labels),
3. poduprto učenje (eng. scaffolding).

Kaasbøll (1998) definira tri osnovna pristupa poučavanju programiranja, te jedan dodatni koji je definiran nakon razgovora s nastavnicima iz područja: semiotičke ljestve, taksonomija kognitivnih ciljeva, rješavanje problema, iterativni ciklus razvoja.

Semiotičke ljestve (eng. semiotic ladder) označavaju pristup koji je zasnovan na alatima koji se koriste (programski jezik, jezik za modeliranje i sl.). Redoslijed učenja je sljedeći: sintaksa, semantika, pragmatika. Sintaksa se odnosi na kombiniranje znakova u riječi (naredbe jezika), semantika na njihovo značenje, a pragmatika na primjenu. Svaki sljedeći korak ovisi o prethodnom tako da npr. bez sintakse nije moguće ništa iskazati.

Taksonomija kognitivnih ciljeva (eng. Cognitive objectives taxonomy) podsjeća na Bloom-ovu taksonomiju:

- Izvršavanje programa - učenik pokreće i primjenjuje gotovi program,
- Čitanje programa - učenik ima uvid u kod i prepoznaje i razumije konstrukte,
- Izmjena postojećeg programa - učenik može mijenjati i prilagođavati postojeći program,
- Stvaranje programa - učenik samostalno izrađuje program.

Svaki od navedenih ciljeva temelji se na prethodnom tako da na primjer učenik neće biti u stanju mijenjati program ako nije svladao fazu čitanja.

Iterativni ciklus razvoja temelji se na iterativnom razvojnom ciklusu programske podrške (eng. iterative software development cycle) koji se sastoji od faza:

1. analiza problema,
2. definiranje podataka,
3. pronalaženje sličnog programa (uzorka),

4. izmjena programa,
5. testiranje programa,
6. odabir metode testiranja,
7. provjera ispunjenosti zahtjeva.

Ciklusi se ponavljaju po potrebi opet od 1.

Bruce i ostali (2004) na temelju fenomenografskog istraživanja definiraju kategorije učenja: praćenje, kodiranje, razumijevanje i integracija, rješavanje problema, sudjelovanje. U kategoriji praćenje (eng. following) učenik prati strukturu nastavne jedinice nastojeći izvršiti pojedine zadatke koji su postavljeni, a koji utječu na ocjenu bez promatranja konteksta ili "šire slike". Učenik se trudi izvršiti ono što se od njega u danom trenutku traži ne povezujući pojedine dijelove u cjelinu što kasnije vodi do problema kod stvaranja složenijih programa. Kod faze kodiranja (eng. coding) učenik smatra da je najvažnije učenje sintakse, vježba sintaksu i memorira naredbe i dijelove koda kako bi što više naučio, te metodom "pokušaja i pogreške" nastoji uklopiti poznate dijelove koda u novi zadatak. Takvo učenje je površinsko, a posljedica je frustracija učenika kod nemogućnosti rješavanja složenijih problema jer učenik ne može razumjeti zašto dio koda koji je bio ispravan u jednom programu ne radi u drugom kontekstu. Razumijevanje i integracija odnosi se na gledanje šireg konteksta, učenici žele shvatiti, motivirani su i smatraju da se programi sastoje od sintakse, napisanog koda, ali i općih koncepata nevezanih za jezik. Rješavanje problema autori ovdje definiraju kao kategoriju koja ovisi o prethodnom razumijevanju i integraciji te se fokusira na planiranje rješenja. Sudjelovanje (eng. Participating) se odnosi na učenike (možda je bolje reći studente) koji se žele baviti programiranjem i visoko su motivirani. Oni se na primjer brinu i o čitljivosti programa, žele ne samo da radi već mora raditi brže, izgledati bolje, traže optimalno rješenje.

Kod naših studenata u nastavi često primjećujemo sljedeće: žele što prije početi kodirati preskačući planiranje. Ako se radi o vježbama i jednostavnim zadacima za koje se očekuje gotovi program, studenti žele čim prije početi raditi na konačnom obliku rješenja jer plan ne treba predati. Boljim studentima je motiv za preskakanje faze planiranja želja vidjeti odmah kako to radi dok lošiji gledaju kako bi si „skratili posao“ preskačući faze koje prema njihovom poimanju ne vode stvaranju konačnog rješenja odnosno kôda.

Za poučavanje programiranja početnika se koriste različite strategije (Yang, Hwang, Yang, & Hwang, 2015). Neki autori predlažu skrivanje detalja dok učenici ne budu spremni za njih, drugi predlažu i izrađuju različite računalne sustave, no možemo primijetiti kako se često

spominje učenje rješavanjem problema (eng. problem-based learning, PBL) kao pristup koji obećava prednosti u obliku motivacije, boljeg usvajanja koncepata, razvoja računalnog razmišljanja i sl. Ako smatramo da je programiranje samo po sebi rješavanje problema onda ne možemo izbjeći ni PBL.

2.3.5 Problemska ili projektna nastava

Naglasak kod odraslih učenika je na aktivnom učenju i sudjelovanju u nastavi (sudjelovanje u raspravama i učenje u skupinama uz suradnju) (McKeachie & Svinicki, 2013) gdje nastavnik preuzima ulogu voditelja i pomagača umjesto predavača. Henderson (1986) ističe kako su sposobnosti rješavanja problema i analitičko razmišljanje glavne slabosti studenata na informatičkim predmetima te da je ključno naučiti koristiti apstrakciju. Poučavanje moderne Net generacije (djeca rođena iza 1982 godine) može biti veliki izazov za nastavnika jer su zahtjevni, nestrpljivi i s ciničnim stavom prema autoritetu (Nilson, 2003). Međutim, cijene komunikaciju i kad su informirani o tome zašto i kako se nešto uči, te reagiraju drugačije na specifične situacije u odnosu na "tradicionalne" učenike. Primjerice, više će zamjeriti nedostatak iskustva nastavnika i nepripremljenost za nastavu. U početku visokog obrazovanja fakulteti su birali najbolje studente koji su morali biti visoko motivirani kako bi završili studij. Danas je cilj obrazovati što više ljudi čime dolazi i do širokog spektra različitih karakteristika učenika te samim tim i većih problema oko motivacije.

Nastava usmjerena na učenika odnosi se na nastavu u kojoj je učenik aktivniji od učitelja ili je aktivan barem u istoj mjeri kao i učitelj, te nikako ne obrnuto (Matijević, 2010). U situaciji kad je učitelj isključivo demonstrator jasno je da to nije moguće ostvariti već učitelj mora postati mentor, suradnik i organizator. Nije dovoljno da učenici uče površinski odnosno da memoriraju činjenice jer je to najniža razina Bloom-ove taksonomije. Matijević iznosi zanimljivu rečenicu:

"*Ima mnogo boljih medija za čuvanje i obradu informacija od dječjih glava!*".

U skladu s navedenim, predlaže se problemska i projektna nastava. Na taj bi se način učenici uključili aktivno uz rad (eng. learning by doing) ili istraživanje (eng. learning by discovery). Nedostatak nastave usmjerene na učenika što takva nastava može zahtijevati prilagođene prostore i opremu kao što učenje programiranja u principu zahtijeva računala radi praktičnog iskustva ili eventualno dodatne troškove ako bi se na primjer uključili i roboti.

U svim područjima ljudske djelatnosti su prisutni problemi, te je pojam *problem* nešto što se svakodnevno može čuti u komunikaciji. Problem je subjektivni doživljaj (spoznajni i emocionalni), odnosno može se reći kako čovjek *doživljava problem* (Poljak, 1990). Što je

problem veći, to su intenzivniji spoznajni i emocionalni doživljaji. Također, problem izražava određene suprotnosti: poznato - nepoznato, otkriveno - neotkriveno itd. koje čovjek doživljava kao nedostatak kojeg mora nadoknaditi ili prevladati upravo rješavanjem tog problema, a može biti odgovor na pitanje, izračunato rješenje i sl. U nastavi se često rješavaju znanstveni problemi (matematički, fizikalni i sl.). Tijekom školovanja bi učenike trebalo osposobiti za rješavanje problema iz stvarnog života što je bit *problemske nastave* (Poljak, 1990) ili učenja rješavanjem problema. Rješavanje problema (eng. problem solving) je jedan od najvažnijih tipova kognitivnih procesa koji se odvijaju tijekom učenja, a smatra se ključnim za učenje znanosti (eng. science) i matematike (Schunk, 2012). Odnosi se na napor kojeg učenici ulažu kako bi ostvarili cilj za kojeg nemaju unaprijed definirano ili automatsko rješenje. Premda „učenje“ i „rješavanje problema“ nisu sinonimi, rješavanje problema smatra se dijelom učenja. Bez obzira na sadržaj učenja ili područje te složenost, svi problemi imaju neka zajednička obilježja. Problemi imaju početno stanje koje se odnosi na trenutni status ili razinu znanja i sposobnosti učenika koji kreće s rješavanjem tog problema. Problemi imaju cilj, odnosno ono što učenik želi postići. Većina problema također zahtijeva od učenika rastavljanje na manje podciljeve, a ostvarivanjem tih manjih podciljeva dolaze do glavnog.

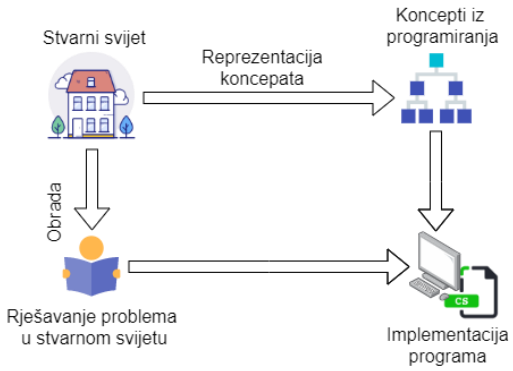
Učenje rješavanjem problema (eng. Problem-Based Learning, PBL) uključuje konstruktivističke ideje (Psycharis, Makri-Botsari, & Xynogalas, 2008). Osnovna ideja je učenje rješavanjem problema iz stvarnog svijeta, a dodatne bitne osobine takvog učenja su: učenje u kontekstu, pojašnjenje znanja kroz društvene interakcije (tijekom rada u skupinama), razvoj sposobnosti kritičkog načina razmišljanja (Fee & Holland-Minkley, 2010). Učenik mora biti aktivan sudionik procesa učenja i poučavanja. Prvi korak je poticanje učenika na rješavanje problema, odnosno potrebno je kod učenika razviti psihološku potrebu za samostalnim rješavanjem problema (Poljak, 1990). Rješavanje problema nije dio svakog procesa učenja, odnosno neće se događati kad je učenik dovoljno usavršio svoje vještine tako da automatski izvršava akcije koje ga vode do cilja ili rješenja (Schunk, 2012). Isto tako, neće se dogoditi na nižim razinama ili učenjem trivijalnih stvari kad učenici znaju što treba raditi kako bi naučili.

Uključivanje rješavanja problema u visoko obrazovanje evidentirano je 1966. godine na Sveučilištu Medical School (Nuutila, Törmä, & Malmi, 2005). Studenti medicine su morali usvojiti velike količine sadržaja, ali uz PBL su se koncentrirali na probleme ili situacije s kojima se liječnici u praksi svakodnevno susreću. Na taj način su mogli bolje integrirati usvojeno gradivo. Postoje različiti primjeri implementacije PBL, te se često spominje model od sedam koraka kojeg je 1983. godine predstavio Schmidt (Schmidt, 1983): razjasniti termine i koncepte

koji nisu razumljivi, definirati problem, analizirati problem (prema prethodnom znanju), sistematizacija objašnjenja iz prethodnog koraka analize, formuliranje ciljeva učenja (što se treba napraviti, podjela zadataka), prikupljanje dodatnih podataka van skupine (individualni rad), sinteza i testiranje novih informacija (skupiti i prezentirati sve iz prethodnih koraka).

Pristup je usmjeren na učenika, obično se odvija u malim skupinama gdje je učitelj voditelj (eng. facilitator) i organizator (De Graaf & Kolmos, 2003; Nuutila i ostali, 2005). Implementacija kurikuluma koji se temelji na PBL-u zahtijeva veliku angažiranost učitelja (Fee & Holland-Minkley, 2010). Često se događa da nastavnici osmisle probleme koje bi studenti trebali rješavati no ne uče ih tehnikama rješavanja problema već oni sami primjenjuju strategije koje su usvojili u prethodnim razinama školovanja. Jedna od takvih strategija koju možemo primijetiti kod studenata je strategija *pokušaja i pogreške* (eng. trial and error) kada studenti izvršavaju akcije dok ne dođu do rješenja. S vremenom studenti mogu zaključiti koje akcije brže dovode do rješenja, no ako na taj način rješavaju ispit onda jednostavno ne bude dovoljno vremena za rješavanje jer takva strategija nije pouzdana, a često ni učinkovita (Schunk, 2012). Može se dogoditi da nikad ne dođu do rješenja ili odustaju zbog frustracija.

Prema (Rogalski & Samurcay, 1990) programiranje kao i sposobnost rješavanja problema se može analizirati ili istraživati na dva načina. U prvom se koristi model iskusnog programera koji se uspoređuje s početnicima s ciljem analize početničkih pogrešaka i miskoncepcija. Drugi način se odnosi na sadržaj gdje se analizira poznavanje područja ili domene programiranja (koncepti, procedure, notacije, alati i sl.). Uspješno rješavanje problema različite složenosti može poslužiti kako demonstracija poznavanja područja programiranja. Slika 2.13. prikazuje jednostavnu shemu aktivnosti rješavanja problema u programiranju. Problem se može riješiti najprije u stvarnom svijetu tako da se obrada problema obavlja prije implementacije.



Slika 2.13. Rješavanje problema u programiranju, prema (Rogalski & Samurcay, 1990)

Drugi pristup je kad se krene iz programskog jezika prema stvarnom svijetu. Proces usvajanja koncepata u tom slučaju kreće od razumijevanja već napisanog programa te otkrivanja i uklanjanja pogrešaka. Često se i kod definiranja zadataka iz programiranja koje učenici rješavaju kreće od koncepata iz programskog jezika, na primjer: napišite metodu *Prosjek()* koja prima niz ocjena i računa prosjek.

Učenje rješavanjem problema u literaturi na engleskom jeziku često ima isti akronim PBL kao i *učenje izradom projekata* (eng. project-based learning), ali osim dijeljenja akronima također se znaju pomiješati termini *projekt* i *problem* kao i sami modeli nastave (Gallagher, Stepien, & Rosenthal, 1992; Thomas, 2000). Prema (Frank, Lavy, & Elata, 2003; Larmer, 2014) termin *projektno učenje* (eng. project learning) proizlazi iz teorije J. Dewey-a te predstavlja širi pojam od učenja rješavanjem problema. Dewey je smatrao kako škole i učionice trebaju reprezentirati stvarne životne situacije gdje će učenici učiti i rješavati probleme zajedno te da bi u fokusu trebao biti učenik, a ne sadržaj (M. K. Williams, 2017). Također, Slaughter (Slaughter, 2009) smatra kako bi trebalo iskoristiti prednosti tehnologije u nastavi jer se time može naglasiti društveni aspekt Dewey-jeve teorije učenja. Obzirom da radimo s odraslim učenicima, odnosno studentima, smatrat ćemo da su neke osnovne strategije rješavanja problema usvojili u prethodnim razinama školovanja, a neke specifične probleme koji se mogu pojaviti tijekom učenja prolaze na vježbama. Dovoljnim brojem uvježbavanja tj. rješavanja manjih problema ili zadataka na vježbama očekuje se automatsko rješavanje tipičnih problema koje studenti dobiju na ispitu. Psiholozi koji podržavaju *gestalt* pravac u psihologiji ističu kako naglasak na uvježbavanju (eng. drill and practice) i memoriranju rezultira trivijalnim učenjem te da se razumijevanje ostvaruje stvaranjem pravila za prepoznavanje koncepata (Schunk, 2012). Stvaranje pravila vodi boljem zadržavanju znanja ili retenciji (eng. retention) u odnosu na memoriranje jer pravila pomažu organizaciji znanja. Pravila su zapravo jednostavniji opis neke pojave tako da se pamti manje informacija. Na primjer, ako znamo da je volumen prizme i valjka, bilo uspravnih ili kosih, s bazom površine B i visinom v jednak $V = B * v$, onda ne moramo posebno pamtiti formule za kocku, kvadar, valjak i sl. Na taj način nećemo pamtiti više različitih formula bez razumijevanja.

Na početnim predmetima iz programiranja kao što je Programiranje II na PMFST čini se kako studenti dovoljnim uvježbavanjem uspiju riješiti probleme na ispitu, ali da ne ostvaruju dublje razumijevanje te vrlo brzo zaboravljaju naučeno. Prema tome, odlučili smo razmotriti nešto složenije što bi ih eventualno potaknulo i motiviralo, a to je uključivanje projekata u nastavu.

2.3.5.1 Učenje programiranja kroz izradu jednostavnih projekata

Projektna nastava (eng. project-based instruction) ili *učenje izradom projekata* (eng. project-based learning, PjBL) je model u kojem se učenje organizira oko izrade projekta. Kako bi izbjegli zabune oko akronima, neki autori za „project-based learning“ koriste akronim PjBL (Tseng, Chang, Lou, & Chen, 2013). Projekt je složeni zadatak koji se temelji na manjim problemima i izazovima, a uključuje učenike u oblikovanje, donošenje odluka, rješavanje problema, istraživanje i sl. (Thomas, 2000). Moursund (1999) ističe kako je za projekt važan autentičan sadržaj te da nastavnik ne bi trebao imati jak utjecaj na tijek izrade projekta već više biti u ulozi moderatora, voditelja ili mentora.

Thomas (2000) u pregledu PjBL zaključuje kako nema jedinstvene definicije ili modela te da je rezultat toga veliki broj raznovrsnih istraživanja. S druge strane, zbog toliko raznovrsnosti teško je procijeniti što točno čini projekt, a ponekad su razlike među primjerima projekata jače od sličnosti te nije lako definirati općenite karakteristike PjBL.

Premda mnogi tvrde kako koriste projekte u nastavi, mora biti jasno da se to ne bi trebalo odnositi na male zadatke koje na primjer studenti inače rješavaju tijekom vježbi već na širi opseg (J. Kay i ostali, 2000). Opseg projekata se često mora smanjivati zbog nedostatka vremena. Na primjer Nuutila i ostali (2005) su radili projekte manjeg opsega za vježbu isključivo individualno zbog potencijalne opasnosti rada u skupini kad svi sudionici ne sudjeluju jednako te možda čak samo jedan od njih odradi cijeli posao. U tom slučaju se gube sve prednosti rada u skupini te se javlja problem vrednovanja individualnog rada, odnosno ocjene svakog od učenika.

Tipovi projekata prema (De Graaf & Kolmos, 2003) mogu biti: *projekt sa zadaćama* (eng. task project), projekt s djelomičnom kontrolom i *problemski projekt* (eng. problem project). Projekt sa zadaćama detaljno planira učitelj, a učenik nema slobodu odlučivanja, unaprijed je poznat svaki korak. Učenici pri tome nemaju osjećaj da je to njihov projekt već nešto što im je nametnuto od nastavnika, a o tome moramo voditi računa radi gubitka motivacije. Prema tome, ovakav bi se pristup trebao izbjegavati. Projekt s djelomičnom kontrolom omogućuje nešto više slobode učeniku, ali je i dalje pod kontrolom učitelja jer ovisi o zahtjevima nastavnog plana i programa. Metaforički bi se mogao usporediti s nogometnom utakmicom gdje je teren unaprijed zadan uz neke minimalne upute vezano za pravila igre, ali utakmica nije počela, te igrači (metafora za učenike) moraju izaći na teren i započeti igru. Treći pristup je projekt u pravom smislu gdje učitelj ne planira aktivnosti u detalje i nije točno poznat izlaz. Tu se radi o

projektima otvorenog tipa pa prema tome svaka učenik ili skupina učenika može napraviti različit projekt. Početna faza planiranja može biti dosta vremenski zahtjevna.

Učenici stječu znanje tijekom istraživanja te postavljanjem pitanja potaknutih vlastitom znatiželjom (Bell, 2010). Rezultat projektne nastave je bolje razumijevanje, dublje znanje i povećanje motivacije. Učenici grade znanje na postojećim temeljima, odnosno znanju, te nauče više kad uče kroz rad, što odgovara idejama konstruktivizma. Curtis (2002) ističe kako projekti mogu poslužiti za motivaciju učenika do kojih je inače u tradicionalnoj nastavi teže doprijeti te da se takvom nastavom omogućuje ispunjavanje različitih interesa učenika i bolja retencija znanja jer se učenici više angažiraju na projektima koji iz zanimaju. Naravno, postoje i ograničenja kod primjene projekata kao na primjer: vrijeme koje je potrebno učenicima za izradu, vrijeme potrebno nastavnicima za pripremu i planiranje nastave, definiranje autentičnih projekata i pomaganje svakom učeniku (ili skupini učenika) oko njihovog jedinstvenog projekta, a naravno ne smijemo zaboraviti ni vrednovanje projekata.

Bell (2010) smatra kako je ključ uspjeha projektne nastave upravo mogućnost odabira teme projekta, obzirom da će na taj način učenici ispunjavati i razvijati svoje individualne interese. Tijekom izrade projekata učenici također sami biraju materijale ili izvore učenja koji su u skladu s njihovim sposobnostima. To potiče njihovu unutarnju motivaciju jer će raditi dulje i istraživati više. Vještine koje stječu tijekom izrade projekata se često ne može mjeriti uobičajenim ispitima znanja.

Thomas (2000) definira pet kriterija za lakše prepoznavanje što se može smatrati projektima u nastavi. Prvi kriterij je da projekt mora biti glavni pristup u kurikulumu. Projekti kao primjeri ili ilustracije, odnosno sporedni dijelovi kurikuluma se ne smatraju stvarnim projektima koji pripadaju projektnom učenju. Drugi kriterij je fokusiranje projekata na pitanja koja će poticati učenike na stvaranje poveznica između aktivnosti i koncepata koje usvajaju, a s tim je blisko vezan i treći kriterij koji se odnosi na poticanje konstruktivističkog istraživanja gdje učenici nadograđuju postojeće znanje. Četvrti kriterij odnosi se na angažman i kontrolu učenika, odnosno projekt bi trebao biti više pod njihovom kontrolom, umjesto da ga nastavnik precizno definira i radi scenarije. Posljednji kriterij podrazumijeva izradu „realističnih“ projekata tj. projekata koji imaju stvarnu primjenu u stvarnom svijetu ili rješavaju problem iz stvarnog svijeta ili je tema projekta nešto što je učenicima blisko.

Autentični projekti ili projekti otvorenog tipa (J. Kay i ostali, 2000) zahtijevaju autentični način vrednovanja (Moursund, 1999). Nastavnici moraju vrednovati projekte kako bi ocijenili rad

učenika, a istraživači u obrazovanju kako bi vrednovali učinak takve nastave u odnosu na druge vrste. Prema (Thomas, 2000) rezultat primjene projektne nastave može se ispitivati primjenom standardiziranih ispita znanja, definiranjem individualnih mjera, kombinacijom prethodna dva pristupa, fokusiranjem na stjecanje specifičnih vještina i samoprocjenom učenika (ankete). Boaler (1998) je nakon tri godine istraživanja primjene projektne nastave na učenje matematike utvrdila kako su učenici eksperimentalne skupine (projektna nastava) ostvarili bolje rezultate na državnim standardiziranim testovima u odnosu na kontrolnu (tradicionalna nastava). Osim toga, učenici eksperimentalne skupine su imali i pozitivnije stavove prema matematici, dok je učenicima iz kontrolne skupine matematika bila „dosadna i naporna“. Rosenfeld & Rosenfeld (1999) su tijekom svog istraživanja otkrili kako su učenici s lošijim ocjenama bili uspješniji na predmetima s projektnom nastavom, no učenici koji inače imaju bolje ocjene su bili manje uspješni na predmetima s projektnom nastavom nego inače. Zaključak je kako bi zapravo nastavu trebalo više prilagoditi učenicima, a bolje učenike poticati na razvijanje dodatnih sposobnosti.

Thomas (2000) također identificira nekoliko važnih aspekata projektne nastave: vrijeme (primjena projekata uvijek traje dulje od planiranog), kontrola (nastavnik mora balansirati količinu informacija koju daje učenicima u odnosu na ono što znaju ili su sami sposobni doznati), podrška (balansiranje razine podrške tako da im ne daju previše neovisnosti i premalo povratnih informacija), tehnologija (tehnologija koja pomaže pri izradi), vrednovanje (kako vrednovati projekte u kojima učenici moraju pokazati razumijevanje). Nastavnici i učenici se susreću s izazovima PjBL-a kao što su područno znanje nastavnika, nedostatak iskustva kod učenika (draže im je ono što je već poznato, kao što je uobičajeni pristup) i sklonost učenika prema pristupima koji zahtijevaju manje truda (Frank i ostali, 2003). Slično je i kod nekih studenata na našem fakultetu koji žele da im se točno definira što moraju naučiti, umjesto projekata u kojima moraju istraživati. Frank i ostali (2003) su primijetili kako je nastavnik kod njihovog istraživanja imao poteškoća pri prilagodbi svojih uobičajenih metoda poučavanja novom pristupu projektne nastave.

Ako nastavnici biraju projekt, onda ni njima nije lako jer projekt mora biti dovoljno blizak problemu iz stvarnog svijeta radi poticanja interesa učenika, ali ipak ne pretjerano složen da ga učenici nisu u stanju riješiti. Na primjer, izrada knjigovodstvenog sustava je projekt iz stvarnog svijeta, no da bi ga učenici mogli izraditi moraju razumjeti i knjigovodstvo. Osim odabira projekta, potrebno je strukturirati jednostavnije probleme kojima učenici vježbaju osnovne vještine prije početka rada na projektu, ali koje su ključne za izradu projekta (Soares, Fonseca,

& Martin, 2015). Projektna nastava dosta ovisi o profilu studenata, njihovim očekivanjima i interesima (De Graaf & Kolmos, 2003).

Alternativa traženju problema iz stvarnog svijeta je izrada jednostavnih igara ili simulacija. *Učenje zasnovano na igrama* (eng. game-based learning, GBL, gamification) se globalno može podijeliti na tri pristupa (Li & Watson, 2011; Mladenović i ostali, 2016):

- učenje kroz igranje igara (eng. Play-based approach),
- izradu igara (eng. Authoring-based approach),
- vizualizacijski pristup (eng. Visualization-based approach).

U prvom pristupu radi se o obrazovnim igrama, odnosno igrama izrađenima s ciljem poučavanja nekih koncepata, a u drugom pristupu je cilj usvajanje koncepata iz programiranja tijekom izrade igara (temeljeno na idejama konstruktivizma) pri čemu tipovi igara mogu biti različiti. Jedan primjer obrazovne igre izrađen pomoću OTTER okvira je na slici ispod (Slika 2.14). Učenici u toj igri pokušavaju dovesti robota do cilja navodeći ga jednostavnim naredbama za kretanje koje postoje i u programskom jeziku Logo: naprijed, okreni se lijevo, okreni se desno. Iako se čini da se igraju također uče i koncepte iz programiranja.

Slika 2.14. Primjer igre za poučavanje koncepata programiranja

Shabalina i ostali (2017) ističu dodatni pristup učenja programiranja kroz razvoj igara namijenjenih upravo učenju programiranja. Međutim za razvoj takvih igara potrebno je dosta složenih znanja i vještina iz područja: programiranja, programskog inženjerstva, oblikovanja igara i obrazovanja tako da taj pristup nije primjeren za početnike. Treći pristup s vizualizacijom je nešto između prethodna dva pristupa jer se ne može smatrati igrom (korisnik ne može igrati jer ne postoje osnovna obilježja igre kao što su npr. bodovi, priča, i sl.), a ne može se koristiti ni za izradu igre. Na taj se način vizualiziraju koncepti i izvršavanje koda dok

korisnik samo promatra izvršavanje. Korisnik u ovom pristupu donekle podsjeća na ulogu *igrača* u prvom pristupu.

Igre mogu imati pozitivan utjecaj na motivaciju (De-Marcos, Domínguez, Saenz-De-Navarrete, & Pagés, 2014), a kod pristupa gdje se igre rade prve (eng. Game-first approach) se kod nekih istraživanja pokazalo bolje usvajanje osnovnih koncepata programiranja kao i veće zadovoljstvo studenata (Leutenegger & Edgington, 2007). Igra u principu predstavlja zabavu, neki učenici shvaćaju izradu igara kao zanimljiv izazov, te misle da je cilj učenja izraditi igru. Zapravo kroz izradu igara uče razne koncepte iz programiranja (situacija kad učenik uči jedno dok radi nešto drugo naziva se "head fake"). Za izradu igara se mogu koristiti različite tehnologije jer postoji veliki broj različitih tipova igara kao što su logičke igre, slagalice, strategije, društvene igre itd. (Kirriemuir & McFarlane, 2004) pa mogu biti veliki izvor ideja za izradu samostalnih projekata.

Soares i ostali (2015) opisuju uspješnu primjenu PBL u početnom predmetu iz programiranja uz izradu igara. Odabrali su izradu projekata u obliku igre koja ima početne zahtjeve (npr. koliko nivoa mora imati, koje su osnovni elementi sučelja i sl.) koje učenik mora ispuniti. Učenici su bili motivirani završiti svoje igre, a autori su nakon provedene nastave zaključili kako početnicima nije problem usvajanje koncepata već njihova primjena. Također smatraju da je za poticanje interesa studenata potrebno pokazati dobre primjere s prethodnih godina kako bi učenici vidjeli što se stvarno može napraviti.

Djeca u dobi od 5-8 godina razvijaju asocijacije između konkretnih i simboličkih reprezentacija upotrebom manipulatora (eng. manipulatives) (Clements & McMillen, 1996). To mogu biti kartice, drvene kocke, ali i roboti (Elkin, Sullivan, & Bers, 2014). Sve što je prethodno spomenuto, ideje konstruktivizma, PBL i aktivno sudjelovanje u nastavi uz razvoj računalnog razmišljanja uklapa se u upotrebu robota kao manipulatora za učenje programiranja. Oblikovanje i planiranje robota, programiranje pri rješavanju zadanog problema može integrirati najmanje dvije STEM discipline. Upotreba robota pri učenju i poučavanju programiranja potiče aktivno učenje i suradnju unutar grupe koja radi na zajedničkom projektu odnosno rješavanju problema (Arlegui, Menegatti, Moro, & Pina, 2008). Često se temelji na konstruktivizmu, radu u skupinama ili timovima, sa ili bez aspekta natjecanja, učenje kroz igru, interdisciplinarno je (integrirani STEM) (Arlegui i ostali, 2008; Atmatzidou, Markelis, & Demetriadis, 2008; Micheli, Avidano, & Operto, 2008; Papanikolaou, Frangou, & Alimisis, 2008; Petrovič & Balogh, 2008), te omogućuje rad u paru (Alimisis, 2009).

Robotika omogućuje učenicima učenje o različitim konceptima (npr. senzori, motor i sl.) iz područja robotike (Bers & Horn, 2010), ali kako te robote mogu i programirati onda uče i koncepte iz programiranja. Učenici uz pomoć robota mogu dodatno učiti i koncepte iz matematike i PBL (Rogers & Portsmore, 2004). Učenici vide kako se robot ponaša, odnosno reagira na njihove naredbe, mogu se lakše "staviti na mjesto" robota, odnosno shvatiti zašto ne izvršava ono što su očekivali, te nakon toga ispraviti program. Slična ideja je kod programiranja Logo kornjače.

Roboti i igre čine se kao zanimljive ideje no ništa nije savršeno. Roboti su često skupi, tehnologija nepouzdana ili nepredvidiva (utjecaj baterije na rad motora ili proklizavanje na podlozi). Često se i od igara očekuje previše iako nema previše dokaza za njihovu učinkovitost (Connolly, Boyle, MacArthur, Hainey, & Boyle, 2012). Igre mogu utjecati na ishode učenja u dva oblika: motivacija i stav te znanje i vještine. Učenje igranjem igara može biti problem ako igre nisu dobro oblikovane. Bakar i ostali (2006) su za učenje pokušali koristiti komercijalne igre koje uopće nisu oblikovane za učenje. Takve igre mogu sadržavati neprikladne sadržaje, poticati negativna ponašanja i sl. Igre se u prvom redu igraju jer su zabavne, a ne radi nekog drugog cilja. Postizanje tog drugog cilja (učenja) je ono što nastavnik želi postići te je ponekad tu kontradikciju teško prevladati (Pohl, Rester, & Judmaier, 2009). Učenici su tijekom igranja koncentrirani na tijek igre što ih ometa u kasnijoj analizi svojih postupaka ili refleksiji što su zapravo naučili. Učenje programiranja izradom igara može također otići u pogrešnom smjeru kad početnici moraju usvojiti dodatne koncepte isključivo vezane za područje oblikovanja igara jer se time stvara dodatno irelevantno kognitivno opterećenje te se pomiče fokus s koncepata programiranja. Kod studenata na PMFST primijetili smo da neki zaista nisu zainteresirani za igre pa im to nije motivacijski faktor. U skladu s tim nije dobro forsirati takav tip projekata jer se time kod studenata koji nisu oduševljeni igrama stvara otpor i negativan stav. Početniku je teško rješavati problem kojeg ne razumije, a pogotovo kojeg ne želi razumjeti jer to zapravo nije cilj učenja predmeta kojeg treba položiti.

Nastavnici bi trebali osposobiti učenike za prijenos naučenog u novi kontekst ili nove situacije. *Prijenos rješavanja problema* (eng. problem solving transfer) događa se kad učenik na temelju postojećih znanja osmisli rješenje za novi problem (Mayer, 2002).

2.3.6 Poučavanje za prijenos

Primjena blokovskih jezika za poučavanje programiranja može poslužiti kao uvod u programiranje te na taj način postaviti temelj za kasniji prijenos ili prijelaz u tekstualni

programski jezik. Ostaje otvoreno pitanje da li se to stvarno događa onako kako nam se intuitivno može činiti da jest.

Prijenos se općenito odnosi na primjenu znanja na nove načine, u novim situacijama te u poznatim situacijama, ali u drugačijem kontekstu (Schunk, 2012). *Prijenos učenja* (eng. transfer of learning) se događa kad učenje u jednom kontekstu (ili sadržaju učenja) utječe na uspješnost u drugom kontekstu ili sadržaju učenja (Perkins & Salomon, 1992). Na primjer, osoba koja nauči voziti osobni automobil će lakše naučiti voziti kombi, učenje matematike priprema učenike za učenje fizike i sl. Proučavanjem prijenosa možemo doći do objašnjenja kako postojeće znanje utječe na novo učenje, a sposobnosti prijenosa su važne jer bi bez toga trošili previše vremena na poučavanje za svaku specifičnu situaciju. U obrazovanju je cilj omogućiti prijenos naučenih znanja i vještina u stvarni život, pogotovo na fakultetima koji obrazuju „učenike“ koji su korak do primjene naučenog u stvarnom zaposlenju. Ograničenja početnika u području organizacije i reprezentacije znanja te strategije rješavanja problema utječu i na sposobnost rješavanja problema, a samim tim i programiranja (Ormerod, 1990). Iskusni programeri mogu rješavati veći skup problema, a mehanizam koji im to omogućuje je upravo prijenos iz jedne domene u drugu. Prema tome, prijenos je iznimno važan za rješavanje problema i za programiranje.

Međutim, prijenos nije uvijek uspješan. Premda su mnogi oduševljeni Scratch-om i sličnim blokovskim jezicima, smatra se da je jedan od nedostataka Scratch-a što je težak prijelaz iz tog okruženja u tradicionalno bez nekog posrednog alata koji bi bio poveznica između koncepata koji se usvajaju u Scratch-u i načina na koji se ti koncepti mogu primijeniti u nekom drugom programskom jeziku gdje je važna sintaksa (Koorsse, 2012; Resnick i ostali, 2009) kao što je ideja s prijelazom iz Alice u Java programski jezik (Dann i ostali, 2012). Čak i kada samo promijenimo okruženje za rad, početnici se nađu u problemima (Kölling i ostali, 2003).

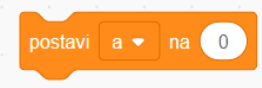
R. Lister (2011) iznosi mišljenje kako početnici koji uče programirati u okruženjima kao što su Scratch ili Alice mogu biti uspješniji na početku od onih koji se moraju boriti sa sintaksom i kompajlerom, ali dobar dio početnika će ipak u nekom trenutku dosegnuti maksimum svojih kognitivnih sposobnosti bez obzira na način na koji su počeli učiti programiranje. Mnogi će tvrditi kako početnici i dalje imaju iste probleme te će vjerojatno biti u pravu. Powers i ostali (2007) su koristili Alice u početnom predmetu iz programiranja tijekom jednog semestra. Pri poučavanju su odgodili primjenu varijabli tek za kraj semestra, no premda je Alice okruženje pomoglo kod nekih koncepata, nakon što su uvedene varijable pojavili su se uobičajeni problemi s tim konceptom koji su prisutni i u tekstualnim jezicima. Slično se dogodilo kod

prijelaza iz vizualnog okruženja *Karel the Robot* u tekstualni programski jezik Pascal (Clancy, 2004). Naime, početnici u vizualnom okruženju nisu upoznali koncept varijabli pa su kasnije imali problema kod procedura koje koriste parametre jer nisu razumjeli koncept varijabli.

Prijenos učenja može biti pozitivan i negativan (Perkins & Salomon, 1992). Pozitivan prijenos se događa kad učenje u jednom kontekstu poboljšava uspješnost u drugom kontekstu. Negativni prijenos se događa kad učenje u jednom kontekstu negativno utječe na učenje u drugom kontekstu. Na primjer, kod učenja stranog jezika učenik nastoji uklopiti novi jezik u svoj te se često događa da zadrži redoslijed riječi iz svog jezika. Negativni prijenos može biti problematičan, ali se ipak lakše ispravlja i to u ranim fazama učenja. Prema (Clancy, 2004) negativni prijenos u učenju programiranja može se odnositi na: engleski jezik, matematičke notacije, prethodno programersko iskustvo, modificiranje ispravnih pravila, netočno generaliziranje na temelju primjera te pogrešna primjena analogije. Primjer koji zbunjuje početnike je uvećavanje vrijednosti varijable za 1:

$$a = a + 1$$

Problem je što izraz u matematici nema smisla, ali u programiranju znači uvećavanje vrijednosti varijable *a* za 1, naravno ako podrazumijevamo da znak = predstavlja operaciju pridruživanja vrijednosti varijabli. Slično, ako napišemo: *3 = a*, to može biti prihvatljivo kod rješavanja matematičkih zadataka, ali u programiranju je zapravo sintaksna pogreška. U Scratch-u se problemi pridruživanja vrijednosti varijablama rješavaju posebnim blokovima naredbi u kojima se izbjegava znak =, npr.:



Slika 2.15. Pridruživanje vrijednosti varijabli u Scratch-u

U programskom jeziku Pascal se za pridruživanje vrijednosti koristi posebna kombinacija dvotočke i znaka jednakosti „:=“ kako bi se izbjegla zabuna s relacijskim operatorom. U programskom jeziku C# kao logički operator se koriste dva znaka „= =“.

Primjer netočnog generaliziranja na temelju primjera je kad studenti vide primjere klasa u kojima su sve varijable istog tipa te netočno zaključe da tako mora biti u svakoj klasi ili ako klasa ima samo jednu varijablu onda znaju pogrešno poistovjetiti objekt s varijablom sadržanom u klasi. Kod petlje *for* u Pythonu koristi se funkcija *range()* koja generira niz vrijednosti u

zadanom intervalu. Ono što nije jasno na prvi pogled je činjenica da upravo zbog *range* funkcije sljedeća dva programa ne ispisuju isti rezultat.

Python	C#

`for i in range(1,5):`
`  print(i)`
`  i = i + 1`
	`for (i=1; i<5; i++)`
	`{`
	`  Console.WriteLine(i);`
	`  i = i + 1;`
	`}`

Program napisan u Pythonu ispisat će brojeve: 1, 2, 3, 4, a program napisan u C# ispisat će brojeve: 1, 3. Funkcija *range* generira niz [1, 2, 3, 4], a petlja *for* u Pythonu „uzima“ vrijednosti za varijablu *i* iz tog niza, bez obzira na povećanje vrijednosti unutar tijela petlje.

Pogrešna primjena analogije je primjer varijable i kutije (Du Boulay, 1986). Varijabla je kutija, a kako kutija može sadržavati više stvari, onda se može pogrešno zaključiti da i varijabla može istovremeno sadržavati više vrijednosti.

Schunk (2012) također spominje i *nulti prijenos* (eng. zero transfer) koji se odnosi na situacije kad jedno učenje nije imalo nikakvog utjecaja na sljedeće.

Perkins & Salomon (1988) dijele prijenos učenja na dvije kategorije:

- bliski prijenos (eng. near transfer) i
- udaljeni prijenos (eng. far transfer).

Bliski prijenos se odnosi na prijenos među sličnim kontekstima. Na primjer kad student na ispitu dobije zadatke koji su slični zadacima s vježbi. *Udaljeni prijenos* se događa među kontekstima koji se na prvi pogled čine dalekima i različitima, manje je preklapanja. Pojednostavljeno, bliski prijenos je u sličnim situacijama, a daleki u novim i nepoznatim situacijama. Bliski prijenos se češće i lakše događa, ali ponekad se prijenos jednostavno ne dogodi iako nastavnici očekuju. Premda se neki prijenosi događaju jednostavno ili automatski često prijenos od učenika zahtijeva primjenu složenih vještina i sposobnosti te uvjerenje o korisnosti znanja kojeg usvaja (Schunk, 2012). Za uspješnost prijenosa je također bitna organizacija znanja u memoriji. Naime, ako su informacije u memoriji učenika međusobno dobro povezane veća je vjerojatnost da će se neki dio aktivirati u dugoročnoj memoriji te će biti moguće pozvati informacije koje su povezane s tim dijelom. Ako učenici samo memoriraju informacije bez smisla onda se teže događa aktivacija. Na primjer, studenti rješavaju zadatak gdje je potrebno napisati program koji

će izračunati aritmetičku sredinu ocjena te nakon nekog vremena dobiju zadatak u kojem je potrebno izračunati prosječnu cijenu proizvoda. Neki studenti neće shvatiti da se u principu radi o istom zadatku. Tu se radi o površnom učenju bez dubljeg razumijevanja. Za dublje razumijevanje je potrebno više vježbati rješavanje sličnih problema (Ormerod, 1990).

Postavlja se pitanje što nastavnici mogu učiniti kako bi potaknuli prijenos učenja. Jednostavan primjer je zadavanje domaćih zadaća gdje učenici vježbaju ono što su učili u školi. Uvježbavanjem se može riješiti dio problema oko prijenosa znanja, ali potrebno je uložiti dosta truda kako bi se omogućio učinkovitiji prijenos.

Perkins & Salomon (1988) definiraju dvije strategije poučavanja za poticanje prijenosa:

- premošćivanje (eng. bridging) i
- prigrljivanje (eng. hugging).

„Prigrljivanje“ se događa kad nastavnik stvara sličan kontekst učenja u kojem se prijenos očekuje. Na primjer, kad studentima da probni ispit koji je sličan stvarnome umjesto objašnjavanja kako se nešto rješava. Kod „premošćivanja“ nastavnik pomaže učenicima prenijeti naučeno u novi kontekst poticanjem apstrakcija, svjesnog traženja poveznica među kontekstima i sl. Prijenos ponekad ovisi o tome jesu li učenici uspjeli izdvojiti bitne atribute iz trenutne situacije (apstrakcija).

Odabirom vizualnog programskog jezika koji će biti početni jezik također uvjetujemo naknadno učenje i poučavanje programiranja. Primjerice, Alice i Greenfoot su zasnovani na Java programskom jeziku, te bi bilo logično da budu prvi susret s programiranjem, ako učenici kasnije trebaju učiti Javu. Svaki od navedenih primjera jezika ima određene prednosti ili nedostatke ovisno o kontekstu. Smatra se da Scratch nije početno zamišljen za poučavanje objektno-orijentiranog programiranja (OOP), dok Alice i Greenfoot predstavljaju pravi uvod u OOP obzirom da u potpunosti podržavaju OOP pristup (Utting, Cooper, Kölling, Maloney, & Resnick, 2010). Greenfoot je moćniji od Scratch-a primjerice za složenije izračune kod kojih je Scratch spor kao i Alice, ali je s druge strane početak rada ili učenja Greenfoot-a teži. Možda se taj početni trud isplati za kasnije učenje OOP, odnosno konkretno programskog jezika Java, kod Scratcha se učenici više mogu koncentrirati na usvajanje osnovnih koncepata programiranja kao što su slijed, grananje i ponavljanje (Böhm & Jacopini, 1966), no Greenfoot je ipak primjereniji starijim učenicima kojima je tekstualni dio jezika bliskiji. Papert je smatrao da programski jezik mora imati *niski prag* (eng. low floor) (odnosno jednostavan početak) i *visok strop* (eng. high ceiling) (mogućnost izrade složenih projekata tijekom vremena) (Resnick

i ostali, 2009). Dodatno se smatra da bi morao imati i *široke zidove* (eng. wide walls) kako bi podržavao različite interese i stilove. Nije ni malo jednostavno zadovoljiti sva tri uvjeta, ali se Scratch nastoji tome približiti. Nakon početnog rješavanja problema uz pomoć vizualnog programskog jezika, studenti u skladu s idejom *posredovanog prijenosa* (eng. mediated transfer) (Dann i ostali, 2012) prelaze na složeniji programski jezik.

Microsoft Visual Studio je korisničko okruženje kojeg studenti koriste pri programiranju u C#-u. Obzirom da je to također i profesionalno okruženje za razvoj, ima mnoštvo mogućnosti i samim tim *ometala* (eng. distractors) koji mogu zbuniti početnika. Stoga je potrebno početniku dati problem kojeg već zna riješiti u jednostavnom okruženju kao što je Scratch kako bi ga prenio u profesionalno okruženje tipa MS Visual Studio. Ako početnik kod učenja programiranja počne koristiti profesionalno okruženje, onda će kasnije uštedjeti vrijeme potrebno za privikavanje jer će s novim potrebama samo naučiti nove detalje iz okruženja koje mu trebaju (Rainalee Mason, 2012).

Scratch omogućuje usvajanje koncepata paralelnog izvršavanja skripti (Meerbaum-Salant i ostali, 2013) (eng. concurrency) jer tako likovi funkcioniraju. Primjerice, ako želimo da papiga istovremeno maše krilima i kreće se, jedna skripta će se baviti mahanjem, a druga pomicanjem lika. To je jako jednostavno riješiti u Scratch-u, no problem paralelnog izvršavanja odnosno *višenitnih aplikacija* (eng. multithreading) u C# je jako složen, te spada u napredno programiranje. Studenti bi mogli koristiti Scratch i za brzu izradu prototipa igre.

Vizualni programski jezici kao što je npr. Scratch, Byob i sl. omogućuju učenicima različitih dobi koncentriranje na rješavanje problema bez opterećenja sa sintaksom. Takvi jezici se mogu primijeniti i na visoko obrazovanje za početni predmet iz programiranja (Krpan, Mladenović, & Zaharija, 2014).

Kod nekih istraživanja pokazuje se rast vještina rješavanja problema tijekom igranja igara, ali i prijenos tih vještina na različite probleme u igri. Međutim, prijenos vještina van igre može biti problematičan (Egenfeldt-Nielsen, 2006), a utvrđene razine prijenosa niske (Egenfeldt-Nielsen, 2007; M. Lister, 2015). Studenti za razliku od djece brzo "prerastu" vizualno programsko okruženje Scratch-a koje kod složenijih zadataka postaje nepregledno jer se primjerice ne može pretraživati. Međutim, da bi se mogao ostvariti prijenos naučenih koncepata iz programskog jezika tipa Scratch u moćniji programski jezik kao što je npr. C#, potrebno je na neki način intervenirati u ciljno tekstualno okruženje kako bi se lakše mogli identificirati koncepti koji se prenose. Na slici (Slika 2.16) je prikazana usporedba koda napisanog u Scratch-u i C#.


```
private void Pozornica_GreenFlagClickedEvent()
{
    sprite.IdiKoraka(50);
    sprite.OkreniSe(90);
}
```

Slika 2.16. Usporedni prikaz vizualnog i tekstualnog jezika

Programski kôd u oba programska jezika izgleda dovoljno slično kako bi početnici mogli pokušati nastaviti raditi u tekstualnom okruženju. Jedini problem u navedenom je što zapravo standardni programski jezik C# nema klasu Sprite niti metode za kretanje. Kako bi to mogli napraviti, potrebno je uvesti dodatne stvari, kao što su gotove biblioteke koje podržavaju grafiku i kretanje likova, biblioteke koje će pripremiti nastavnik pa ih dati studentima ili prepustiti studentima da sami rade sve iz početka.

2.3.7 Didaktičko skrivanje

Razvoj tehnologije i povećanje količine znanja stvara probleme u obrazovanju jer se kapacitet ljudskog mozga neće povećavati kako bi se uskladio s novom količinom sadržaja, a da bi mogli odgovoriti na to potrebno je prilagoditi i organizirati nastavne materijale na drugačiji način. Također, poučavanje apsolutnih početnika programiranju stvara potrebu za pronalaženjem načina kako im to složeno područje pojednostavniti i približiti te je li takvo što uopće moguće.

Didaktička redukcija (eng. didactic reduction) je termin koji potječe iz literature na njemačkom jeziku, a odnosi se na pojednostavljivanje materijala za poučavanje (Grüner, 1967; Mesch, 1994). Smatra se korisnom za učenje složenijih koncepata iz programiranja kao što su strukture podataka ili modeliranje (Futschek, 2013). Konkretan termin nije puno prisutan u literaturi na engleskom jeziku, ali Futschek smatra da ostali pojmovi ne pokrivaju u potpunosti ideju. Programeri početnici imaju dosta stvari koje moraju svladati kao što je već više puta spomenuto te se nastoji reducirati detalje koje im je teže shvatiti kako bi se smanjilo kognitivno opterećenje. Učenicima se daje problem kojeg pokušavaju riješiti i smisliti svoj algoritam na temelju postojećeg iskustva, umjesto da analiziraju te nastoje memorirati i shvatiti algoritme koje je smislio netko drugi jer se to često zapravo svodi isključivo na memoriranje.

Didaktičku redukciju možemo podijeliti na kvantitativnu i kvalitativnu (Mesch, 1994). Kvantitativna znači smanjivanje broja koncepata ili izostavljanje dijela materijala. Kvalitativna se dijeli na horizontalno i vertikalno skrivanje. Kod horizontalnog skrivanja sadržaj postaje konkretniji ili se prezentira u obliku analogije, a kod vertikalnog skrivanja prikazuju se samo

mali dijelovi složenog sadržaja. U inženjerstvu to znači da ćemo složeni sustav prikazati pomoću podsustava, a detalji se izostavljaju ovisno o razini apstrakcije. Postupak pojednostavljivanja mora zadržati valjanost osnovnog značenja originalnog koncepta tako da se trenutno reducirani dijelovi mogu naknadno dodati (Futschek, 2013). Nastavnicima je poznato kvantitativno izbacivanje nekih koncepata iz predmeta, no problem je procjena što će se izbaciti, a što zadržati. Izostavljanje detalja koji čine neki koncept može dovesti do površnog učenja.

Pojam „didaktička redukcija" prema gornjim podjelama više se zapravo odnosi na sadržaje koje ćemo trajno izbaciti iz trenutnih materijala, tako da ćemo pojmom „didaktičko skrivanje" označiti metodu privremenog skrivanja sadržaja od učenika, odnosno smanjenje složenosti dok učenici ne postanu u stanju usvojiti složenije detalje. Ako se neka teorija pojednostavljuje iz didaktičkih razloga treba paziti da koncepti budu i dalje ispravni i da ne stvaraju miskoncepcije (Mesch, 1994). Kod nižih uzrasta je potrebno stvari pojednostavniti i reducirati, a kod starijih (studenata) se više oslanjati na apstrakciju i didaktičko skrivanje. Od studenata se očekuje sposobnost prijenosa znanja među različitim područjima pa tako i prijenos iz obrazovnog sustava u stvarni život. Budući programeri će morati pratiti i brze promjene tehnologije.

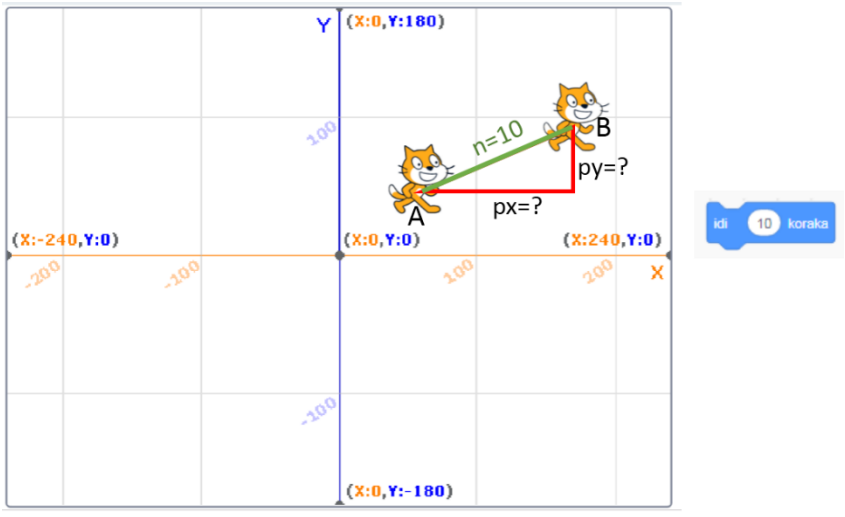
Djelomično na tragu skrivanja je i korištenje obrađenih primjera (Sweller i ostali, 1998), ali uz prijelaznu fazu *djelomičnih obrađenih primjera* (eng. fading worked examples) (Gray i ostali, 2007; Hesser & Gregory, 2015). Tu se radi o izostavljenim koracima u primjerima koje studenti sami nadopunjavaju, a ideja se temelji na učenju pomoću potpore (eng. learning from scaffolding). *Potpora* (eng. scaffolding) u učenju programiranja odnosi se na pomoć koja se pruža početniku kako bi riješio problem, izvršio zadatak ili ostvario cilj koji je inicijalno iznad njegovih sposobnosti i što zapravo ne bi mogao uspjeti bez pomoći (Kunkle, 2010; Michaelson, 2018). Zapravo se kontroliraju složeni elementi koje početnik teže može razumjeti kako bi ostvario cilj koji u danom trenutku može ostvariti. Programer početnik uči kako obaviti zadatak i kad se potpora ukloni. U početku se pojam potpore koristio za opis pomoći koju pruža roditelj ili tutor, no sve se više koristi za opis potpore koju učenik dobiva od programskih alata, kurikuluma ili drugih sredstava koja se koriste u učenju i poučavanju. Nastavnik cijelo vrijeme mora procjenjivati sposobnosti i razumijevanje učenika kako bi mogao prilagoditi razinu potpore te tu potporu s vremenom postupno uklanja. U tom trenutku učenik treba biti sposoban sam generalizirati proces te ga primijeniti na neki drugi zadatak.

Smanjivanje irelevantnog kognitivnog opterećenja je važna metoda, ali potrebno je obratiti pažnju i na složenost sadržaja jer van Merriënboer & Sweller (van Merriënboer & Sweller,

2005) navode kako nema potrebe smanjivati irelevantno opterećenje kod sadržaja s niskom razinom interaktivnosti obzirom da postoji dovoljno kognitivnih resursa za učenje takvih sadržaja. Problem je kod sadržaja s visokom interaktivnosti, odnosno sadržaja koji imaju veliko intrinzično opterećenje ili jednostavnije rečeno: kod jako složenih sadržaja.

Sweller (1994) smatra kako se intrinzično opterećenje ne može mijenjati, no par godina kasnije van Merriënboer & Sweller (van Merriënboer & Sweller, 2005) utvrdili su smanjenje intrinzičnog kognitivnog opterećenja pomoću metoda kao što su: „od jednostavnog prema složenom“ i „od dijela do cjeline“. U prvoj metodi radi se o napredovanju od pojednostavljenih verzija problema prema složenijima, a u drugoj se započinje od djelomično započetog zadatka kojeg treba dovršiti (van Merriënboer & De Croock, 1992). To su umjetne metode kako smanjiti interaktivnost elemenata, a temelje se na pretpostavci da studenti u ranoj fazi učenja ne moraju znati sve detalje niti koliko je zapravo sadržaj složen.

Na primjer, pomak lika na zaslonu računala za *n* točkica može biti jednostavna naredba *Idi (n) koraka* kao što je u Scratchu, bez ulaska u detalje. No da kad bi sami morali pisati funkciju za pomak, onda bi analizirali što imamo i što moramo dobiti. Ako je poznata točka A (x_1, y_1) gdje se lik trenutno nalazi i *n* ili broj točkica koje mora prijeći, onda moramo izračunati koordinate ciljne točke B(x_2, y_2). Koordinate ćemo dobiti „jednostavno“ nakon što izračunamo pomak po *x* i po *y* osi (px i py sa slike, Slika 2.17), samo nam treba malo trigonometrije.



Slika 2.17. Pomak lika u Scratchu

Xinogalos (Xinogalos, 2015) izdvaja nekoliko problema koji se pojavljuju u literaturi o miskoncepcijama početnika u vezi osnovnih koncepata objekta i klase: teško razlikuju pojam klase od instance odnosno objekta, smatraju kako je klasa skup ili kolekcija objekata (a ne

predložak ili apstrakcija), smatraju objekt dijelom klase (npr. ako je klasa računalo, onda je za njih objekt tipkovnica), stvaranje programa koji uključuju više od jedne klase im je ponekad zbunjujuće te ponekad ne shvaćaju da klasa modelira entitet stvarnog svijeta. Početnici ne shvaćaju u potpunosti ni koncepte učahurivanja i konstruktora (Fleury & Fleury, 2000; Sanders & Thomas, 2007). Također, neki početnici ne razumiju koncept ili svrhu *pristupnih metoda* (eng. accessors) (Garner, Haden, & Robins, 2005). Garner i ostali (Garner i ostali, 2005) su uočili probleme s osnovnim tehnikama (eng. basic mechanics) koje se odnose na osnove sintakse i strukture programa. S druge strane Sanders & Thomas (Sanders & Thomas, 2007) nisu primijetili probleme s osnovnim tehnikama, a koristili su konkretne zadatke koje su studenti trebali riješiti. Garner i ostali (Garner i ostali, 2005) također uočavaju probleme rada s iznimkama i *tijekom kontrole* (eng. flow of control) u aplikacijama vođenima događajima (eng. event driven).

Nakon što svladaju sintaksu tijekom predmeta Programiranje II, mogli bi očekivati kako studenti na predmetu Objektno orijentirano programiranje imaju manje problema s osnovnim tehnikama, no to je i dalje prisutno jer neki jednostavno zaborave nakon položenog ispita ili upisuju OOP kao izborni predmet bez prethodnog predznanja. Prilikom rješavanja konkretnih zadataka na kolokviju neki studenti još uvijek pokazuju te probleme predajući neispravne programe. Iz tog razloga se prvi dio vježbi na predmetu OOP posvećuje prolasku kroz sve ključne koncepte spomenute ranije uključujući podsjećanje na sintaksu i strukturu C# programa.

Dodatan problem je „učenje prema predlošku ispita“ odnosno rješavanje konkretnih ispitnih zadataka s prethodnih godina umjesto usvajanja koncepata objektno orijentirane paradigme. Studenti koji se pripremaju za kolokvije na taj način imaju poteškoće jer kod malih izmjena ispitnog zadatka više nisu u stanju taj zadatak riješiti. Na primjer, računanje prosjeka godina studenata i računanje prosječne cijene autobusnih karata smatraju različitim zadacima iako su konceptualno isti. Klasični kolokvij na OOP kojim se ispituju osnovne tehnike je bitan jer se time potiče studente da se prisjete sintakse i osnovnih tehnika, obzirom da ulazi u ocjenu, a kasnije se prilikom izrade projekata mogu koncentrirati na druge elemente. Obzirom da su takve ispite i kolokvije studenti prošli i tijekom predmeta Programiranje II, a i dalje imaju iste probleme postavlja se pitanje što napraviti. Sanders & Thomas (Sanders & Thomas, 2007) su detaljno čitali i analizirali programski kôd studenata kako bi uočili koncepte koje su studenti primijenili, ali kod zadavanja konkretnih ispitnih zadataka je teško pripremiti odgovarajući tekst zadatka tako da studenti znaju što točno nastavnik očekuje bez da se eksplicitno napiše.

Ponekad studenti riješe zadatak bez primjene objektno orijentiranih koncepata koje nastavnik želi ispitati tako da svedu problem na prethodno usvojeno znanje. Rješenje će ispisati traženi rezultat, ali time se ne dokazuje primjena očekivanih koncepata. Također, uočili smo da se na taj način više ispituje prepoznavanje pojmova nego smislena primjena što spada u najnižu razinu Bloomove taksonomije.

Studenti opterećeni konceptima jezika i okruženja se teško koncentriraju na koncepte objektno orijentirane paradigme. S druge strane, koncentriranje na koncepte paradigme uz pomoć posebnih alata ili jezika namijenjenih isključivo poučavanju ponekad vodi do nezadovoljstva dijela studenata koji ne vide primjenu tih alata u kasnijem zaposlenju te bi radije naučili neki od komercijalnih jezika (Spigariol, 2016).

Obzirom da je cilj predmeta Programiranje II usvajanje osnova sintakse objektno orijentiranog programskog jezika uz osnovne tehnike, cilj predmeta Objektno orijentirano programiranje je usvajanje i primjena ostalih ključnih koncepata. Zbog svega navedenog bilo je potrebno promijeniti pristup poučavanja na predmetu OOP.

3 OKVIR ZA POUČAVANJE OBJEKTNO ORIJENTIRANOG PROGRAMIRANJA

Kölling i ostali (2003) smatraju da poučavanje objektno orijentiranog programiranja samo po sebi nije složenije od drugih paradigmi već nedostaju odgovarajući alati te pedagoško iskustvo s paradigmom. Programiranje može biti zanimljivo i kreativno iskustvo. Važno je postaviti odgovarajuću razinu izazova, potaknuti interes i znatiželju kod studenata. Učenje zasnovano na igrama (eng. game-based learning, GBL) smatra se sredstvom kojim se to može ostvariti (Futschek, 2013). Međutim, obzirom na različite profile studenata koji se upisuju na PMFST, ne možemo očekivati da će svi biti jednako oduševljeni s igrama. U svakom slučaju, vizualni elementi i interaktivnost prisutna u igrama ima potencijala i za izradu drugih tipova projekata kao što su simulacije ili poslovne aplikacije, ali problem je odabir odgovarajućeg okruženja odnosno okvira za razvoj koji sadržavaju primjenjive elemente bez dodatnog opterećivanja studenata sa složenim konceptima računalnih igara. Okviri za izradu igara (eng. game development framework, GDF) često ne odgovaraju kriteriju jednostavnosti i usklađenosti s ciljevima predmeta. Premda radovi opisuju kako se igre koriste za poučavanje ili kako se vizualni elementi igara mogu koristiti za poticanje interesa studenata, nema dovoljno dokaza o stvarnom utjecaju na učenje u odnosu na klasično poučavanje (Jenkins, 2002).

Obzirom da je cilj kolegija OOP naučiti objektno-orijentirane koncepte, a ne izradu igara niti arhitekturu igara, onda zapravo složeni GDF-ovi kao što su Microsoft XNA ne odgovaraju za tu namjenu. Učenje kako se koristi takav okvir, koje su osnovne karakteristike okruženja i sl. je zapravo nemoguće bez usvajanja koncepata igara, što je u suprotnosti sa smanjivanjem kognitivnog opterećenja. Motivacija ipak ne može svima biti toliko velika da bi se svladali problemi oko opterećenja.

Vizualni programski jezici kao što je Scratch se također mogu smatrati okvirima za razvoj jednostavnih 2D igara jer su dosta jednostavni za učenje, ali postavljaju ograničenja na vrstu projekata koji se u njima mogu izrađivati te također nisu vezani za ni jedan profesionalni tekstualni programski jezik. Prijenos iz vizualnog programskog jezika u tekstualni nije jednostavan niti se događa prirodno (Mladenović i ostali, 2016).

Wang & Wu (2009) su identificirali tri načina primjene okvira za razvoj igara u nastavi. Nastavnici mogu koristiti projekte umjesto tradicionalnih vježbi kako bi povećali motivaciju studenata te samim tim i njihovu angažiranost u nastavi. Integriranjem jednostavnih projekata u predavanja može se povećati sudjelovanje i motivacija studenata. Također, nastavnik može

tražiti od studenata da razviju jednostavan projekt (igru, simulaciju, poslovnu aplikaciju) kao dio kolegija te bi na odabranom okviru mogli naučiti i vještine softverskog inženjerstva.

Istraživanja predlažu različite pristupe poučavanju kako bi učenje i poučavanje koncepata OOP-a bilo ugodnije i učinkovitije. Kako bi izbjegli miskoncepcije, primjeri i zadaci za vježbu moraju se pažljivo oblikovati (Holland, Griffiths, & Woodman, 1997). Poučavanje bi trebalo početi uvođenjem koncepta objekata koji imaju uočljivo ponašanje i interakciju s drugim objektima (Proulx, Raab, & Rasala, 2002) Nastavnik bi na početku trebao koristiti primjere istraživanja koja omogućuju prikaz složenih primjera kao i mogućnost usvajanja objektno orijentiranog pristupa rješavanju problema (Nevison & Wells, 2003). Nedostatak odgovarajućih alata i iskustva u poučavanju može biti izvor zaključka da je poučavanje OOP teško (Kölling i ostali, 2003).

Prilikom istraživanja postojećih okvira pronađen je i okvir za izradu igara pod nazivom *Otter2d.* Poklapanje naziva je slučajnost, a implementacija i osnovne ideje se prilično razlikuju. Otter2d također koristi C# i .NET okruženje, ali temelji se na biblioteci *SFML* (eng. Simple and Fast Multimedia Library) (Pulver, 2019). Biblioteka SFML olakšava razvoj igara i multimedijskih aplikacija, a napisana je u C++ jeziku. Otter2d je okvir namijenjen upravo za izradu igara, a trenutno aktualna verzija 1.0.0 sastoji se od 103 *cs* datoteke, 129 klasa te ukupno oko 17000 naredbenih linija. Za usporedbu, OTTER ima 7 *cs* datoteka, 8 klasa i oko 700 naredbenih linija. OTTER radi isključivo na temelju elemenata prisutnih u Windows Forms .NET aplikacijama koje su studenti učili tijekom predmeta *Programiranje II*. Za razliku od našeg jednostavnijeg OTTER okvira, Otter2d se može koristiti za napredniji razvoj igara, a to je ono što se ovdje želi izbjeći. Boaventura & Sarinho (2017) predlažu minimalni okvir za izradu igara, no ako pogledamo detaljni prikaz okvira, sama struktura sadrži dvadesetak klasa koji su isključivo koncepti vezani za igre (npr. Camera, ControlMap, PhysicsNode, …). Sam okvir je možda prikladan za manje igre, ali smatramo da ideja ipak previše naglašava taj tip projekata.

Različiti alati su razvijeni za poučavanje objektno orijentiranog programiranja i oblikovanja, a Georgantaki & Retalis (Georgantaki & Retalis, 2007) su iz literature izdvojili nekoliko kategorija u koje se ti alati mogu razvrstati: programiranje mikrosvijetova, obrazovna okruženja za programiranje, alati za proširenje mogućnosti okruženja za programiranje, alati za podršku pristupa „objekti prvi“, okruženja za igranje (eng. gaming environments) i alati temeljeni na specifičnim alatima za poučavanje. U kategoriji mikrosvijetova nalazimo alate koji se temelje na alatu „Karel the Robot“ (Pattis, 1981) koji je u početku bio namijenjen proceduralnom programiranju do pojave verzije Karel++. *Alice* je također primjer okruženja koje prati tradiciju

Karel okruženja, ali dodaje još jednu dimenziju (Utting i ostali, 2010). Prikaz animacija u 3D omogućuje početnicima jaču vizualizaciju objekata. Poznati alat koji spada u obrazovna okruženja za programiranje (eng. educational programming environments) je BlueJ u kojem se struktura objektno orijentiranog programa prikazuje grafički u obliku dijagrama (Kölling i ostali, 2003). BlueJ je okruženje za razvoj namijenjenu učenju programskog jezika Java, a također je i implementirano u istom jeziku. *Greenfoot* je također alat za poučavanje objektno orijentiranog programiranja namijenjen studentima ili srednjoškolcima (Kölling, 2008). Pri oblikovanju Greenfoot-a koristile su se ideje postojećih alata kao što je vizualni dio iz „Karel the Robot“ okruženja ili interakcija objekata iz BlueJ.

Primjer okruženja za igranje namijenjenog učenju objektno orijentiranog programiranja u programskom jeziku Java je *Robocode,* a to je zapravo obrazovna igra u kojoj je potrebno programirati tenkove koji su u interakciji s drugim tenkovima (Hartness, 2004). Tijekom izrade takvih igara potrebno je balansirati između obrazovanja i zabave jer se može dogoditi da učenici kojima su takve igre namijenjene budu uspješni u igranju, ali ne usvoje znanja koja su bila očekivana (Wong, Yatim, & Tan, 2014). Slično se može primijeniti i na poučavanje objektno orijentiranog programiranja izradom igara obzirom da se lako mogu izgubiti ciljevi učenja jer studenti koji izrađuju igre mogu imati problema oko samih koncepata igre, načina igranja, interakcije likova, ciljeva igre i sl. čime će izgubiti iz fokusa koncepte objektno orijentiranog programiranja.

3.1 Koncepti objektno orijentiranog programiranja

Konstruktivistički pristup je prikladan za prijelaz iz proceduralne paradigme u objektno orijentiranu jer se oslanja na učenikovo postojeće znanje iz programiranja, odnosno osnovne koncepte programiranja koji su prisutni i kod objektno orijentirane paradigme (na primjer slijed, grananje, ponavljanje, varijable i sl.). Važno je da učenik aktivno sudjeluje u izgradnji objektno orijentiranog znanja, a s konstruktivističkog stajališta razlikujemo tri tipa znanja koja su važna za objektno orijentirano programiranje (Hadjerrouit, 1999): objektno orijentirani koncepti (klase, objekti, nasljeđivanje, …), objektno orijentirani jezik za implementaciju i problemi koje treba riješiti (moraju biti usko vezani uz koncepte objektno orijentiranog programiranja. Sva tri tipa znanja moraju biti povezani, a da bi to bilo moguće nije dovoljno samo znati kodirati već veliku ulogu igraju analiza i oblikovanje. Početnici se koncentriraju na uske dijelove vezane isključivo za pisanje programa i sintaksu, a teško im je rješavati nove probleme kad nisu u stanju prepoznati i primijeniti objekte iz prethodnog problema.

Tradicionalni pristup, odnosno pristup kojeg je u početku koristila većina je poučavanje proceduralne paradigme u početnim predmetima iz programiranja, tj. poučavanje koje obično kreće od tri osnovna algoritamska koncepta slijeda, grananja i ponavljanja (Böhm & Jacopini, 1966; Rainalee Mason, 2012; Meerbaum-Salant i ostali, 2013). Prema toj ideji objekti se uvode tek kasnije. Jednostavan razlog za takav pristup je što se proceduralna paradigma pojavila prije objektno orijentirane pa je na neki način to bio „prirodni" redoslijed poučavanja.

Studenti koji se obrazuju za buduće nastavnike informatike, a istovremeno su i početnici, odnosno kao odrasle osobe se prvi put susreću s programiranjem, moraju usvojiti ne samo osnovne već i napredne koncepte u relativno kratkom vremenu od jednog semestra po predmetu, te ih naučiti prenijeti na druge. Vrijeme koje studenti imaju na raspolaganju je relativno kratko jer prema istraživanjima (Robins i ostali, 2003; Winslow, 1996) potrebno je bar deset godina da bi netko od programera početnika postao stručnjak ili iskusan programer. Kod obrazovanja nastavnika zapravo nije cilj od njih stvoriti inženjere već ih naučiti konceptima programiranja, kao i primjerima dobre prakse koje će moći primijeniti u svom budućem radu. Studenti na Prirodoslovno-matematičkom fakultetu koji u trećem semestru upisuju predmet *Objektno orijentirano programiranje* (OOP) su tijekom prvog semestra upoznati s osnovnim algoritamskim strukturama i proceduralnim pristupom, te uvodom u sintaksu C# tijekom drugog semestra.

Objektno orijentirano programiranje (OOP) se dugo smatralo naprednim sadržajem te se poučavalo tek kasnije u kurikulumu. Trend se mijenja, obzirom i na često citiran problem pomaka paradigme (eng. paradigm shift) (IEEE-CS & ACM, 2001). Objektno orijentirana paradigma je danas dosta raširena te je poznato kako je poučavanje OOP-a prilično problematično. Postojeća istraživanja pokazuju kako postoji mnogo problema s kojima se susreću nastavnici prilikom poučavanja OOP-a (Carlisle, 2009; Chen & Cheng, 2007; Decker & Hirshfield, 1993; Kölling, 1999a, 1999b; Kölling & Rosenberg, 2001). Studenti koji su već upoznati s nekom paradigmom, npr. proceduralnom, imaju poteškoće s prijelazom na objektno-orijentiranu paradigmu jer sadrži koncepte koji se preklapaju, ima nove koncepte te zahtijeva drugačiji način razmišljanja i organizacije kôda (Bennedsen & Schulte, 2007).

3.1.1 Definiranje koncepata objektno orijentiranog programiranja

U proceduralnoj paradigmi, programi se pišu kao niz naredbi u glavnom dijelu programa. Naredbe koje su povezane, odnosno zajedno čine neku smislenu cjelinu obično se grupiraju u jedan blok kôda koji se naziva procedura ili funkcija. Kada se procedura pozove, izvršavaju se naredbe koje joj pripadaju. Pisanje kôda unutar procedura poboljšava čitljivost koda te smanjuje

broj ponavljanja istih naredbi ili blokova naredbi. Prema tome, u proceduralnoj paradigmi glavni fokus je na definiranju procedura i određivanju kad ih treba pozvati.

Za razliku od proceduralnog programiranja, OOP dodaje razinu logičke apstrakcije te umjesto procedura koristi koncept objekta kao osnovnog elementa za izgradnju programa. Objekt u OOP je logička reprezentacija nekog entiteta kojeg programer koristi u svom programu, a obično se sastoji od nekoliko različitih dijelova. Objektno orijentirani programi se ne izvršavaju sekvencijalno već ovise o događajima (eng. event-driven), a fokus je na interakciji među objektima.

Kramer i ostali (2016) na temelju pregleda literature zaključuju da je kod učenja OOP potrebno usvojiti znanja i vještine vezane za osnovne koncepte (strukture podataka, klase i strukturu objekata, algoritamske strukture), usvojiti neki formalni sustav za opis (sintaksa i semantika programskog jezika), rješavati probleme te biti motiviran.

Armstrong (2006) na temelju pregleda literature iz razdoblja od 1966-2005 izdvaja ključne koncepte objektno orijentiranog programiranja: nasljeđivanje, objekti, klase, učahurivanje, metode, slanje poruka, polimorfizam i apstrakcija te na temelju toga stvara taksonomiju.

Tablica 3.1. Taksonomija osnovnih elemenata objektno orijentiranog razvoja (Armstrong, 2006)

Konstrukt	**Koncept**	**Opis**
Struktura	Apstrakcija	Izdvajanje klasa s ciljem pojednostavljivanja aspekata stvarnog svijeta
	Klasa	Opis organizacije i akcija zajedničkih jednom ili više sličnih objekata
	Učahurivanje	Oblikovanje klasa koje ograničavaju pristup podacima i ponašanju definiranjem ograničenog skupa poruka koje objekt može primiti
	Nasljeđivanje	Podaci i ponašanje jedne klase su uključeni u drugu klasu ili su osnova za drugu klasu
	Objekt	Individualni element kojeg je moguće identificirati, može biti stvaran ili apstraktan, sadrži svoje podatke i načine manipulacije tim podacima
Ponašanje	Slanje poruka	Objekt šalje podatke drugom objektu ili traži pokretanje metode drugog objekta
	Metoda	Način pristupa, postavljanja ili manipulacije informacijama sadržanima u objektu
	Polimorfizam	Različite klase mogu reagirati na istu poruku, a svaka od njih s drugačijom implementacijom

Pretraživanjem literature koja se bavi objektno orijentiranom paradigmom sigurno ćemo naći različita objašnjenja koncepata objektno orijentirane paradigme. Objektno orijentirani koncepti se predstavljaju na različite načine koji ovise o kognitivnim zahtjevima prema početnicima, načinu modeliranja (npr. mape koncepata, UML), programskim jezicima (npr. Python, Java, C#), pseudokodu ili formalnom matematičkom prikazu (M. Kramer, Hubwieser, & Brinda, 2016). U nastavku ćemo navesti nekoliko primjera objektno orijentiranih koncepata koji ovise o načinu implementacije pa ih u ovom trenutku nećemo definirati.

Na primjer, Sanders & Thomas (2007) su „ručno“ analizirali studentske programe, što znači da su svaki program više puta čitali kako bi utvrdili koriste li studenti objektno orijentirane koncepte koje su željeli ocijeniti. Studenti su dobivali zadatke pa je bilo potrebno u skladu sa zadacima provjeriti u kôdu jesu li studenti instancirali odgovarajuće objekte klase, da li se instance podklasa ponašaju po principima polimorfizma, jesu li studenti napisali klase koje nasljeđuju od postojećih klasa i sl. No, ovdje treba napomenuti da su programi pisani u Javi koja dozvoljava višestruko nasljeđivanje (nasljeđivanje od više klasa) dok primjerice u C#-u to nije moguće. Hubwieser & Mühling (2011) su također ispitivali implementaciju objektno orijentiranih koncepata, ali su umjesto programskog jezika koristili program za vizualizaciju (yEd) u kojem su studenti crtali mape koncepata koje dozvoljavaju puno više slobode u odnosu na programski jezik. Objektno orijentirani programski jezik mora programeru omogućiti definiranje novih klasa te također implementaciju koncepata objektno orijentirane paradigme, ali to nije uvijek jednostavno. Način definiranja svakog koncepta prilično ovisi o odabranom programskom jeziku. Na primjer, konstruktor u programskom jeziku Python uvijek ima isti naziv: *__init__*, dok se u C#-u mora zvati jednako kao i klasa. Nadalje, ako promatramo način implementacije preopterećenih konstruktora u Javi i C#-u onda bi mogli reći da preopterećenje konstruktora u Pythonu nije podržano, no postoje tehnike kojima se to može napraviti. Problem je što ne postoji jedinstven način na koji se to radi te je zapravo jednostavnije reći da mora biti jedan konstruktor. Ako nastavnik želi analizirati primjenu koncepta preopterećenog konstruktora u kôdu, onda će analiza programa pisanih u Pythonu biti složenija.

Zbog svega navedenog zaključujemo kako koncepte objektno orijentiranog programiranja moramo promatrati u odgovarajućem i konkretnom kontekstu pa ćemo u skladu s tim u ovoj disertaciji definirati promatrane koncepte objektno orijentiranog programiranja i njihovu reprezentaciju u programskom jeziku C# koja je bitna za prepoznavanje koncepata u programskom kôdu te kasniju analizu. Za koncepte programskog jezika koji ovdje nisu eksplicitno definirani vrijede pravila i definicije programskog jezika C#. Za razliku od gore

spomenutog istraživanja (Sanders & Thomas, 2007), projekti u okviru ove disertacije nisu zadani zadaci s precizno definiranim zahtjevima pa ne možemo ocjenjivati provjeravanjem zahtjeva. Također, studenti su morali predati ispravne projekte (projekte bez sintaksnih pogrešaka).

Snyder (1986) navodi da je za OOP ključno: definicija klasa (ili tipova objekata), akcije (ili operacije) definirane nad objektima, akcije koje vrijede za više različitih tipova objekata i da definicije klasa dijele zajedničke karakteristike pomoću nasljeđivanja.

Klase predstavljaju opis jednog ili više objekata ili skup objekata koji dijele zajedničku strukturu, a ako ih usporedimo s proceduralnim jezicima koncept klase odgovara tipu podataka (Stefik & Bobrow, 1985). Možemo je smatrati nacrtom ili predloškom prema kojem se stvara objekt. Definiranjem klase organiziramo informacije o tipu podatka kojeg možemo ponovo koristiti. U skladu s navedenim definiramo koncept klase.

Definicija 3.1 *Klasa.*

> Klasa predstavlja korisnički definirani tip podatka. Klasa u C#-u započinje ključnom riječi *class* iza kojeg slijedi naziv klase i tijelo klase. Tijelo klase predstavlja blok koda omeđen vitičastim zagradama. □

Sintaksa:

```
class NazivKlase
{
    //tijelo klase
}
```

Članovi klase koji opisuju objekt odnose se na podatke, odnosno njihovo spremanje i dohvaćanje (polja i svojstva), a članovi koji definiraju ponašanje su metode klase. Koncept metode je nastao pojavom objektno orijentirane paradigme, odnosi se na dohvaćanje, postavljanje i manipulaciju podacima objekta (Armstrong, 2006). Sličan je konceptu procedure iz proceduralne paradigme (ili funkcije), ali je metoda pripada isključivo objektu ili klasi. Svaka klasa definira skup imenovanih akcija koje se mogu izvršavati nad instancama klase, a svaka akcija implementira se u obliku metode koja ima pristup poljima klase, odnosno može postavljati i čitati vrijednosti pojedinog polja za konkretnu instancu (Snyder, 1986). Prvi korak u definiranju koncepta metode kako ga vidimo u C#-u je definiranje razine pristupa pomoću modifikatora.

Definicija 3.2. *Modifikator pristupa u tijelu klase.*

Modifikator pristupa kontrolira pristup varijabli ili metodi unutar klase. Može biti *public* (pristup varijabli/metodi dozvoljen je iz svih klasa), *private* (pristup varijabli/metodi dozvoljen samo unutar dane klase) i *protected* (pristup varijabli/metodi dozvoljen je unutar klase te svih klasa koje od nje nasljeđuju). □

Modifikator pristupa *public* u C#-u se koristi i za klase, ali u kontekstu studentskih projekata ih nismo promatrali jer studenti nisu učili niti pisali ugniježđene klase, a njihova rješenja su se sastojala isključivo od jednog projekta.

Definicija 3.3. *Metoda.*

Metoda je imenovani blok kôda koji se sastoji od potpisa i tijela metode. Potpis metode sastoji se od modifikatora pristupa, povratnog tipa, naziva metode i popisa parametara u okruglim zagradama. □

Sintaksa:

```
modifikator tip Naziv(popisParametara)
{
}
```

popisParametara može biti prazan ili se sastojati od deklaracija varijabli.

Definicija 3.4. *Polje klase.*

Polje klase je varijabla deklarirana u tijelu klase. Može biti *public* ili *private*. □

Definicija 3.5. *Svojstvo klase.*

Svojstvo klase koristi se za ograničavanje pristupa privatnim poljima klase, a sastoji se od ključne riječi *public*, naziva svojstva i tijela. Tijelo svojstva sadrži *pristupne čvorove* (eng. accessor nodes), a to su: *get* i *set*. Zadatak *get* dijela je dohvaćanje sadržaja privatnog polja klase, a zadatak *set* dijela je spremanje vrijednosti u privatno polje klase.□

Definicija 3.6. *Jednostavna svojstva.*

Jednostavna svojstva su svojstva koja u tijelu sadrže pristupni čvor *get* koji u svojem tijelu ima najviše *return* naredbu i *set* koji u svojem tijelu ima najviše naredbu pridruživanja vrijednosti (ključna riječ *value*). □

Sintaksa:

```
public tip NazivPolja
{
```

```
    get {return nazivPolja;}
    set {nazivPolja = value;}
}
```

Definicija 3.7. *Složena svojstva.*

Složena svojstva za razliku od jednostavnih u tijelima pristupnih čvorova *get* i/ili *set* sadrže više naredbi, odnosno u tijelu od *get* pored *return* naredbe najmanje još jednu i/ili u tijelu *set-a* pored naredbe pridruživanja najmanje još jednu. □

Ako svojstvo ima samo *get* onda se može samo čitati, a ako ima samo *set* onda se može samo pisati. Studenti u principu nisu radili previše složena svojstva, tako da je bilo dovoljno utvrditi postojanje dodatnih naredbi.

Definicija 3.8. *Konstruktor klase.*

Konstruktor klase je posebna metoda klase koja za razliku od metode klase (Definicija 3.3) nema tip. □

Zadatak konstruktora klase je postavljanje početnih vrijednosti polja klase, ali nije nužno pisati programski kôd u tijelu konstruktora. Često se dogodi da studenti ne napišu ništa u tijelu konstruktora. Sam konstruktor se isto tako može izostaviti. Ako se napiše više konstruktora s istim nazivom, onda su oni preopterećeni (eng. overloaded).

Definicija 3.9. *Članovi klase.*

Članovi klase su metode, polja klase, svojstva i konstruktori. Članovi klase nalaze se u tijelu klase. □

Objekti su osnovni entiteti ili instance klasa koji se stvaraju tijekom izvršavanja programa. Koncept objekta uveden je u programskom jeziku Simula (Armstrong, 2006). Definiran je svojim stanjem, ponašanjem i identitetom. Objekt je konkretan primjerak ili instanca klase.

Definicija 3.10. *Objekt.*

Objekt je instanca klase, a ima tri obilježja: identitet, stanje i ponašanje. Identitet objekta služi za identifikaciju objekta. Stanje objekta određeno je vrijednostima njegovih polja, a ponašanje je definirano metodama klase kojoj objekt pripada. □

Definicija 3.11. *Apstraktna klasa.*

Apstraktna klasa je klasa koja se ne može instancirati. Koristi se kao predložak za stvaranje drugih klasa. U C#-u se identificira pomoću ključne riječi *abstract* koja se piše prije definicije klase. □

Definicija 3.12. *Statička klasa.*

> Statička klasa započinje ključnom riječi *static*, ne instancira se, a članovi statičke klase su uvijek statički. Članovi statičke klase pripadaju klasi, a ne instanci klase. □

Članovi ne-statičke klase isto tako mogu biti statički, ali se u tom slučaju mogu pozvati kao dio klase te nije potrebno stvarati instance. Studenti koji koriste statičke klase u svojim projektima pokazuju razumijevanje kad je potrebno instancirati objekte, a kad nije. Za razliku od apstraktne klase, članove statičke klase možemo koristiti, odnosno pozivati po potrebi.

Enkapsulacija omogućava skrivanje pojedinih elemenata klase, a po potrebi pruža pristup putem javnih metoda ili svojstava.

Definicija 3.13. *Enkapsulacija.*

> Enkapsulacija ili učahurivanje je proces ograničavanja pristupa članovima klase. U studentskim projektima se identificira kroz korištenje modifikatora *private, protected* te svojstava. □

Enkapsulacija omogućuje ponovnu upotrebu dijelova kôda bez ugrožavanja sigurnosti. Ako želimo izmijeniti postojeći kôd skrivenih elemenata onda to smijemo napraviti jer neće imati utjecaja na dijelove kôda koji koriste javne elemente. Obzirom da svojstva prema svojoj definiciji ograničavaju pristup poljima klase, jasno je da će biti uključeni u koncept enkapsulacije. Isto tako je potrebno napomenuti kako MS Visual Studio omogućava automatsko generiranje dijelova kôda (tzv. snippet) što od korisnika ne zahtijeva poznavanje sintakse niti posebno razumijevanje te se tek pisanjem dodatnog kôda može bolje shvatiti primjerna. Prema tome, potrebno je razdvojiti jednostavna svojstva (Definicija 3.6) od složenih (Definicija 3.7).

Armstrong (2006) u svojoj taksonomiji definira apstrakciju kao stvaranje klasa s ciljem pojednostavljenja stvarnosti pomoću specifičnosti koje postoje u problemu koji se rješava. Apstrakcija je na neki način kao proširenje enkapsulacije jer je usko vezana uz definiranje bitnih obilježja koja će biti skrivena ili javna. Korisnik neke klase će vidjeti ono što je bitno. Na primjer, korištenje mobilnog telefona za neke osnovne stvari je jednostavno: poziv, primanje i slanje poruka, no ono što se događa u pozadini je skriveno i nije bitno korisniku mobitela. Promjena implementacije slanja poruka za krajnjeg korisnika ne znači puno ako će njegovo sučelje ostati isto.

Definicija 3.14. *Apstrakcija.*

Apstrakcija podrazumijeva oblikovanje klasa pri čemu se izdvajaju bitna obilježja klase (stanje i ponašanje). U studentskim projektima se identificira prema definiranim klasama, poljima i metodama.□

Nasljeđivanje polazi od ideje da različite klase imaju slične komponente ili dijelove te da bi izbjegli ponavljanje kôda, povezujemo osnovnu klasu s izvedenim klasama. Klasa čiji članovi se nasljeđuju naziva se *osnovna* (eng. base) ili *roditelj*, a klasa koja nasljeđuje od osnovne naziva se *izvedena* (eng. derived) ili *dijete*. Koncept nasljeđivanja omogućuje ponovnu upotrebu kôda (Stefik & Bobrow, 1985).

Definicija 3.15. *Izvedena klasa.*

Izvedena klasa nasljeđuje od osnovne klase. Definicija izvedene klase u C#-u iza naziva klase sadrži dvotočku i naziv klase od koje nasljeđuje. □

Sintaksa:

```
class NazivIzvedeneKlase : NazivOsnovneKlase
{

}
```

Kad jedna klasa nasljeđuje od druge onda je njihov odnos drugačiji te se javlja potreba za blažim ograničenjima pristupa, ali posljedica je narušavanje prednosti enkapsulacije jer se smanjuju mogućnosti izmjena u klasi (Snyder, 1986). Programer više ne može jednostavno mijenjati nazive ili implementaciju polja kojima mogu pristupati izvedene klase jer izmjene izravno utječu na implementaciju izvedenih klasa. Snyder (1986) ističe kako bi svakako trebalo zaštititi polja instanci od izravnog pristupa te im pristupati isključivo putem akcija (ili operacija). To se odnosi na metode ili svojstva, ovisno da li jezik podržava svojstva ili ne. Programski jezik C# omogućuje implementaciju blažeg ograničenja pristupa pomoću modifikatora *protected* (Definicija 3.2).

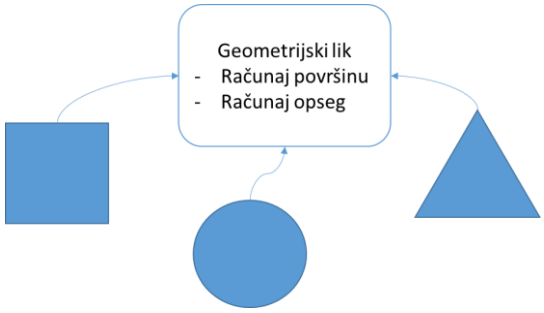
Definicija 3.16. *Nasljeđivanje.*

Nasljeđivanje je proces korištenja članova jedne klase kao osnove za drugu klasu. U studentskim projektima koncept nasljeđivanja određuje izvedena klasa, definiranje apstraktne klase, računanje maksimalne dubine nasljeđivanja, virtualna svojstva i pozivanje konstruktora osnovne klase. □

Konstruktor osnovne klase C#-a poziva se u izvedenoj klasi pomoću ključne riječi *base*. Višestruko nasljeđivanje podrazumijeva nasljeđivanje od više osnovnih klasa (više roditelja).

Kao što je već spomenuto u primjeru, C# ne podržava višestruko nasljeđivanje pa ga nećemo definirati. Moguće je definirati *sučelja* (eng. interface) što dijelom podsjeća na višestruko nasljeđivanje, ali studenti u svojim projektima nisu koristili sučelja. Jednostavno za manje projekte nije postojala svrha definiranja sučelja.

Polimorfizam označava mogućnost promjenjivosti oblika. Koncept polimorfizma povezan je s nasljeđivanjem. U objektno orijentiranom programiranju na primjer omogućuje definiranje više metoda s istim nazivom, ali različitim tipovima parametara. Drugi primjer je kad u osnovnoj klasi definiramo neke metode, ali izvedena klasa sadrži drugačije implementacije tih metoda. Na primjer, ako imamo klasu „GeometrijskiLik“ onda ona može sadržavati predloške metoda za računanje opsega i površine, ali svaki od likova (Slika 3.1) ima svoje implementacije.



Slika 3.1. Polimorfizam

Programer će u C#-u *premostiti* (eng. override) metodu osnovne klase.

Definicija 3.17. *Preopterećene metode.*

> Metode koje imaju iste nazive, ali se razlikuju po parametrima nazivaju se preopterećene metode (eng. overloaded). □

Obzirom da su konstruktori isto metode, onda i za njih vrijedi isto. U studentskim projektima se pronalaze metode i konstruktori s istim nazivima. Korisnik mora znati naziv metode i tipove podataka koje metoda prima.

Definicija 3.18. *Premošćene metode i svojstva.*

> Premošćene metode i svojstva imaju iste nazive u izvedenoj i osnovnoj klasi, ali im se implementacija razlikuje. Premošćene metode/svojstva u studentskim projektima sadrže ključnu riječ *override.*□

Definicija 3.19. *Polimorfizam.*

Polimorfizam označava sposobnost izvršavanja akcije istog naziva na različite načine u različitim kontekstima. U studentskim projektima se manifestira se na dva načina: primjenom preopterećenih metoda i premošćivanja.

Polimorfizam je vidljiv kod preopterećenja metoda, ali i operatora. Možemo smatrati i da je operator + preopterećen jer ako se koristi s brojčanim tipom podataka onda kao rezultat operacije dobijemo zbroj dva broja (2 + 3 = 5), a ako se koristi s tekstualnim podacima onda kao rezultat operacije dobijemo spojeni tekst ("Dobar" + "dan" = "Dobardan"). Moguće je definirati i vlastito preopterećenje operatora, ali to studenti nisu koristili tako da se ne analizira.

Potrebno je vrijeme kako bi studenti usvojili OO koncepte te ih koristili u projektima. Studenti nauče pisati dijelove koda u nekom OO jeziku, ali njihovo razumijevanje i praktična primjena OO koncepata općenito nisu na zadovoljavajućoj razini (Xinogalos, 2009). Neki istraživači smatraju da je čak OO paradigma prirodnija jer se objekti iz programa mogu povezati s objektima iz stvarnog svijeta (Livovský & Porubän, 2014). Međutim, prilično je teško mapirati osnovne koncepte OOP (npr. klase, objekte, atribute, metode, nasljeđivanje, polimorfizam i učahurivanje) sa stvarnim projektima (Yan, 2009). Alati koji se koriste za poučavanje OOP uključuju odgovarajuća okruženja kao što su mikrosvjetovi, vizualizacije i interaktivnost (Kölling, 2008; Milne & Rowe, 2002; Robins i ostali, 2003). Činjenica je da dominiraju pristupi poučavanju koji su usmjereni prema jeziku (eng. language-oriented approaches) uz nadopunu tehnikama oblikovanja koje se koriste u industriji te tek malo pedagogije.

3.2 OTTER

Kako bi odgovorili na prethodno spomenute izazove, odlučili smo oblikovati novo okruženje za razvoj jednostavnih projekata s grafičkim elementima nazvan OTTER (ObjecT orienTed lEarning fRamework) koji je zasnovan na metodi didaktičkog skrivanja (Krpan, Mladenović, & Zaharija, 2019).

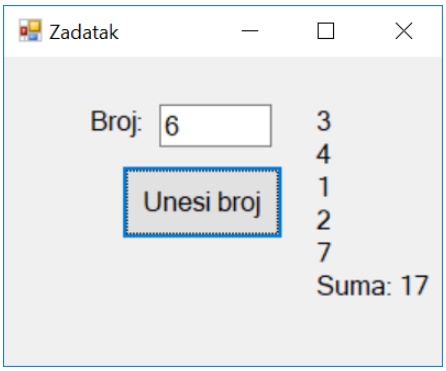
Najvažniji zadatak nastavnika koji koristi OTTER je primjena didaktičkog skrivanja što je manje moguće i samo onoliko koliko je potrebno. Okvir mora biti otvorenog kôda kako bi studenti samostalno po potrebi mogli ukloniti pojednostavljenja kad budu spremni za korak dalje. Iz perspektive nastavnika, upotreba OTTER okvira ne zahtijeva veći napor kod pripreme nastavnih materijala u odnosu na uobičajenu pripremu. Ako bi počeli iz početka, onda je vrijeme pripreme otprilike slično onome kao što to inače radimo, a ako već postoje pripremljeni materijali za drugačiji oblik nastave (bez okvira) onda ne bi trebalo biti teško prilagoditi postojeće lekcije novom okruženju obzirom da su koncepti koje se želi poučavati isti. Zadatak

nastavnika je povezati odgovarajuće koncepte koji se mogu pripremiti u okviru s objektno orijentiranim konceptima koje studenti trebaju usvojiti.

Hadjerrouit (1999) definira nekoliko principa poučavanja objektno orijentiranog programiranja:

- objektno orijentirano znanje učenici moraju aktivno konstruirati,
- razvoj programa mora biti vođen konceptima objektno orijentiranog programiranja umjesto osobinama jezika,
- treba paziti da prethodno znanje studenata ne bude u konfliktu s objektno orijentiranim pristupom,
- objektno orijentirani koncepti se moraju povezati s problemskim situacijama i jezikom,
- tradicionalna nastava se mora zamijeniti aktivnostima u kojima su studenti aktivno uključeni,
- problemi bi trebali biti stvarni i motivirajući.

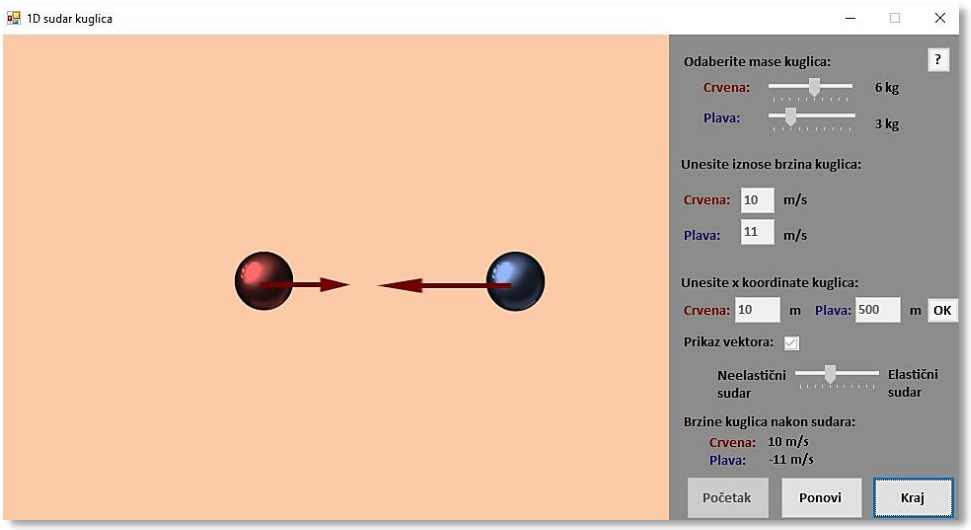
Vježbe iz predmeta OOP su oblikovane tako da se smisleno rješavaju mali problemi stvarajući potrebu za upotrebom odgovarajućih objektno orijentiranih koncepata. Nastavnici na predmetu OOP su detaljno upoznati s prethodnim znanjem studenata tako da vode računa o eventualnim konfliktima. Poseban primjer su studenti koji su upisali neki od smjerova s fizikom obzirom da oni nisu imali predmet Programiranje II te nisu imali prilike usvojiti osnove sintakse C# programskog jezika. Obzirom da su ti studenti učili isključivo proceduralnu paradigmu bili su više izraženi konflikti s objektno orijentiranom paradigmom i okruženjem vođenim događajima. Primjerice, kod proceduralne paradigme navikli su da se program izvršava od početka do kraja te da na tijek programa utječu tri osnovne algoritamske strukture slijeda, grananja i ponavljanja. Kod programiranja vođenog događajima tijek programa ovisi o interakciji s korisnikom i događajima koji se ne moraju izvršiti istim redoslijedom. Uzmimo primjer zadatka gdje je potrebno napraviti grafički program za unos brojeva klikom na „Unesi broj“ te nakon što se unese pet brojeva ispisati njihov zbroj (Slika 3.2).



Slika 3.2. Primjer zadatka

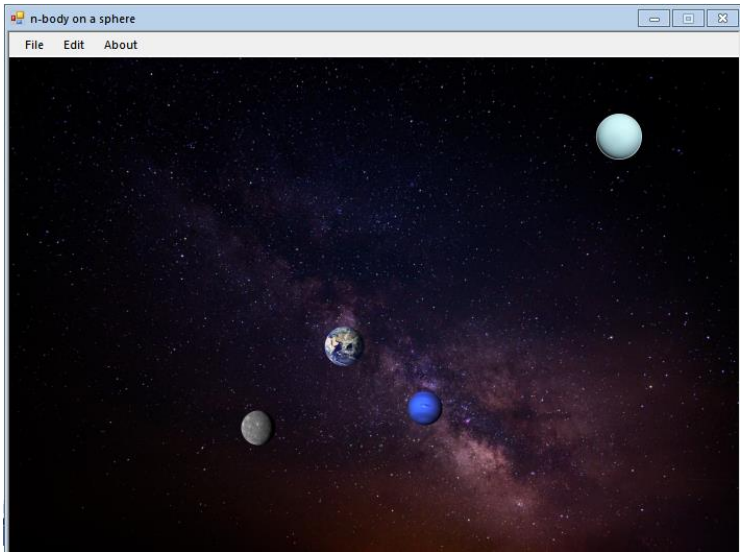
Studenti koji imaju konflikt s proceduralnom paradigmom su pokušavali riješiti problem pomoću petlje tako da se klikom na „Unesi broj“ petlja izvrši pet puta te u svakoj iteraciji učita broj iz polja za unos i nakon toga ispiše zbroj. Pri tome su se iznenadili što njihov program nakon klika pročita pet puta prvi uneseni broj te nemaju mogućost unosa ostalih. Međutim, ideja je da se klikom na „Unesi broj“ učita samo jedan broj, a korisnik je taj koji treba kliknuti pet puta, odnosno pet puta inicirati događaj klik. U proceduralnom pristupu je u svakoj iteraciji petlje program čekao na unos korisnika dok ovdje korisnik unese broj, aplikacija čeka na klik te nakon toga preuzme uneseni broj iz polja za unos.

Nastavnici su nastojali ukloniti takve konflikte organiziranjem dodatnih konzultacija za studente koji su nakon toga lakše svladali ostatak vježbi te su im kasnije koncepti koji se povezuju sa stvarnim svijetom bili jasniji. Posebno je značajno što su studenti različitih profila kasnije birali različite tipove projekata, što pokazuje i primjer simulacije 1D sudara kuglica na slici (Slika 3.3). Studenti su primijenili znanja iz svog područja i izradili simulaciju. Pri tome je OTTER odlično poslužio za vizualizaciju simuliranih kuglica.



Slika 3.3. Primjer simulacije izrađen u OTTER-u

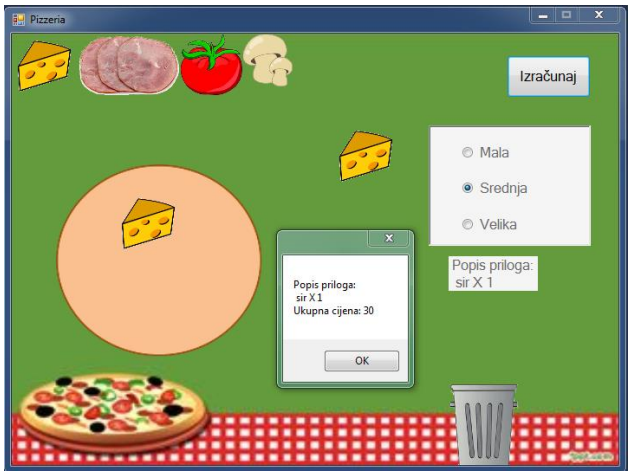
Neki studenti su nastojali povezati svoj projekt sa sadržajem drugih predmeta ili eventualno svojim područjem koje ih više zanima nego računalne igre pa tako imamo i primjer kretanja planeta gdje se simulira i treća dimenzija (Slika 3.4).



Slika 3.4. Simulacija kretanja planeta

Studenti su na vježbama dobivali jednostavne probleme koje trebaju riješiti uz pomoć OTTER okvira. Nakon opisa problema, nastavnici su vodili studente prema rješenju uz istovremenu primjenu gotovih elemenata iz OTTER-a. Studenti bez primjene okvira ne bi bili u mogućnosti riješiti zadatke s vježbi. Premda koncepti programskog jezika koji su u samom okviru nisu previše složeni, jer se koriste samo *Winows Forms* elementi, izrada grafičkog dijela prikaza likova na *formi* „od nule“ je vremenski zahtjevna. Vizualizacija izvršavanja programa je od velike pomoći studentima, no dijelovi programskog kôda koji su vezani za grafiku su dio

irelevantnog kognitivnog opterećenja te nije nužno da ih studenti svladaju kako bi uspješno riješili zadatke. Na slici (Slika 3.5) je primjer aplikacije izrađene u OTTER okruženju koja koristi kombinaciju OTTER elemenata s Windows Forms kontrolama. Radi se o aplikaciji s piktogramima koja omogućuje slaganje narudžbe i ispis cijene.



Slika 3.5. Primjer aplikacije za narudžbe

Učenici koji uče izradom projekata su aktivni, istražuju i konstruiraju znanje na temelju stečenog iskustva (Frank i ostali, 2003). Na primjer, kod učenja programiranja u Scratch-u, učenici uče kroz iskustvo metodom „pokušaja i pogreški“ gdje često sami istražuju upotrebljavajući gotove blokove naredbi koji su im jednostavno dostupni u okruženju. Takav konstruktivistički pristup olakšava upravo ta dostupnost blokova i ograničeni skup naredbi. Naravno, nastavnici moraju usmjeravati učenike u radu kako bi bili u skladu s ciljevima učenja na predmetu.

Pristup poučavanju programiranja pomoću ograničenog mikrosvijeta u kojem korisnik programskim naredbama kontrolira jednog ili više likova (kao Logo, Alice, Scratch i sl.) je popularan, posebno kod nižih uzrasta, ali mnogim učenicima je upravo problematičan prijelaz iz ovakvog okruženja u profesionalno okruženje (Michaelson, 2018). Ograničeni jezik u trenutku prijelaza predstavlja problem. U pristupima poučavanju gdje se uči najprije objekte nastoji se stvoriti okruženje analogno mikrosvjetovima.

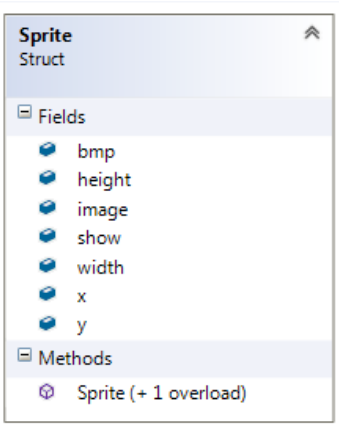
A. Kay (1993) je uočio kako je odraslim početnicima teško prijeći iz jednostavnih problema u složene te je smatrao kako bi trebalo eksplicitno poučavati i način oblikovanja programa. Ovdje treba napomenuti da se u to vrijeme bavio poučavanjem Smalltalk-a koji je objektno orijentiran programski jezik. Njegova suradnica A. Goldberg je došla na ideju *predložaka oblikovanja* (eng. design templates) (A. C. Kay, 1993). Ideja predložaka je bila da početnici imaju gotove

male dijelove kôda koje bi upotrijebili u svojim rješenjima. Početnici bi promatrali problem, rastavili ga na klase i poruke tako da ne moraju razmišljati kako svaka metoda radi. Goldberg i Kay su shvatili da programeri početnici ne mogu samostalno rješavati teške probleme bez obzira koliko je dobar programski jezik kojeg koriste, a čak i nakon što usvoje i shvate koncepte programskog jezika čini se da im ipak nedostaje izražajnosti (Liu, 2000), odnosno nisu u mogućnosti formulirati rješenje problema u programskom jeziku (Medeiros i ostali, 2019). Prema tome, njihovi *predlošci oblikovanja* predstavljaju složene elemente (male uzorke kôda, (eng. software patterns)) koji su konceptualno iznad izvornog kôda, a koje početnici ne mogu samostalno napraviti (Liu, 2000) ili detalji implementacije tih elemenata u tom trenutku nisu ključni za ono što početnik treba postići. Možemo ih smatrati oblikom potpore (Michaelson, 2018). Zbog ograničenog vremena u kojem se očekuje da studenti početnici usvoje neke koncepte, svakako im je potrebno pomoći, odnosno morali bi krenuti od nečeg započetog kako bi mogli napraviti bilo što složenije što uključuje objektno orijentirane koncepte (J. Kay i ostali, 2000). Powers & Powers (1999) tvrde kako je poučavanje najbolje kad se kombinira više različitih metoda poučavanja te da je važno prilagoditi metode kontekstu poučavanja, sadržaju, populaciji studenata i resursima. Također ističu kako je za učinkovitost poučavanja važno da je iskustveno (eng. experiential) što znači da bi studenti trebali iskusiti i doživjeti sadržaje. To se može primijeniti na vizualne elemente poučavanja.

Ideja razvoja posebnog okvira je krenula od ideja podržanih u Scratch-u: snižavanje početnog praga i omogućavanje vizualnog prikaza mikrosvijeta uz smanjivanje kognitivnog opterećenja vezanog za jezik, ali bez ograničavanja jezika ili upotrebe nekog drugog razvojnog okruženja. Pri tome je također bio cilj koristiti nešto posebno prilagođeno konkretnim studentima. OTTER ne spada u kategoriju predložaka oblikovanja kao što su to radili A. Kay i A. Goldberg niti je posebno okruženje koje kombinira profesionalni programski jezik s vizualnim elementima kao Greenfoot, a nije ni profesionalni okvir za razvoj igara kao XNA ili Otter2d. Jednostavno je upotrijebljeno sve što su studenti već koristili na predmetu Programiranje II (elementi *Windows Forms* aplikacija) uz dodatak crtanja po *formama* te paralelnog izvršavanja procesa (ideja iz Scratch-a).

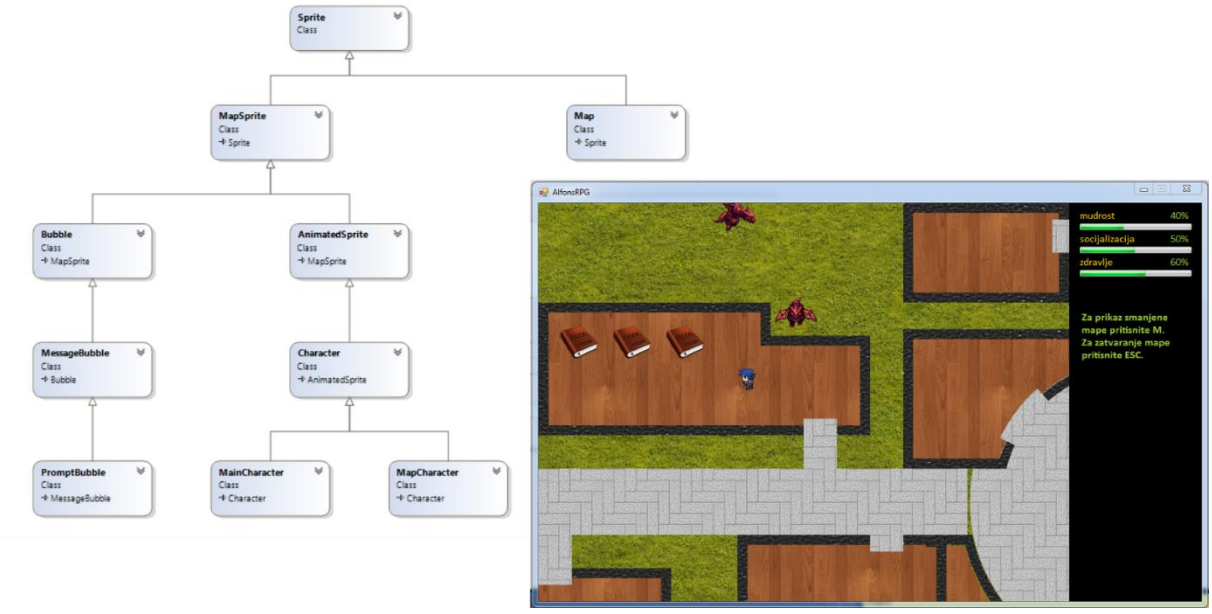
3.2.1 Razvoj okvira

Prva verzija okvira koja se počela koristiti u ak. godini 2013/14 te bila je najjednostavnija. Okruženje se sastojalo od glavne forme pod nazivom BGL čija kratica je predstavljala „background layer“ odnosno pozadinu. Glavni element cijelog okvira bio je *Sprite* koji je struct (Slika 3.6).



Slika 3.6. Sprite u prvoj verziji okvira

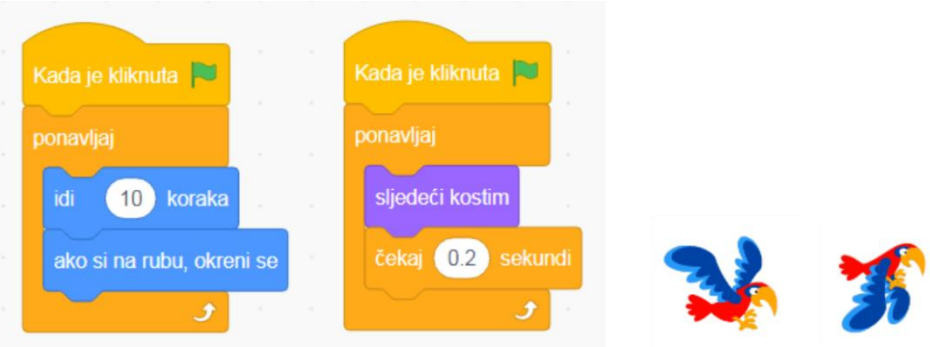
Cijeli programski kôd okvira sadržan je u glavnoj formi, a osnovna ideja izvršavanja temeljila se na osvježavanju zaslona, odnosno forme pomoću *programskog brojača* (eng. timer). Slično kao u crtanim filmovima, likovima se mijenjaju koordinate, a zaslon računala se osvježava dovoljno brzo kako bi se stekao dojam kretanja. Međutim, premda je okvir bio jednostavan i odabran kao motivacija za izradu projekata, ipak sam okvir nije sadržavao dovoljno ugrađenih objektno orijentiranih koncepata. Na primjer, prvo što se može uočiti je *struct* kao glavni lik, a ne klasa koja je osnovni koncept objektno orijentiranog programiranja. Na taj način studenti u samom okviru nisu imali dostupan primjer napisane klase. Jedan student je bio inspiriran sadržajima na nastavi te je napravio sve iz početka u obliku igre s igranjem uloga (eng. role playing game) (Slika 3.7). Projekt je primjer potpunog odbacivanja potpore (odnosno okvira). Student je napravio vlastiti okvir jer mu postojeći nije odgovarao za implementaciju ideje.



Slika 3.7. AlfonsRPG - studentski projekt

Na slici je vidljiva razrađena struktura elemenata projekta u dijagramu klasa gdje je jasna primjena nasljeđivanja, a lik odnosno *Sprite* je postao klasa. U navedenom primjeru su uočeni nedostaci postojećeg okvira kao i potencijali za buduće verzije. AlfonsRPG je studentski projekt izrađen iz početka, a to na neki način garantira da nije jednako složen kao profesionalni okviri za razvoj igara. Jedini problem s tim projektom je prevelika orijentiranost na koncepte računalnih igara tako da taj projekt nije mogao poslužiti kao osnova za neki novi okvir.

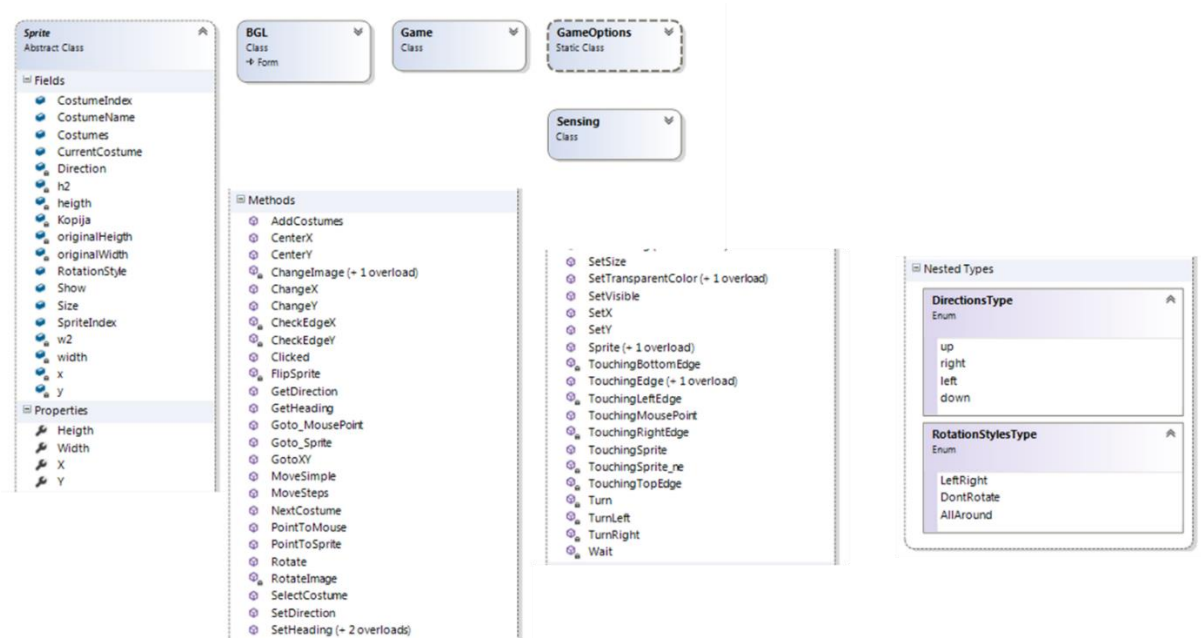
Postojeći OTTER je donekle podsjećao na *Scratch*, jer ima slične elemente: likove i pozornicu za izvršavanje. Problem je način izrade projekata u Scratch-u i u OTTER-u nisu bili ni malo povezani jer bi za prijenos najjednostavnijih projekata bilo potrebno mijenjati način razmišljanja. Na primjer, u jednostavnom programu gdje ptica maše krilima i kreće se brzinom 10 točkica (Slika 3.8) radi se o dvije paralelne *skripte* ili procedure u Scratch-u, a to nije bilo moguće na isti način realizirati u OTTER-u jer u trenutnoj verziji nije bio podržan paralelizam.



Slika 3.8. Scratch i paralelne skripte

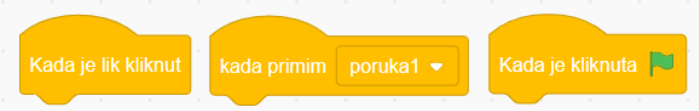
Zbog potencijalnog prijenosa znanja iz Scratch-a i sličnih vizualnih jezika cilj je bio približiti okvir Scratch-u i sličnim okruženjima kako bi taj prijenos bio što jednostavniji. Jedna od ideja je bila da studenti imaju mogućnost u Scratch-u napraviti osnovni prototip ili ideju projekta neopterećeni detaljima C# sintakse, te bi kasnije mogli implementirati prototip u OTTER-u.

Iz svih navedenih razloga, početni okvir je većim dijelom izmijenjen, ali je i dalje zadržana osnovna ideja rada s Windows Forms okruženjem bez dodatnih biblioteka i novih koncepata. Klasa *Sprite* je napisana iz početka na način da je imala više sličnosti s likovima u Scratch-u (slične akcije i svojstva likova koji prate i nazive blokova naredbi u Scratc-u). Na primjer: *NextCostume(), Rotate()* i sl.



Slika 3.9. Pregled klasa s metodama (ak. god. 2015/16)

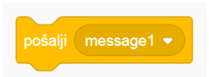
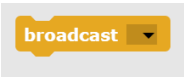
Problem paralelnog izvršavanja koji podsjeća na Scratch je zahtijevao uvođenje koncepata paralelnog programiranja i istovremenog izvršavanja, no to je bilo van opsega nastavnog sadržaja na predmetu. Bilo je lako shvatiti ideju da lik može istovremeno obavljati dvije aktivnosti kao na primjer mahanje krilima i istovremeno pomicanje lika definiranom brzinom, no dalje slijede problemi sinkronizacije ponašanja i interakcije likova. Na primjer, ako dodir dvaju likova donosi bodove, a svaki ima svoje ponašanje, odnosno dio kôda ili skriptu koja reagira na taj dodir, onda treba odrediti koji lik broji bodove ili prekida izvršavanje igre u slučaju da se ispuni neki uvjet. Logički je također razumljiva ideja, ali za implementaciju u C# treba koristiti nove koncepte programskog jezika kao što je klasa *Thread* te pokretanje procesa u posebnim programskim nitima (eng. threads). Obzirom da je takav način rada bio jedna od bitnih funkcionalnosti Scratch-a te bi drugačija implementacija udaljila model OTTER-a od Scratch-a, uvedene su metode koje **skrivaju** stvarnu implementaciju. Scratch ima koncept slanja poruka (eng. broadcast), što idejno odgovara pozivanju (eng. invoke) događaja u C#. U Scratch-u postoje blokovi koji „slušaju“ i očekuju takve poruke ili ako ćemo koristiti terminologiju C#-a: metode za upravljanje događajima (eng. event handlers). Postoji nekoliko posebno definiranih blokova kao što je „kad je lik kliknut“ ili „kad je kliknuta zastavica“, a za sve ostalo koristi se „kada primim [naziv poruke])“ (Slika 3.10).

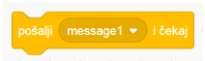


Slika 3.10. Blokovi za upravljanje događajima u Scratch-u

Blokovi s gornje slike su *početni blokovi* (eng. hat blocks), a na njih se spajaju sve naredbe/blokovi koji se trebaju izvršiti. Može biti više istih početnih blokova s pripadajućim naredbama koji će se pokrenuti istovremeno, npr. kad se primi *poruka1* s gornje slike. To na prvi pogled možemo promatrati kao pozivanje metoda koje reagiraju na događaje i stvaranje novih događaja, no s druge strane, također se događa i istovremeno izvršavanje. Kako bi se to realiziralo u C#-u, uvedeno je pokretanje metode kao posebnog procesa. Sam programski kôd kao što se može vidjeti na slici ispod (Slika 3.11) je prilično jednostavan i kratak no koncepti na kojima se temelji su složeni. Ako bi to ostavili studentima da sami pišu, time bi uveli irelevantno kognitivno opterećenje te je bilo nužno to sakriti u metodu.

OTTER	Scratch	Snap!

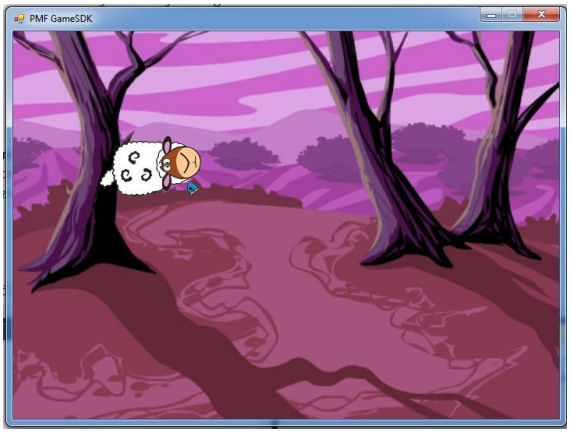
`public static void StartScript(Func<int> scriptName)`
`{`
`  Task t;`
`  t = Task.Factory.StartNew(scriptName);`
`}`
		

`public static void StartScriptAndWait(Func<int> scriptName)`
`{`
`  Task t = Task.Factory.StartNew(scriptName);`
`  t.Wait();`
`}`
		

Slika 3.11. Primjer didaktičkog skrivanja

Neki studenti će kasnije tijekom studija slušati predmet *Paralelno programiranje* te će tek tada koristiti i pisati metode sa slike, dok neki neće uopće, a neki će biti znatiželjni pa će pogledati implementaciju. Dodatno, studenti se odmah pri upoznavanju s OTTER-om susreću sa slanjem metoda kao parametara što im se objašnjava kao „naredba za pokretanje akcija“. Na primjer, ako napišemo: *Game.StartScript(OvcaSeta)* jasno je da će se pokrenuti metoda za kretanje lika. Slanje metoda kao parametara je priprema za upotrebu delegata.

Postavljanje pozornice i likova u Scratch-u odvija se kroz sučelje, a u OTTER okviru pomoću naredbi za instanciranje objekata i pozivanjem metoda.

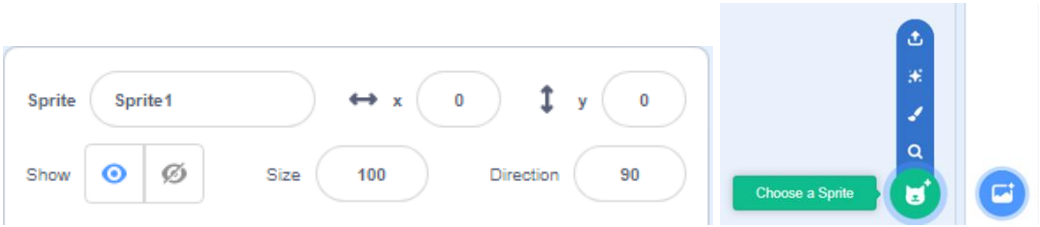


Slika 3.12. Izgled pozornice s pozadinom i jednim likom

Kako bi postavili sliku pozadine i jednog lika na pozornicu (Slika 3.12) potrebno je u C#-u napisati naredbe:

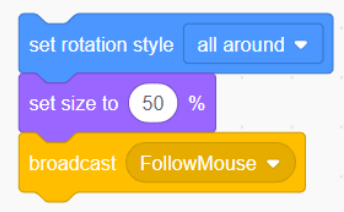
```
//Postavljanje pozornice
SetStageTitle("PMF GameSDK");
setBackgroundPicture("backgrounds\\woods.gif");
setPictureLayout("stretch");

//Dodavanje lika
Sprite sheep = new Sprite("sprites\\sheep01.png", 100, 100);
Game.AddSprite(sheep);
sheep.RotationStyle = "AllAround";

//Kad se pokrene igra postavlja se veličina i poziva odgovarajuća skripta
sheep.SetSize(50);
Game.StartScript(FollowMouse);
```

U Scratch-u se te stvari obavljaju pomoću grafičkog sučelja, što je bolje za mlađe učenike (Slika 3.13).



Slika 3.13. Postavke lika u Scratch-u i dodavanje novog lika

Likovi se dodaju također pomoću grafičkog sučelja, a ekvivalenti gornjih C# naredbi napisani u Scratch-u su prikazani na slici ispod (Slika 3.14).



Slika 3.14. Primjer u Scratch-u

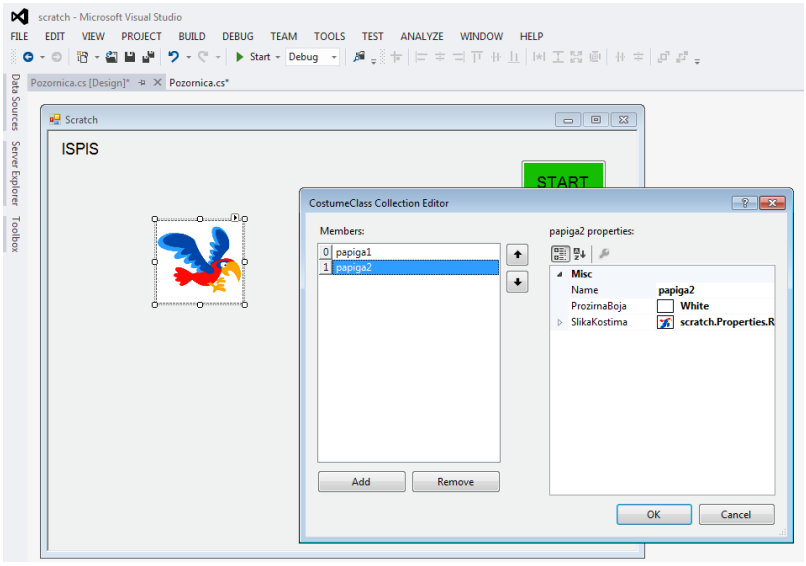
Drugi primjer didaktičkog skrivanja je jednostavna metoda *Cekaj(brojSekundi)* koja omogućuje pauzu ili čekanje programa prema upisanom broju sekundi. U primjeru je skriveno pretvaranje u milisekunde (jer *Thread.Sleep()* prima milisekunde) i pozivanje programske niti (eng. thread).

```
private void Cekaj(double brojSekundi)
{
    int ms = (int)(brojSekundi * 1000);
    Thread.Sleep(ms);
}
```

Slika 3.15. Primjer kôda

Ovdje je bila važna višenitna implementacija jer kad se metoda za čekanje pozove u jednonitnom programu onda jasno cijeli program čeka. Ovako samo jedan lik može čekati. Ovo se uklapa s idejom računarstva kao interakcije (poglavlje 2.3.3) (S. Fincher, 1999).

Na slici ispod (Slika 3.16) prikazan je primjer izrade grafičke aplikacije u prototipu okvira gdje je napravljena *kontrola* (eng. control) koja predstavlja *Sprite*. Za razliku od trenutnog OTTER okvira gdje se sve s likovima odvija u tekstualnom okruženju u ovom prototipu moguće je koristiti grafičko sučelje i miša za postavljanje likova i njihovih osnovnih svojstava.

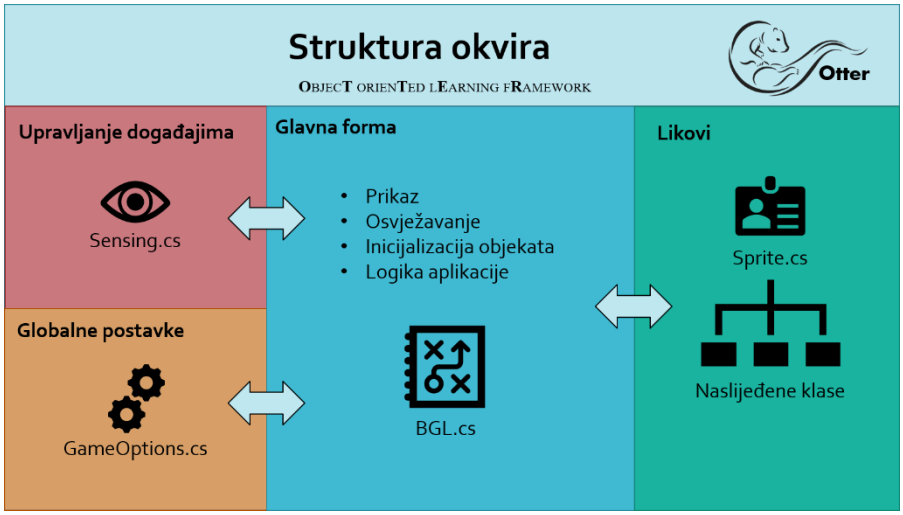


Slika 3.16. Prototip s likovima kao kontrolama

Student u grafičkom okruženju oblikuje likove jednostavnim povlačenjem mišem i odabirom slike, imena lika i sl. te koristi gotove metode za kretanje, animaciju, ... Ideja pri izradi prototipa je bila stvoriti okruženje još bliskije Scratch-u. Međutim, studenti općenito nemaju problema sa stvaranjem likova i postavljanjem svojstava pomoću tekstualnih naredbi tako da je za njih zapravo bilo korisnije ostaviti tekstualnu verziju u kojoj će vježbati pozivanje konstruktora umjesto dovlačenja lika mišem, pogotovo nakon što su to često činili na predmetima prije OOP. U prototipu je skrivena naredba za stvaranje novog objekta, odnosno MS Visual Studio je doda u pozadini. Spomenuti prototip bi se mogao pripremiti za neku raniju fazu poučavanja prije nego što su usvojeni koncepti stvaranja novih objekata pomoću tekstualnih naredbi.

3.2.2 Struktura okvira

Okvir OTTER sastoji se od četiri glavne komponente (klase) pri čemu svaka ima specifičnu namjenu. Cilj izrade okvira je bio pripremiti okruženje koje će studentima omogućiti jednostavniji uvod u objektno orijentirane koncepte, ali koji također omogućuje nadogradnju te može podržati složenije funkcionalnosti u slučaju potrebe. Okvir je izrađen pomoću Windows Forms .NET biblioteka što je studentima bilo blisko obzirom da je to glavni dio sadržaja kolegija P2. Na slici je prikazana struktura okvira.



Slika 3.17. Struktura okvira OTTER

- BGL.cs – Ova klasa je glavni dio okvira. Nasljeđuje klasu „Form“ koja predstavlja sučelje aplikacije. U ovom slučaju se koristi za prikaz sadržaja jednostavne 2D igre. Unutar klase nalazi se više različitih metoda koje se koriste za akcije koje se po potrebi pozivaju unutar same aplikacije. U početku je većina metoda skrivena. Najvažnija metoda za početak je “StartGame()” koja se koristi za inicijalizaciju svih objekata koji će se koristiti u igri te postavljanje njihovih početnih pozicija i karakteristika. Također,

unutar ove metode student može pozvati ostale metode vezane za ponašanje dinamičkih objekata koje sam izrađuje. Sve metode koje se odnose na ponašanje likova u igri ili izvršavanje same igre šalju se kao parametar u metodu „StartScript()“ koja ih izvršava paralelno. Likovi se prema tome mogu kretati istovremeno na zaslonu računala. Kod nekih okvira to radi drugačije – pronaći kako – glavna petlja Pygame. Implementacija pozivanja metoda kao paralelnih procesa je skrivena od studenata jer se radi o konceptu više dretvenog (eng. multi-threading) programiranja koji nije dio kolegija OOP, ali ostavlja otvorene mogućnosti proširenja. Koncept paralelnog izvršavanja je bliži stvarnom svijetu od nekih drugih pristupa (potraži ref kojih).

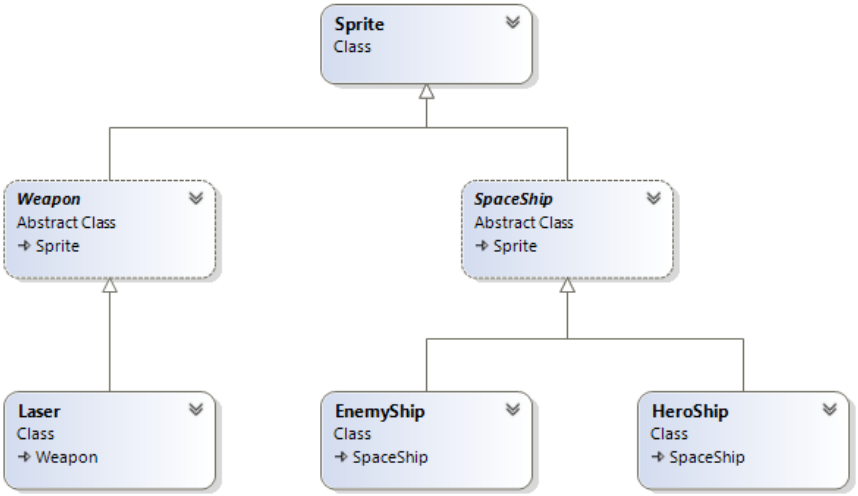
- Sprite – Ova klasa s prethodno opisanom BGL klasom čini glavni dio okvira. Klasa „Sprite“ čini temelj za sve buduće klase s vizualnom reprezentacijom koje će studenti definirati. To se najčešće odnosi na likove (npr. glavni lik, neprijatelj, interaktivni objekti, dekoracije, i sl.). Unutar klase definirana su različita korisna svojstva vezana za samog lika u igri (npr. x i y koordinate, širina, visina, smjer, slika i sl.) te metode koje predstavljaju osnovne akcije (npr. move, rotate i sl.). Klasa Sprite se definira kao apstraktna jer bi zapravo trebala samo služiti kao predložak za nasljeđivanje. U prvih nekoliko vježbi, dok studenti još nisu upoznati s konceptom nasljeđivanja i apstraktne klase, koriste dijelom modificirani okvir kako bi mogli koristiti instanciranje klase na uobičajen način.
- Sensing – Svrha ove klase je učahurivanje svojstva i metoda povezanih s obradom korisničkog unosa (odnosi se na miš i tipkovnicu). Jedna instanca ove klase se stvara u glavnoj BGL klasi, a njena svojstva se koriste kod upravljanja događajima vezanima za miš i tipkovnicu (npr. keyUp, keyDown, mouseClick i sl.). Cilj je čuvati osnovne informacije o događajima koji su važni za interakciju korisnika i likova u igri na jednom mjestu. Privremeno se skrivaju detalji implementacije te se studenti upoznaju s njom kroz koncept događaja. Kasnije studenti uče sami definirati svoje metode za upravljanje događajima kao i svoje događaje.
- GameOptions – Ova klase koristi se za demonstraciju upotrebe statičkih klasa. Sadrži nekoliko različitih (statičkih) polja i svojstava koji se koriste za definiranje uobičajenih i zajedničkih parametara igre (npr. dimenzije pozornice, zadana veličina svih likova i sl.). Metode koje pripadaju drugim klasama često koriste ove vrijednosti kako bi na primjer likovi mogli otkriti jesu li izašli van granica svijeta ili ne. Na taj se način standardne vrijednosti definiraju na jednom mjestu, a kako postoji samo jedan svijet

onda je logično da će klasa biti statička. Studenti mogu mijenjati sadržaj i vrijednosti ove klase isto kao što to mogu činiti za bilo koju drugu postojeću klasu.

Gore opisane klase, metode i sve što se nalazi pripremljeno u okviru usklađeno je s prethodnim predmetom P2, odnosno konceptima koje studenti uče na tom predmetu uzimajući u obzir problematični prijelaz iz proceduralne paradigme te poštujući donekle i strukturu Scratch okruženja (prostor za izvođenje, likovi, mjesto za pisanje kôda igre, metode za kretanje i sl.). Obzirom da je sama ideja Scratcha i sličnih jezika iznimno dobro prihvaćena, krenuli smo od te početne strukture, ali prilikom izrade samog okvira također smo se vodili pravilom o minimalnom uvođenju novih koncepata tako da su skoro svi elementi okvira izrađeni od struktura podataka i elemenata sučelja koje su studenti upoznali u prethodnim predmetima. Jedina novost je minimalni dio vezan za paralelno izvršavanje, ali za taj dio studenti ne moraju znati nikakve tehničke detalje niti ikad moraju otvoriti metodu koja se brine o pokretanju posebnih procesa.

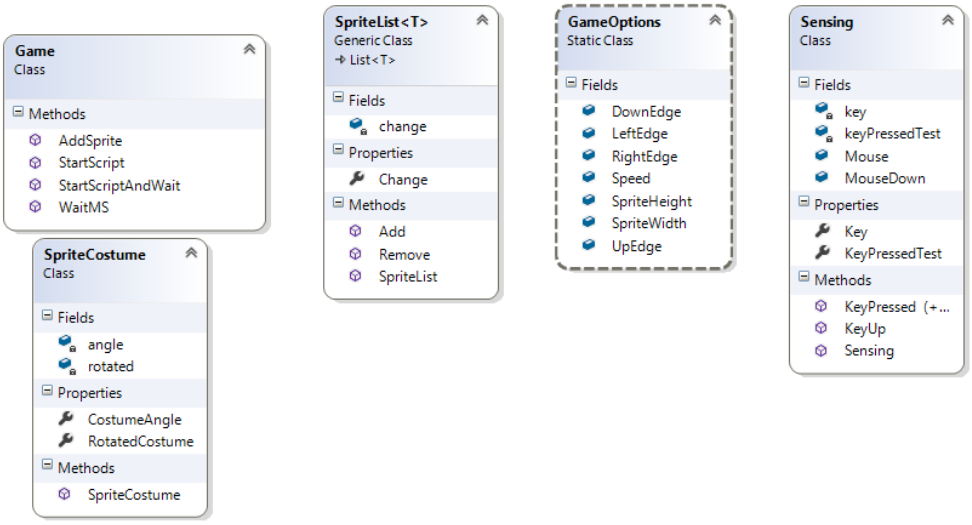
Sve klase koje su uključene u okvir su otvorene i dostupne studentima te ih studenti mogu mijenjati i prilagođavati svojim potrebama, obzirom na to da su tipovi projekata različiti. Na primjer, nije isto da li se igra odvija na platformama gdje lika gledamo sa strane i koji se može kretati samo u četiri osnova smjera (lijevo, desno, gore, dolje) ili je definiran pogled „od gore“ gdje se lik može okretati i bilo kojem smjeru (0-360 stupnjeva). Studenti će prilagođavati i mijenjati klase tek pred kraj semestra, tijekom razvoja svojih projekata, ali ovakvo okruženje im pruža dovoljno fleksibilnosti tako da su njihove ideje i kreativnost što manje ograničeni okvirom već samo njihovim znanjem i sposobnostima.

Uzmimo primjer igre koja je dio vježbe vezane za nasljeđivanje. Igra je ovdje zgodan primjer kako bi se pokazala hijerarhija klasa i nasljeđivanje koju također mogu vidjeti u samom programu kojeg izrađuju tijekom vježbi. Najprije je potrebno isplanirati koji su likovi u igri te po čemu se razlikuju. Glavna klasa *Sprite* je ostavljena kao predložak od koje će naslijediti svi ostali likovi. Prva podjela likova je na *Weapon* i *SpaceShip* koji predstavljaju oružje i svemirske brodove. Razlog korištenja engleskih naziva klasa je u izbjegavanju slova s dijakritičkim znakovima za koje nije uvijek sigurno da će biti podržani na različitim računalima. Bez obzira na upozorenja, neki studenti će ipak pisati *Oružje* umjesto *Oruzje*. Na ovom jednostavnom primjeru možemo pokazati koncepte apstraktnih klasa i nasljeđivanja, a studenti to realno mogu napraviti tijekom jednih vježbi. Primjer također služi kao osnova ili priprema za buduće projekte.

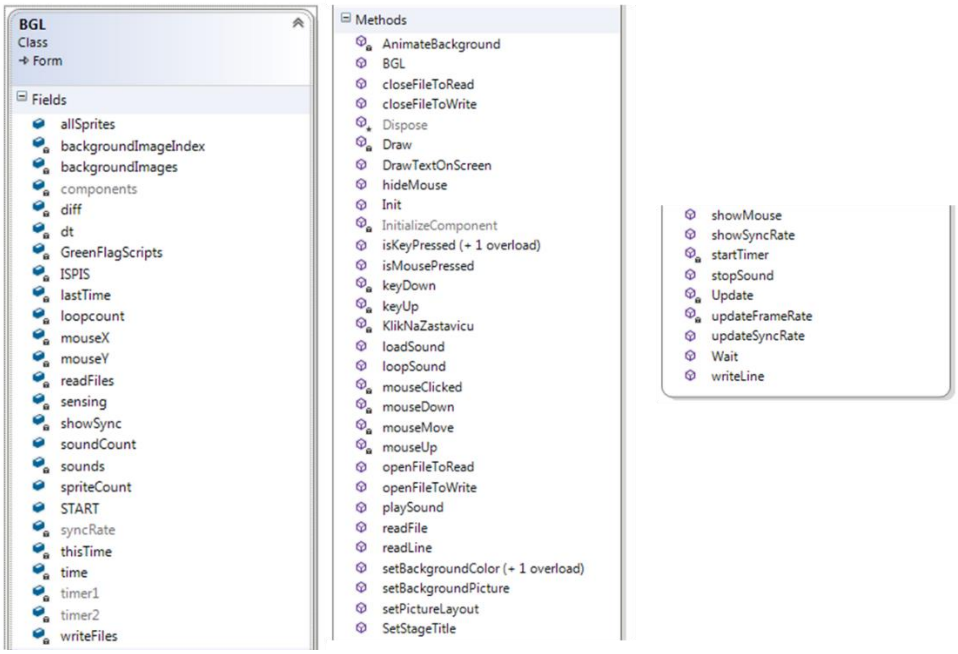


Slika 3.18. Primjer klasa u igri s vježbi

Na idućim slikama (Slika 3.19, Slika 3.20) prikazan je detaljniji pregled klasa i metoda koje su uključene u okvir.



Slika 3.19. Klase u OTTER 2



Slika 3.20. Glavna forma

U glavnoj BGL formi nalazi se niz metoda i svojstava koje omogućuju glavni „prostor“ za pisanje kôda. Radi jednostavnosti, studenti na jednom mjestu u formi (BGL klasa) pišu glavnu „logiku“ projekta. Ovdje je potrebno napomenuti da bi se neki dijelovi kôda prema principima objektno orijentiranog programiranja trebali prebaciti u posebne klase (npr. u Sprite), ali to bi od studenata zahtijevalo dodatnu razinu apstrakcije na koju jednostavno nisu spremni jer su početnici koji su tek nedavno počeli usvajati proceduralnu paradigmu. Ovakvo okruženje koje ne forsira apstrakciju za sve i pod bilo koju cijenu je bliskije profilu naših studenata. Dodatni razlog za pisanje glavnog dijela projekta ili interakcije likova je i dijeljenje resursa i lakša interakcija likova u istom „prostoru“ ili klasi. Ako dva lika spremljena u varijable *lik1* i *lik2* moraju razmijeniti podatke ili promijeniti jedan drugome trenutno stanje onda je studentima bilo jednostavnije to realizirati kad su *lik1* i *lik2* globalne varijable unutar klase te ih ne moraju slati kao parametre. Slanje varijabli kao parametara u metode je bio problematičan na prethodnom predmetu P2. Jasno je da će se povremeno naći i netko napredniji, ali okruženje ne postavlja nikakva ograničenja (npr. AlfonsRPG).

3.3 Alat za analizu projekata

Analiza izvornog kôda nije novost, postoje različiti pristupi i alati, a sve ovisi što se analizom želi postići. Možemo analizirati programe s ciljem procjene kvalitete, traženja sličnog kôda (klonova ili plagijata), traženja pogrešaka ili propusta u oblikovanju, optimizacije kôda i sl. Analizu može provoditi čovjek samostalnim ili tzv. „ručnim“ pregledavanjem izvornog kôda

ili se može provoditi pomoću računalnog alata. Ručna analiza koju bi obavljao isključivo čovjek je spora, naporna, dugotrajna i podložna pogreškama zbog ljudske nepažnje i zamora.

Za istraživanje je bilo potrebno analizirati veći broj studentskih projekata kako bi ispitali primjenu objektno orijentiranih koncepata. Obzirom na upotrebu početnog OTTER okvira koji se razlikuje po generacijama, bilo je potrebno izbaciti iz analize ono što su studenti već dobili kao gotovo ili nisu uopće primjenjivali u svojim projektima. Ne bi bilo u redu da se elementi OTTER okvira koje su pripremili nastavnici vrednuju i ocjenjuju u individualnim studentskim projektima jer neke od tih elemenata studenti neće nikad ni upotrijebiti. Prema tome, postavljeni su sljedeći zahtjevi:

- usporediti svaki projekt s početnim okvirom i utvrditi je li korišten okvir ili ne,
- ako je korišten okvir, izdvojiti dijelove kôda koje su studenti pisali sami – razlike,
- izdvojiti i analizirati elemente objektno orijentirane paradigme iz razlika.

Ako studenti nisu koristili okvir, onda će se cijeli projekt razlikovati te će se analizirati cijeli kôd. Nije bila dozvoljena nadogradnja nekih drugih postojećih okvira. Obzirom na zahtjeve, bilo je jasno da će se trebati ispitati na koji način se mogu analizirati koncepti objektno orijentirane paradigme te da će trebati analizirati i sličnost programskog kôda kako bi se utvrdilo što su studenti nadogradili u svoj projekt u odnosu na početni. Istražili smo što postoji u literaturi te da li se može primijeniti neki od postojećih programskih alata ili pristupa kako bi izbjegli ili ubrzali „ručnu“ analizu, ali su zahtjevi analize prilično specifični. Prvi korak je istražiti pristupe i metrike za provjeru kvalitete programske podrške.

3.3.1 Metrike kvalitete programske podrške

Kvaliteta programske podrške je važna za industriju. Metrika kvalitete programske podrške (eng. software quality metrics) je „alat“ koji omogućuje mjerenje i predviđanje potrebnih resursa koji će se koristiti u projektu (npr. računalnih resursa, vremena potrebnog za razvoj, cijene razvoja i sl.), a sastoji se od skupa različitih mjera za pojedine elemente ili procese (Punia, Kumar, & Gupta, 2016; Shaik, 2012). Objektno orijentirana programska podrška temelji se na konceptima kao što su klase, metode, enkapsulacija, nasljeđivanje i polimorfizam, a ti koncepti su razlog što je teže definirati metriku za objektno orijentirane projekte (Singh, 2013).

Abreu & Melo (1996) definirali su *Metrike za objektno orijentirano oblikovanje* (eng. Metrics for Object Oriented Design, MOOD) koja predstavlja skup metrika, a svaka od njih koristi se za neki od koncepata objektno orijentirane paradigme. Na primjer: *Faktor skrivanja metoda* (eng. Method Hiding Factor, MHF) i *Faktor skrivanja atributa* (eng. Attribute Hiding Factor,

AHF) koriste se za mjeru enkapsulacije. MHF se definira kao omjer broja skrivenih i neskrivenih metoda definiranih u klasama u odnosu na ukupan broj metoda definiranih u promatranom sustavu, a AHF se definira analogno za atribute (Abreu & Melo, 1996), no ako studenti često u projektima rade privatna polja i javna svojstva koja u vezana uz ta polja, onda se taj omjer neće razlikovati i bit će blizu jedan tako da ove dvije metrike nisu imale previše smisla za studentske projekte. *Faktor polimorfizma* (eng. Polymorphism Factor, POF) temelji se na broju premošćenih metoda (Abreu & Melo, 1996; Harrison, Counsell, & Nithi, 1997). Studenti su učili koncept premošćivanja (eng. override) i preopterećenja (eng. overload) na vježbama, te se očekivala primjena tih koncepata u projektima što se opet moglo i prebrojati.

Chidamber & Kemerer (1994) definiraju više metrika od kojih ćemo spomenuti samo nekoliko (primjenjivih na studentske projekte):

- *Broj težinskih metoda za klasu* (eng. Weighted Methods per Class, WMC). Metrika WMC se odnosi na brojanje metoda u klasi uzimajući u obzir i složenost tih metoda, odnosno *težinu* (eng. weight), a može se koristiti za predviđanje vremena i rada potrebnog za razvoj i održavanje klase. Veći broj metoda u klasi ima veći utjecaj na sve klase koje od nje nasljeđuju, a također može značiti da je klasa previše specifična za aplikaciju čime joj se ograničava ponovna upotrebljivost. Kod analize studentskih projekata koristi se ideja ove metrike s brojanjem metoda, ali se ne računa složenost tih metoda obzirom da su većinom metode koje su studenti sami radili u vlastitim klasama prilično kratke, no promjenom opsega projekata ili OTTER okvira moguće je uključiti i težine.
- *Dubina stabla nasljeđivanja* (eng. Depth of Inheritance Tree, DIT). Metrika DIT predstavlja dubinu stabla nasljeđivanja, odnosno u slučaju nasljeđivanja gdje imamo više razina (npr. roditelj – dijete – unuk, …) DIT predstavlja duljinu od korijena do lista. Klase koje su najudaljenije od korijena nasljeđuju najviše metoda i svojstava što potencijalno može značiti njihovu bolju ponovnu upotrebljivost, ali je s druge strane teže predvidjeti njihovo ponašanje. Također, veći DIT broj označava složeniji model odnosno oblikovanje projekta, ali DIT razine 6 već može značiti preveliku razinu nasljeđivanja što na kraju znači teže održavanje. U analizi projekata koristili smo DIT kako bi ispitali koncept nasljeđivanja. Ako je DIT jednak nuli, to znači da se nije koristio koncept nasljeđivanja, a za DIT veći od jedan možemo zaključiti da je objektni model projekta složeniji.

- *Broj djece* (eng. Number of Children, NOC). Metrika NOC predstavlja broj neposrednih podklasa u hijerarhiji. Predstavlja mjeru koliko će podklasa naslijediti metode klase-*roditelja*. Što je veći broj djece, veća je ponovna upotrebljivost, ali isto tako preveliki broj može biti značiti pogrešnu apstrakciju glavne klase, odnosno loše definirane klase. U analizi projekata nismo koristili NOC jer je vezana za koncept nasljeđivanja pa bi time evidentirali isti koncept kao i DIT, ali bi se metrika NOC mogla koristiti za detaljniju analizu koncepta nasljeđivanja u projektima.
- *Povezanost objekata klasa* (eng. Coupling Between Object classes, CBO). Metrika CBO za neku klasu predstavlja broj drugih klasa s kojima je povezana putem objekata, odnosno polazi od pretpostavke da su dva objekta povezana ako jedan djeluje na drugi (npr. koristi metode drugog objekta). Preveliki CBO znači da će biti teže izdvojiti neke klase u druge projekte, a također ih je teže održavati. Metriku CBO nismo koristili pri analizi jer je sama struktura okvira nametnula početnu razinu CBO. Na primjer, glavne aktivnosti projekta odvijaju se slično kao u Scratch-u, na glavnoj pozornici koja je glavna *forma* odnosno klasa igre, pa je prema tome većina objekata uvijek u interakciji s pozornicom, ali isto tako i s drugim elementima okvira. Međutim, metrika CBO se može koristiti za složenije projekte s više klasa, a pogotovo za projekte koji nisu napravljeni u zadanom okviru.

Neke od jednostavnijih metrika odnose se na brojanje pojedinih elemenata (Harrison i ostali, 1997), na primjer: *broj metoda* (eng. Number of Methods, NM) može se koristiti za provjeru broja funkcionalnosti koje obavlja neka klasa (sadrži li klasa previše ili premalo funkcionalnosti), *broj dodanih metoda* (eng. Number of Methods Added, NMA) odnosi se na broj novih metoda podklase koje ne postoje u nadređenoj klasi.

Studentski projekti koji se analiziraju u ovom radu ne mogu se mjeriti na isti način kao projekti u industriji, obzirom da se radi o studentima početnicima koji nisu upoznati s programskim inženjerstvom niti su projekti istog opsega kao profesionalni, a nedostaje im iskustva, znanja i vještina. Uzmimo za primjer jednostavnu metriku LOC. Na prvi pogled, više linija kôda može značiti složeniji projekt, no isto tako programer stručnjak može zbog svog iskustva i poznavanja jezika riješiti isti problem u puno manje linija kôda nego student početnik. Studenti koriste pripremljeni okvir OTTER koji nije namijenjen za profesionalni razvoj već isključivo za učenje. Okvir je prilagođen predznanju studenata našeg fakulteta koji su prije predmeta OOP učili proceduralnu paradigmu u Pythonu i osnove sintakse u C#. Obzirom da su svi projekti koje su

studenti predali bili ispravni (jer su temelj za usmeni dio ispita), nije bilo potrebno tražiti pogreške u predanim projektima već analizirati programski kôd.

3.3.2 Statička analiza kôda

Statička analiza kôda (eng. static code analysis) je analiza programske podrške koja se provodi bez izvršavanja programa (Gomes, Morgado, Gomes, & Moreira, 2009). U većini slučajeva provodi se na izvornom kôdu pomoću nekog programskog alata. Statička analiza koja se provodi pomoću računala je brža od analize koju obavlja čovjek tako da se može provoditi češće.

Alati za statičku analizu kôda obično pronalaze nedostatke u programima koje prevoditelj ne može uočiti. Na primjer može se bez pokretanja programa uočiti greške koje će se pojaviti tek za vrijeme izvršavanja, nepotrebna potrošnja resursa, sigurnosni problemi i sl. (Emanuelsson & Nilsson, 2008). Osim toga, alati za analizu kôda mogu provjeravati stil pisanja kao što je npr. pisanje razmaka prije i nakon operatora, upotreba početnog velikog slova za naziv metoda i sl. (Dankovčíková, 2017).

MS Visual Studio 2017 sadrži ugrađene alate kojima se na primjer može dobiti dubina stabla nasljeđivanja (DIT), broj linija kôda, povezanost klasa (eng. class coupling) i sl. Međutim, ako promatramo studentske projekte, za primjenu alata svaki od njih se mora otvoriti, pokrenuti analiza, te „ručno“ izdvojiti ono što je dio okvira, odnosno što nisu pisali studenti. Dodatan problem pri provjeravanju razlika je činjenica da su studenti mogli koristiti i mijenjati početni okvir dodavanjem različitih elemenata (metoda, klasa, blokova kôda, polja, …). Postojeći alati namijenjeni isključivo za provjeru kvalitete programa ne odgovaraju postavljenim zahtjevima jer je potrebno provjeriti i eliminirati sličnosti.

Procjena sličnosti izvornog kôda općenito ima različite primjene kao što su otkrivanje plagijata, traženje sličnih implementacija, razumijevanje programa, uklanjanje pogrešaka i sl. (Ragkhitwetsagul, Krinke, & Clark, 2018). Izmjene kôda u odnosu na početni možemo podijeliti na *temeljite izmjene* (eng. pervasive modifications) i *primjenu predloška kôda* (eng. boiler-plate code). Primjena predloška kôda događa se kad programer koristi dio postojećeg kôda, obično neku metodu ili blok kôda za obavljanje konkretne zadaće, a tipično se pojavljuje kad studenti rješavaju zadatke iz programiranja uz unaprijed zadane dijelove kôda ili predloške (Burrows, Tahaghoghi, & Zobel, 2007). Takva upotreba postojećeg kôda ne smatra se plagijatom. Studenti su na primjer tijekom izrade projekata na OOP morali koristiti predložak metode u glavnom dijelu programa, odnosno formi koja predstavlja pozornicu (Slika 3.21).

```
private int Metoda()
{
    while (START)
    {

        Wait(0.1);
    }
    return 0;
}
```

Slika 3.21. Predložak metode u glavnoj formi

Primjena ovog predloška nije posebno detektirana u projektima obzirom da gornja metoda bez ikakve izmjene zapravo ne radi ništa osim čekanja tako da je bilo očekivano da će je studenti prilično izmijeniti. Jedino je bilo važno da zadrži oblik potpisa.

Druga vrsta izmjena, odnosno *temeljite izmjene* (Ragkhitwetsagul i ostali, 2018), odnose se na izmjene kôda koje zahvaćaju cijeli projekt (globalne). To se često događa kad se želi skriti kopirani kôd ili plagijat pa se mijenja izgled na globalnoj razini kako nastavnik ne bi primijetio ili kad programer namjerno želi spriječiti razumijevanje postojećeg kôda radi zaštite svog projekta. No, takve izmjene se događaju i tijekom razvoja ili nadogradnje postojećeg programa što je slučaj kojeg ovdje želimo promatrati pri analizi projekata jer je to ciljano ponašanje.

Alati za otkrivanje plagijata ili sličnosti često koriste reprezentaciju programa u obliku tokena (eng. token) ili apstraktnih sintaksnih stabala (eng. abstract syntax tree, AST) (Roy, Cordy, & Koschke, 2009). Svrha takvih alata je otkrivanje promjena u kôdu koje mogu biti *leksičke* (eng. lexical): odnose se na izgled i raspored (eng. layout) i promjenu naziva identifikatora, *strukturalne*: odnose se na umetanje i brisanje naredbi i *ekstremne*: promjena sintakse uz zadržavanje semantike (Ragkhitwetsagul i ostali, 2018).

Obzirom da je traženje sličnosti kôda koje se temelji na metrici manje učinkovito (Kapser & Godfrey, 2003), Ragkhitwetsagul i ostali (2018) predlažu sljedeće pristupe koji se temelje na: *tekstu* (eng. textual), *tokenima* (eng. token based) i *stablima* (eng. tree based).

Tekstualni pristup funkcionira uspoređivanjem teksta ili niza znakova te traženjem sličnosti uzoraka teksta, na primjer traženje najduljeg zajedničkog niza znakova i sl. Pomak od čistog teksta ili niza znakova dobije se pretvaranjem kôda u *tokene* (slično riječima u prirodnom jeziku). Program se prikazuje nizom tokena, a razina apstrakcije može se definirati vrstom tokena. Na primjer, naredba: *int x = 5;* je slična naredbi *int broj = 10;* jer sadrži iste vrste tokena. Ako zadržimo razinu apstrakcije koja promatra samo vrste tokena: tip, naziv varijable,

operator pridruživanja i konstanta bez daljnjeg analiziranja koji je tip, koji je naziv varijable te koja je vrijednost konstante, onda možemo smatrati da su te dvije naredbe iste. Također, naredba *string s = "test";* također odgovara postavljenim kriterijima. Ako ćemo razlikovati brojčane tipove od tekstualnih, onda su prve dvije naredbe iste, a treća nije.

Pristup temeljen na stablima koristi se za izbjegavanje problema oko formatiranja ili leksičkih razlika, obzirom da se programi mogu uspoređivati po strukturi traženjem sličnih podstabala. To je donekle slično apstraktnim sintaksnim stablima (Dankovčíková, 2017; Ragkhitwetsagul i ostali, 2018), ali se ne mora preklapati s AST jer ovisi o samoj izgradnji sintaksnog modela koji opet ovisi o konkretnoj primjeni ili alatu. Računanje sličnosti temeljenih na stablima može biti složeno i sporo tako da se ponekad koriste optimizacije ili aproksimacije (Jiang, Su, & Chiu, 2007). Analiza kôda pomoću računala na gore navedene načine očito implicira rastavljanje studentskih C# projekata na dijelove ili strukture kao što su apstraktno sintaksno stablo, tokeni i sl. Izrada alata koji bi to obavio iz običnog teksta je prilično složen posao jer treba obaviti parsiranje teksta te prevođenje tog teksta u apstraktno sintaksno stablo. Ako se radi o C#-u onda je najbolje odabrati alat koji je namijenjen za taj jezik ili bolje rečeno integriran u samo okruženje, a to je Roslyn.

3.3.3 Roslyn

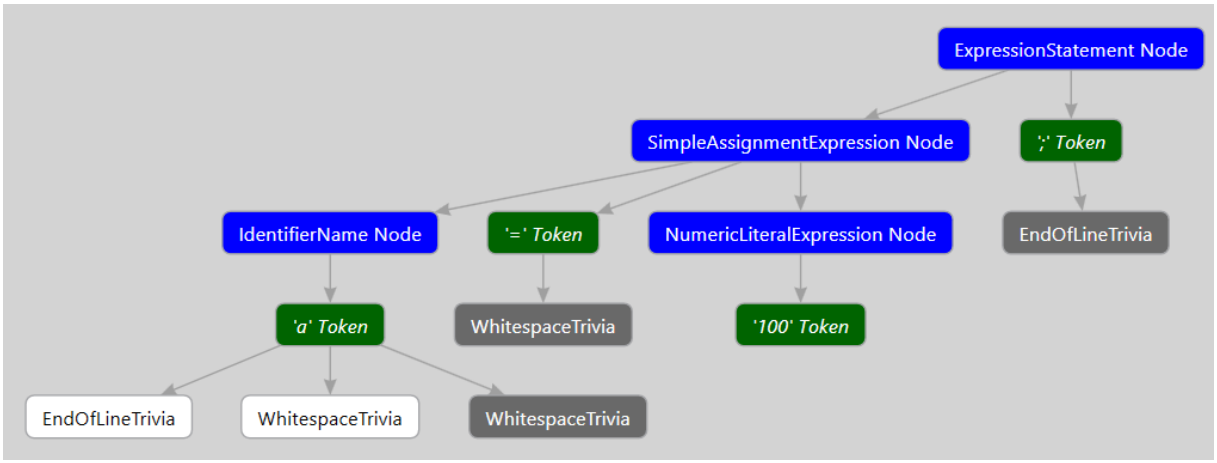
U početku je prevoditelj C#-a bio kao crna kutija koji bi na temelju datoteke s programskim kôdom vratio natrag izvršnu datoteku (Dankovčíková, 2017). Međutim, to se promijenilo 2015 godine uvođenjem .NET prevoditeljske platforme (eng. .NET Compiler Platform) pod popularnim nazivom *Roslyn* (Sole, 2016). Prevoditelji za Visual Basic i C# su ponovno napisani tako da omogućuju pristup *aplikacijskim sučeljima* (eng. Application Programming Interface, API) što je programerima omogućilo analizu i automatsko generiranje kôda te dinamičko prevođenje. Prevoditelj C# programa radi u nekoliko faza: parsiranje (eng. parse) niza znakova u sintaksno stablo, stvaranje simbola i metapodataka prema deklaracijama u izvornom kôdu, povezivanje identifikatora iz izvornog kôda sa simbolima i sl. (Dankovčíková, 2017). U svakoj fazi Roslyn stvara objektni model kojem se može pristupiti, a i samo okruženje MS Visual studio koristi te elemente (na primjer, sintaksno stablo se koristi za bojanje kôda). Za Roslyn je važno napomenuti da je sintaksno stablo koje se stvara *nepromjenjivo* (eng. immutable), što znači da se ne može mijenjati tijekom rada već u slučaju potrebe treba stvoriti novo s novim elementima. Na taj se način omogućuje da u slučaju potrebe više analizatora kôda radi istovremeno na istom programu.

Sintaksno stablo sastoji se od:

- sintaksnih čvorova (eng. syntax nodes),
- sintaksnih tokena (eng. syntax tokens),
- trivijalnih sintaksnih elemenata (eng. syntax trivia).

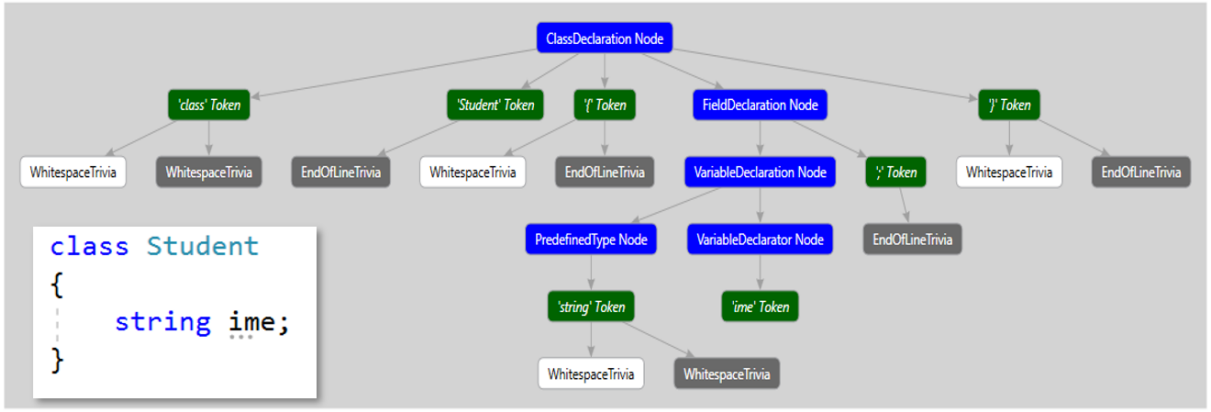
Sintaksni čvorovi su označeni plavom bojom, predstavljaju konstrukte kao što su naredbe i deklaracije. To su čvorovi koji sigurno imaju bar jedno *dijete* (ili podčvor) koji je ili sintaksni čvor ili token. Svaka vrsta sintaksnog čvora predstavljena je klasom izvedenom iz osnovne klase *SyntaxNode*. Osnovna klasa *SyntaxNode* omogućuje dohvaćanje djece (*ChildNodes*) te svih elemenata podstabla: *DescendantNodes, DescendantTokens, DescendantTrivia*.

Na primjer, za izraz: ***a = 100;*** sintaksno stablo izgleda kao na slici ispod (Slika 3.22).



Slika 3.22. Sintaksno stablo pridruživanja

Možemo vidjeti kako je apstraktno sintaksno stablo za kratku definiciju klase već prilično složeno ili bogato detaljima (Slika 3.23).



Slika 3.23. Definicija klase prikazana u DGML pregledniku

Sintaksni tokeni (zelena boja) predstavljaju terminalne (eng. terminal) konstrukte gramatike programskog jezika kao što su ključne riječi, identifikatori (u gornjem primjeru: *Student*, *ime*, *string*, *class*), interpunkcijski znakovi i literali. Svaki od njih ima svojstvo *Kind* kako bi tijekom analize imali povratnu informaciju kojoj vrsti token pripada.

Trivijalni elementi (siva i bijela boja) predstavljaju ostale konstrukte kao što su na primjer prazni znakovi (eng. whitespace), kraj naredbene linije i komentari, a pripadaju tokenima.

Međutim, postoje određene semantičke informacije koje ne možemo dobiti iz sintaksnog stabla pa je potrebno koristiti i Roslyn semantički model koji omogućuje informacije koje nisu dostupne izravno iz sintakse (npr. tip rezultata za neki izraz, vidljivost simbola na nekoj lokaciji, informacija da li klasa nasljeđuje i sl). Svako sintaksno stablo sadrži sintaksni model koji se može dohvatiti. Obzirom da smo već primijetili kako je Roslyn sintaksno stablo već za najjednostavnije dijelove kôda prilično složeno, onda je bilo potrebno reducirati broj detalja koji nam nisu bitni. Na primjer, možemo u potpunosti zanemariti trivijalne elemente jer ne sadrže nikakve bitne informacije o programu, a pogotovo ne o objektno orijentiranim konceptima. Ako bi radili refaktoriranje kôda onda bi nam te informacije bile potrebne.

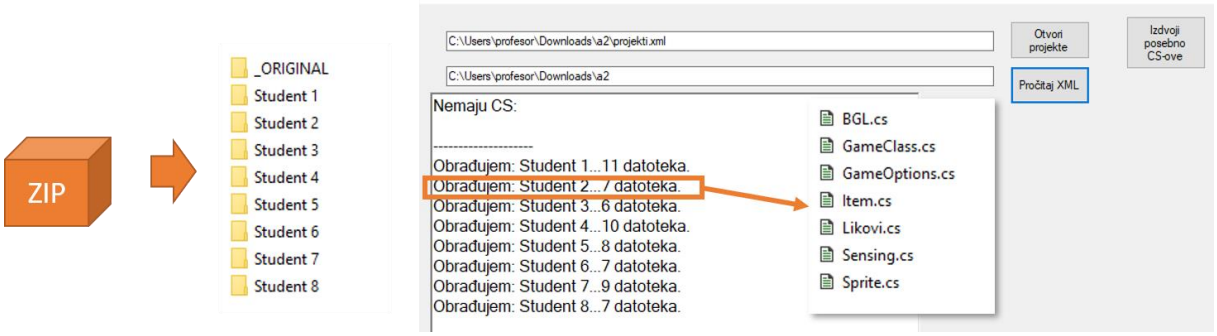
Projekti pisani u C# programskom jeziku u MS Visual Studio okruženju nazivaju se *rješenja* (eng. solutions), mogu se sastojati od jednog ili više projekata (eng. projects), a svaki projekt sastoji se od skupa datoteka. Programski kôd nalazi se u *cs* datotekama tako da je potrebno izdvojiti samo te datoteke za analizu, a ignorirati *Designer.cs* datoteke jer one nastaju automatski dovlačenjem kontrola na forme tako da studenti ne pišu kôd unutar tih datoteka.

Funkcionalnosti analizatora studentskih projekata:

- dekomprimiranje *zip, rar i 7z* formata u zasebne mape s imenima studenata,
- rekurzivno pretraživanje mapa i kopiranje *cs* datoteka koje sadrže programski kôd,
- stvaranje skraćenog sintaksnog stabla koje se zapisuje u xml datoteku za daljnju obradu,
- uspoređivanje početnog OTTER okvira sa svakim studentskim projektom i izdvajanje razlike,
- stvaranje stabla nasljeđivanja za sve klase radi računanja maksimalne dubine nasljeđivanja,
- zapisivanje podataka u *csv* datoteku.

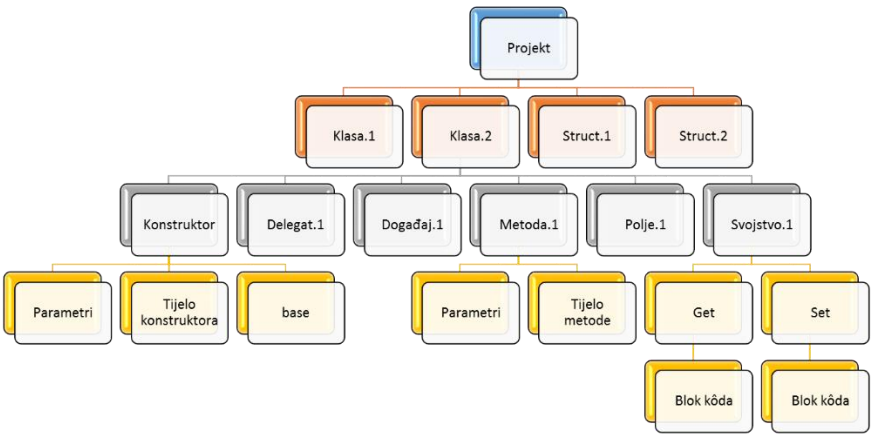
Premda Roslyn ima odgovarajuće klase koje predstavljaju svaki element AST, definiramo posebnu klasu *Cvor* u kojoj su izdvojene informacije potrebne za analizu projekata jer

originalno AST sadrži previše informacija. Svaki čvor reduciranog sintaksnog stabla sadrži sljedeća polja: ime (npr. ako se radi o metodi onda je to naziv metode), vrsta (Roslyn Kind polje), ime roditelja, informacije o tome je li čvor apstraktan, statički, *protected*, virtualan, premošćen, preopterećen, ako je metoda ili konstruktor onda sadrži potpis, ako je klasa onda sadrži informaciju da li nasljeđuje, dubina nasljeđivanja u tom čvoru (računa se samo za klase). Dodatno, svaki čvor ima informaciju o tome je li novi ili promijenjen, a to se koristi kod uspoređivanja početnog OTTER okvira sa studentskim projektima. Čvorovi imaju i kategoriju koja je označena s *GUI* ako promatrana klasa nasljeđuje od klase *Form*, odnosno ako je dio grafičkog korisničkog sučelja (eng. graphical user interface, GUI).



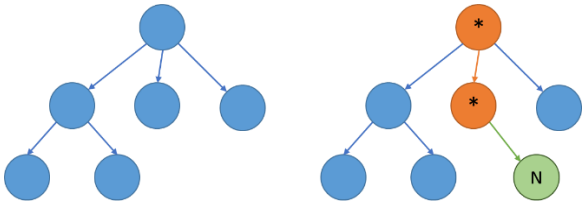
Slika 3.24. Faza izdvajanja *cs* datoteka

Korijen stabla koje se koristi u analizi je uvijek projekt, a njegova djeca su klase i struct-ovi. Korijen bi mogao biti i rješenje (eng. solution), ali studenti nisu nikad radili više projekata u istom rješenju. Svaka klasa ili struct može sadržavati: polja, svojstva, konstruktore, metode, delegate i događaje. Svaka metoda ima parametre i tijelo, konstruktor (koji je isto metoda) osim toga ima i informaciju da li poziva osnovni (eng. base) konstruktor klase od koje njegova klasa nasljeđuje.



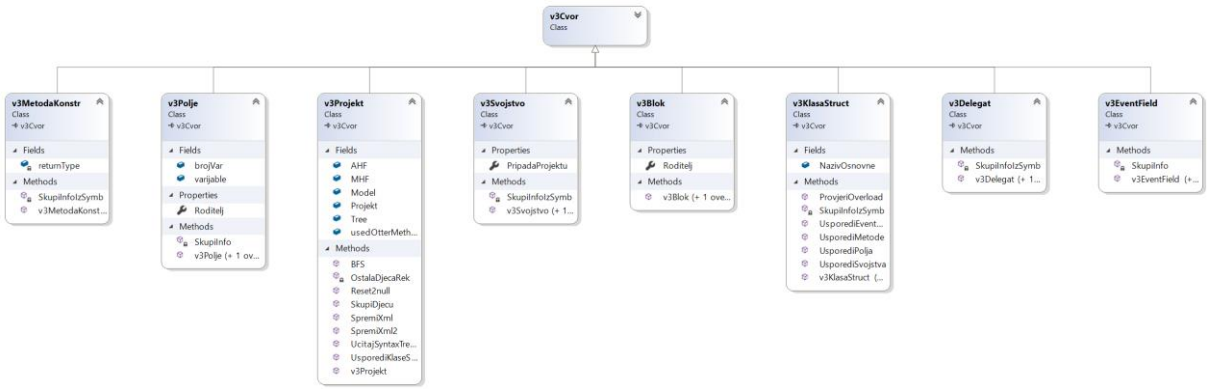
Slika 3.25. Skraćeno stablo za analizu

Svojstva sadrže *get* i *set*, koji dalje mogu sadržavati blok kôda ili su standardni „prazni“ koji ne obavljaju nikakve posebne funkcionalnosti osim *return* naredbe za *get* odnosno postavljanja vrijednosti za *set*. Algoritam uspoređivanja početnog OTTER okvira sa studentskim projektima uspoređuje razinu po razinu stabala pretragom po širini te analizira koji čvor u studentskom projektu je novi. Svaki novi čvor se posebno označava, a cijela grana do korijena dobiva posebnu oznaku da je došlo do izmjene. Na slici su izmijenjeni čvorovi označeni zvjezdicom radi ilustracije, a novi čvor ima oznaku N (Slika 3.26).



Slika 3.26. Razlika stabala

Obzirom da čvorovi stabla za analizu nisu „usitnjeni“ do razine tokena i trivijalnih elemenata kao u Roslyn-u, listovi se uspoređuju kao tekst ignorirajući razmake. U slučaju bilo kakve izmjene, algoritam će cijelu granu do korijena opet označiti kao izmijenjenu u odnosu na originalni okvir. Nakon što se usporede svi čvorovi iz studentskog projekta brišu se oni koji nisu izmijenjeni niti novi. Treba napomenuti da se ne brišu čvorovi koje su studenti koristili odnosno pozivali u svojem projektu već dobivaju oznaku „korišten“.



Slika 3.27. Vrste čvorova uključenih u analizu

Studenti su svake akademske godine dobivali različite početne okvire te je bilo neophodno usporediti studentske projekte s konkretnim okvirom u toj akademskoj godini. Odabran je pristup kombinacije uobičajenih pristupa traženja sličnosti: tekstualni (eng. textual), zasnovan na tokenima (eng. token-based) i zasnovan na stablu (eng. tree-based) (Ragkhitwetsagul i ostali, 2018).

Na kraju provedenog algoritma za svaki studentski projekt ostaje stablo za analizu u kojem se nalaze samo elementi koje je student nadograđivao, mijenjao ili primjenjivao. Daljnji koraci analize ne ovise o Roslyn-u te će biti opisani kod prikupljanja podataka.

4 METODOLOGIJA ISTRAŽIVANJA

Istraživanje ima dva glavna elementa: populaciju i područje istraživanja (Kumar, 2012). Populacija se odnosi na ljude, skupine, organizacije, zajednice i sl., a područje istraživanja na situacije, sadržaje, strukture, uzroke i posljedice, veze među pojavama i sl.

Kod eksperimentalnog istraživanja istraživači namjerno manipuliraju uvjetima u kojima se odvija neki događaj, uvode intervenciju te mjere učinak te intervencije (Cohen, Manion, & Morrison, 2013). Odabir slučajnog uzorka, način rasporeda ispitanika u kontrolne i eksperimentalne skupine te kontroliranje zavisnih i nezavisnih varijabli su dio ključnih pretpostavki eksperimentalnog istraživanja. Stvarni eksperiment (eng. true experiment) provodi se u laboratorijskim uvjetima s dvije ili više skupina, a kvazi eksperiment (eng. quasi experiment) provodi se u prirodnom okruženju, no varijable su ipak izolirane i kontrolirane.

Testiranje hipoteze je jedan od tradicionalnih načina provedbe istraživanja u području kognitivne psihologije (Gilmore, 1990). Međutim, u stvarnom životu nije sve onako kako zamislimo jer je teško izbjeći neočekivane posljedice koje utječu na eksperiment više od onog što očekujemo.

Ex post facto istraživanje se koristi kad su se pojave već dogodile, kad nije moguće ili nije prihvatljivo, primjerice zbog etičkih razloga, manipulirati karakteristikama ljudskih ispitanika te nije moguće provesti eksperimentalno ili kvazi eksperimentalno istraživanje (Simon & Goes, 2013). Iako su se kod *ex post facto* istraživanja pojave koje istražujemo već dogodile, oblik istraživanja je sličan dizajnu eksperimentalnog i kvazi eksperimentalnog istraživanja jer se pokušava objasniti utjecaj jedne varijable na drugu te se isto tako testiraju statističke hipoteze. Koriste se podaci koji su već prikupljeni iako ne nužno prikupljeni s ciljem istraživanja. Istraživač retrospektivno ispituje utjecaj nekog događaja koji se prirodno dogodio na rezultat i pokušava utvrditi uzročno posljedičnu povezanost ili korelaciju. Nije moguće odabrati slučajan uzorak jer su se pojave već dogodile, a to se smatra ograničenjem ove vrste istraživanja, no iz istog razloga nije potrebno unaprijed tražiti pristanak ispitanika kao za eksperiment.

Istraživanje tipa *ex post facto* se koristi za proučavanje skupina koje su imale isto iskustvo osim jednog uvjeta (Cohen i ostali, 2013). Postoje kontrolne i eksperimentalne skupine, a na eksperimentalnoj je primijenjen uvjet čiji se utjecaj ispituje. Obzirom da nije bilo stroge manipulacije varijablama niti slučajnog rasporeda po skupinama treba biti oprezan kod donošenja općenitih zaključaka. Ponekad se može reći da je ovaj tip istraživanja korisniji od eksperimentalnog obzirom da se provedbom eksperimenta stvara umjetna situacija. Ipak, ne

možemo u potpunosti biti sigurni da smo identificirali stvarni faktor koji je utjecao na rezultat niti da je to bio jedini faktor. Raspored ispitanika u ujednačene skupine može biti problematičan, posebno jer se može dogoditi smanjivanje uzorka.

Istraživanje *ex post facto* provodi se u nekoliko faza (Cohen i ostali, 2013). Najprije je potrebno definirati problem ili predmet istraživanja i istražiti literaturu. Zatim se postavljaju hipoteze te bira uzorak i metode prikupljanja podataka. Potrebno je klasificirati podatke te prikupiti dva skupa podataka. U prvom su svi podaci o faktorima koji su uvijek prisutni kad se dogodi očekivani rezultat, a u drugom su podaci o faktorima koji su uvijek prisutni kad se ne dogodi rezultat. Zatim se uspoređuju ta dva skupa podataka, provodi analiza te interpretiraju rezultati. Prilikom odabira uzorka, odnosno raspoređivanja u kontrolne i eksperimentalne skupine bilo bi idealno da su te skupine ujednačene po nekom kriteriju koji se smatra bitnim za istraživanje. Međutim, to može biti teško ako istraživač ne zna koji su to faktori, a dodatan problem je što se može smanjiti uzorak. Alternativa rješavanja ovog problema je uvođenje dodatne varijable prema kojoj se mogu usporediti skupine, ali ta varijable ne mora biti kauzalno povezana sa zavisnom varijablom. Trebalo bi zatim odabrati skupine koje su homogene prema toj varijabli. Dodatna razina kontrole se može uvesti tako da se testiraju alternativne hipoteze koje bi mogle objasniti rezultate istraživanja.

Kod istraživanja primjene tehnologije u obrazovanju mogu se primijeniti različiti instrumenti, a jedna od prednosti takvih istraživanja je činjenica da često takve intervencije omogućuju automatsko prikupljanje korisnički generiranih podataka. Takvi podaci se zapisuju nenametljivo u naturalističkom okruženju, a kako je proces automatski onda ne zahtijeva dodatne troškove (Randolph, 2008). Primjer takvih zapisa (eng. logs) može biti vrijeme koliko dugo su učenici koristili nešto, koliko često su koristili pojedine dijelove i sl. Premda se podaci prikupljaju nenametljivo (za razliku kad istraživač izravno promatra), sudionici bi morali znati da se njihove aktivnosti bilježe. Nedostatak takvog prikupljanja podataka je što se može stvoriti preveliki skup podataka. Primjeri instrumenata kojima se prikupljaju podaci su također ispiti znanja, izravno promatranje, intervjui i upitnici. Kod istraživanja u poučavanju programiranja jedan od načina prikupljanja podataka kod longitudinalnog istraživanja je prikupljanje programa koje su ispitanici izrađivali (Gilmore, 1990).

Znanstvenici istražuju različite pristupe poučavanju objektno orijentiranog programiranja koji se nalaze u rasponu od pristupa visoke razine (odnose se na stvaranje i manipulaciju klasama i objektima) do onih niske razine koji se fokusiraju na pisanje kôda (Kunkle & Allen, 2016). Određivanje učinkovitosti različitih pristupa može biti problematično. Na primjer,

eksperimentalna istraživanja veličine učinka (eng. effect size) zahtijevaju bar jednu eksperimentalnu i kontrolnu skupinu, vrednovanje znanja prije i nakon intervencije kako bi utvrdili veličinu učinka. Bilo bi idealno kad bi se znanje moglo vrednovati nekim standardiziranim testom no takvi testovi iz područja iz informatike su rijetkost jer se tehnologija stalno mijenja, a poseban izazov predstavlja programiranje (Taylor i ostali, 2014). Jezici za poučavanje programiranja se povremeno mijenjaju tako da bi trebalo razvijati testove koji ne ovise o jeziku što je zahtjevan proces, ali ni to nije dovoljno obzirom da se i paradigme programiranja razlikuju. Zapravo je nemoguće osmisliti nešto što ne ovisi o programskom jeziku, npr. pseudokôd koji može obuhvatiti toliku raznolikost. Takva situacija stavlja istraživače u obrazovanju u nezgodan položaj jer nemaju standardizirani mjerni instrument za ispitivanje učinkovitosti inovativnih pristupa. Danielsiek i ostali (Danielsiek, Paul, & Vahrenhold, 2012) su započeli rad na testovima za predmet na prvoj godini studija vezan za algoritme i strukture podataka. Analizirali su 400 ispita kako bi utvrdili koje su česte pogreške, teške teme, identificirali ključne koncepte te definirali pilot provjeru s pitanjima (npr. višestruki odabir, nadopunjavanje i sl.). U nastavku istraživanja su primijenili pismenu provjeru znanja uz dodatne provjere u obliku intervjua i mapa koncepata (Paul & Vahrenhold, 2013). Primijetili su kako pismena provjera ne otkriva miskoncepcije jer na pitanje što je objekt ili što je klasa, studenti odgovaraju naučenim definicijama iz udžbenika.

Problem pismenih ispita i kolokvija na P2, SPA i OOP je što se njima zapravo ispituje sintaksa. Studenti moraju riješiti zadatke koji rade ispravno prema zahtjevima zadatka, sintaksno su ispravni te ispisuju traženi rezultat. Studenti će naučiti i najčešće algoritme koji se pojavljuju u zadacima, ali tu se većinom radi o nižim razinama Bloom-ove taksonomije. Više razine studenti na OOP počnu koristiti kad razmišljaju o svojim projektima, analiziraju problem i primjenjuju naučene koncepte na rješenje. Prema tome je bilo potrebno ispitati koje koncepte su zaista primijenili te koliko se takva primjena razlikuje po generacijama, ovisno o upotrebi OTTER-a.

Problem valjanosti pojavljuje se kad želimo generalizirati na temelju rezultata na neke druge uzorke ili programske jezike ili teorijske postavke. Dva važna tipa valjanosti su unutarnja i vanjska valjanost. Unutarnja valjanost nam osigurava da je naše objašnjenje promatranih razlika jedino moguće objašnjenje, a vanjska se odnosi na sposobnost stvaranja ispravnih generalizacija o posljedicama rezultata (Gilmore, 1990).

U idućem dijelu će se opisati najvažnije metode analize podataka korištene u istraživanju: faktorska analiza i strukturalno modeliranje te provedba pilot istraživanja koje je prethodilo glavnom istraživanju ove disertacije.

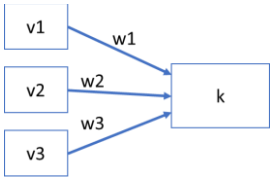
4.1 Faktorska analiza

Istraživanje u obrazovanju bavi se proučavanjem procesa učenja i poučavanja. Pri tome se pojavljuju varijable koje nisu izravno mjerljive, ali ipak imaju utjecaj. Na primjer, stavovi, motivacija i sl. su varijable za koje zaključujemo da postoje na temelju opažanja ponašanja. Faktorska analiza je statistička metoda koja omogućuje da se u većem broju varijabli među kojima postoji povezanost (imaju neke zajedničke karakteristike) utvrdi manji broj osnovnih varijabli koje nazivamo faktorima (Cohen i ostali, 2013). Često se kaže kako se varijable grupiraju prema zajedničkim karakteristikama. Faktori koji se utvrde tijekom analize nazivaju se latentne varijable. Omogućuje se identificiranje različitih varijabli koje se odnose na isti koncept. Latentne varijable predstavljaju neposredno nemjerljive vrijednosti. To znači da do njihove vrijednosti ne možemo doći izravno već preko drugih mjerljivih (eng. observable) varijabli.

Postoje dva glavna oblika faktorske analize: *eksploratorna faktorska analiza* (eng. exploratory factor analysis, EFA) i *konfirmatorna faktorska analiza* (eng. confirmatory factor analysis, CFA). EFA se koristi za ispitivanje prethodno nepoznatih grupiranja varijabli kako bi se pronašli uzorci sličnosti, induktivna je te iz podataka dolazimo do „teorije“. CFA služi za testiranje poznatog skupa faktora prema pretpostavljenom modelu grupiranja ili jednostavnijim riječima: koristimo je za testiranje modela koji je postavljen prema teoriji. CFA se temelji na ograničenjima postavljenima modelom (o odnosu mjerenih i latentnih varijabli), odnosno ako imamo teoriju koji indikatori (izmjerene varijable) određuju faktore (latentne varijable) onda ćemo krenuti najprije od modela te tražiti potvrdu u podacima. EFA ne postavlja početna ograničenja.

U kontekstu faktorske analize često se spominje i *analiza glavnih komponenti* (eng. principal component analysis, PCA). Premda se često poistovjećuju faktorska analiza i PCA se ipak razlikuju. Obje vrste analize bave se reduciranjem podataka, provode se u istom okruženju, rezultati izgledaju gotovo jednako, postupak je sličan (izvlačenje faktora, interpretacija, rotacija, odabir broja faktora). PCA je linearna kombinacija varijabli, a faktorska analiza model mjerenja latentnih varijabli. PCA ne sadrži latentne varijable i nije model već se računa na temelju varijabli (Kasper & Ünlü, 2013). Faktorska analiza je model koji predstavlja pojednostavljenje mjerenih podataka, ali ne reproducira podatke savršeno tako da može uključivati i greške. Ponekad se radi jednostavnosti PCA isto zove faktorska analiza. PCA koristi linearnu kombinaciju (otežani prosjek) skupa varijabli u stvaranju varijabli koje se nazivaju *komponentama* (eng. components). Cilj je dobiti optimalan broj komponenti,

optimalan broj mjerljivih varijabli za svaku komponentu kao i optimalne težine. U primjeru na slici ispod (Slika 4.1) prikazane su tri mjerljive varijable (*v*) s odgovarajućim težinama (*w*) koje pridonose komponenti *k*.



Slika 4.1. PCA komponenta

Faktorska analiza predstavlja model mjerenja latentne varijable koju ne možemo izravno izmjeriti jednom varijablom (npr. kognitivne sposobnosti, zdravlje) već se promatraju u kontekstu povezanosti s mjerljivim varijablama.

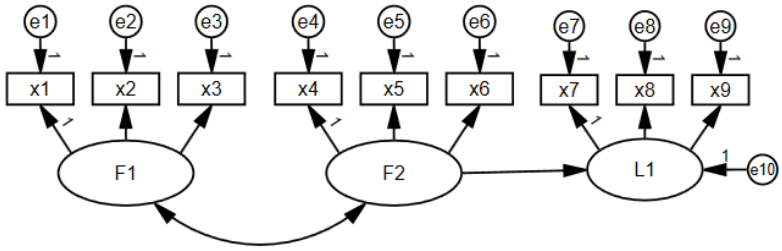
Ovdje ćemo provesti PCA, obzirom da želimo identificirati varijable se mogu grupirati. U prvom koraku istraživač mora provjeriti je li uzorak prikladan za provedbu analize. Veličina preporučenog uzorka u literaturi varira od preporučenih 30 do 300. Prikladnost podataka za analizu može se obaviti statističkim testovima kao što su: Bartlett-ov test (eng. Bartlett test of sphericity) i Kaiser-Mayer-Olkin (KMO) test. Bartlett-ov test bi trebao imati vrijednost $p<0.05$, a prvenstveno se koristi za situacije kad je broj slučajeva po varijabli 5 i manje. KMO test mjeri prikladnost uzorka, a mora biti bar 0.6 ili više. Vrijednost manja od 0.5 je neprihvatljiva, a vrijednost iznad 0.9 je izvrsna. Maksimalna vrijednost je 1. Prilikom izračuna KMO rade se korelacije parova varijabli. Za analizu se obično koristi neki od statističkih programa kao što su SPSS, Statistica, JASP i sl.

Faktorskom analizom dobivamo *opterećenja faktora* (eng. factor loadings) kojima se najbolje reproducira korelacija promatranih varijabli. U prvoj fazi se za *n* varijabli identificira *n* faktora, ali je cilj zadržati manji broj faktora koji objašnjavaju prihvatljivu razinu varijabilnosti korelacija. Granica ovisi o konkretnom istraživanju, što znači da se često određuje heuristički, ali možemo se orijentirati prema preporukama iz literature. Važno je razumjeti značenje dobivenih faktora voditi računa da ne postoji jedinstveno rješenje. Nakon što odaberemo manji broj faktora, često se provodi i rotacija. Često se koristi „varimax“ rotacija, posebno u istraživanjima u obrazovanju (Wetzel, 2011). Cilj rotacije je dodatno „udaljiti“ faktore odnosno postići da opterećenja faktora prema varijablama jedne grupe budu maksimalna, a minimalna za ostale. Na taj se način smanjuje korelacija između pojedinih faktora.

4.2 Strukturalno modeliranje

Metode modeliranja koriste se za proučavanje pojava koje se sastoje od složenog skupa varijabli. Strukturalno modeliranje (eng. Structured Equation Modeling, SEM) koristi se kod proučavanja odnosa izmjerenih i latentnih varijabli (Cangur & Ercan, 2015). Može se koristiti za analizu složenih teoretskih modela odnosno ispitivanje je li teoretski model podržan podacima. SEM se opisuje kao spoj CFA i višestruke linearne regresije (eng. multiple linear regression) jer više spada u konfirmatorne tehnike iako se može koristiti i eksplorativno (Schreiber, Nora, Stage, Barlow, & King, 2006). Sturgis (2019) opisuje SEM kao integraciju mjerenja, faktorske analize latentnih varijabli, analize putanje, regresije i simultanih jednadžbi. Preporučena veličina uzorka za SEM analizu je minimalno 200, osim u situacijama kad nema latentnih varijabli, kod fiksnih opterećenja faktora (svi postavljeni na 1), ako su modeli jednostavni, ako postoje jake korelacije u modelu te kod modela gdje nije moguće ostvariti taj broj (Kenny, 2011).

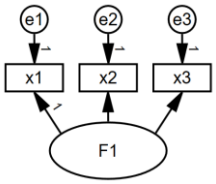
U SEM terminologiji se koriste pojmovi *endogenih* (eng. endogenous) i *egzogenih* (eng. exogenous) varijabli. Naziv endogena (izlazna) varijabla odgovara zavisnoj varijabli (uzrokovana varijablama unutar modela), a egzogena (ulazne) varijabla odgovara nezavisnoj varijabli (Low Stanić, 2014; Schreiber i ostali, 2006). Endogene i egzogene varijable mogu biti *izmjerene* (eng. observed) ili *neizmjerene* (eng. unobserved). Na slici ispod (Slika 4.2) je primjer općenitog SEM modela.



Slika 4.2. Primjer SEM modela

Varijable nacrtane u obliku pravokutnika su mjerljive, a varijable u obliku elipse su nemjerljive (latentne). Sve varijable koje imaju jednosmjernu ulaznu strelicu su endogene, a ostale su egzogene. Jednosmjerna strelica predstavlja regresiju. Varijabla *L1* je endogena i latentna. Dvostruka strelica koja povezuje F1 i F2 predstavlja kovarijancu. Varijable e_1, … e_{10} predstavljaju pogreške u mjerenju. Na jednostruke strelice se obično piše opterećenje faktora. Treba napomenuti da su u primjeru modela sa slike poznate samo vrijednosti varijabli u pravokutnicima i da treba raditi procjenu svih ostalih varijabli uključujući i opterećenja. Iz tog

razloga se za svaku latentnu varijablu obično postavlja ograničenje 1 na jednu strelicu (opterećenje) te se u skladu s podacima radi procjena nepoznatih vrijednosti. Time se smanjuje broj nepoznatih parametara koje je potrebno procijeniti. Model može biti *točno identificiran* (eng. just-identified) što znači da broj poznatih parametara odgovara nepoznatim vrijednostima, odnosno model s nula stupnjeva slobode (eng. degree of freedom, *df*) (Kenny, 2011). Za p*od-identificiran* (eng. under-identified) model nije moguće procijeniti sve parametre. Na primjer, za jednadžbu: $x + 3y = 7$ kažemo da je *neidentificirana*, dok za jednadžbu $4 + 3y = 7$ kažemo da je *identificirana* jer možemo dobiti nepoznati parametar ($y = 1$). U SEM-u su poznate varijance i kovarijance srednjih vrijednosti izmjerenih varijabli, a nepoznanice su parametri koje procjenjujemo. Model koji ima više poznatih parametara od nepoznatih naziva se *pre-identificirani* (eng. over-identified). Za primjer modela na slici ispod (Slika 4.3) treba procijeniti tri varijance pogreške (e_i), dva opterećenja faktora (jedan je fiksiran na 1) i jednu latentnu varijancu (*F1*), što je ukupno šest nepoznatih parametara. Za šest nepoznatih parametara nam je potrebno bar tri poznate varijable (x_i) (Sturgis, 2019). Ako je *s* broj poznatih varijabli, onda se prema formuli: $s(s + 1) / 2$ dobije broj parametara koji se mogu procijeniti.



Slika 4.3. Točno identificirani model

Prema tome, možemo procijeniti model, ali kako je to točno identificirani model onda ne možemo procijeniti sukladnost ($df = 0$). U ovakvim situacijama možemo dodati nove poznate informacije ili ukloniti neke nepoznate parametre dodavanjem dodatnih ograničenja (npr. postavljanje jedinica umjesto opterećenja faktora).

SEM analiza se provodi pomoću odgovarajućih programskih alata u kojima je najčešće uključena grafička komponenta te je na taj način jednostavnije modeliranje. Primjeri poznatih alata su LISREL, Amos, Mplus i sl. Prednost takvih alata je vizualni prikaz u obliku dijagrama umjesto matematičkih jednadžbi. Sturgis (2019) smatra kako se SEM može promatrati kao *analiza putanje* (eng. path analysis, PA) s latentnim varijablama. PA se može zamisliti kao prikaz teoretskog modela u obliku dijagrama, a predstavlja povezanost izmjerenih varijabli. Općenito se PA koristi za izmjerene varijable, a ne za latentne. Schreiber i ostali (2006) ističu kako se PA temelji na restrikcijama koje uključuju pretpostavku da je mjerenje bez pogreške, da greške nisu međusobno korelirane te da je tijek uvijek jednosmjeran (nema povratnih petlji

prema varijablama). Sve navedeno se rijetko događa kod istraživanja u obrazovanju. Kod istraživanja u obrazovanju se također često pojavljuju varijable koje nisu izravno mjerljive (npr. percepcije studenata, interesi, anksioznost kod rješavanja testova i sl.). To su složeni konstrukti koji najčešće imaju više od jednog indikatora (npr. rezultate upitnika i sl.). Iako je fokus SEM analize istraživanje veza među latentnim varijablama, može se koristiti i za testiranje eksperimentalnih podataka (Schreiber i ostali, 2006). Izrada SEM modela u situaciji kad nemamo neku početnu teoriju također može poslužiti da uočimo kako se manipulacijom parametara ili veza među varijablama mijenjaju vrijednosti i odnosi među varijablama. Sturgis (2019) ističe tri istraživačke primjene SEM-a. Prva primjena je kod analize složenih konstrukata koje je inače teško mjeriti ili se pri mjerenju događaju pogreške, a druga za modeliranje složenih „sustava" veza među varijablama umjesto promatranja jedne zavisne varijable sa skupom prediktora. SEM je također dobar odabir za provjeru posrednog (eng. mediated) utjecaja neke varijable.

Često se za procjenu nepoznatih parametara koristi metoda *maksimalne vjerojatnosti* (eng. maximum likelihood, ML). ML kroz više iteracija procjenjuje parametre modela, odnosno mijenja vrijednosti nepoznatih parametara sva dok ne dobije maksimalnu vrijednost vjerojatnosti za odabrani uzorak podataka. Pretpostavka metode je da su varijable kontinuirane i zadovoljavaju pretpostavku multivarijantne normalnosti. U protivnom treba koristiti neku drugu metodu. Pre-identificirani model je poželjniji za SEM analizu i računanje maksimalne vjerojatnosti te samim tim i mjere sukladnosti modela. Konstantna varijanca je povezana s pretpostavkom normalne distribucije podataka. Kako bi postigli konstantnu varijancu, moguće je transformirati podatke ili upotrijebiti neku drugu tehniku kao što je GLS.

Poopćena metoda najmanjih kvadrata (eng. Generalized Least Squares, GLS) (Cangur & Ercan, 2015) je također tehnika za procjenu nepoznatih parametara u linearnoj regresiji i SEM analizi gdje postoji određen stupanj korelacije među odstupanjima (eng. residuals) u regresijskom modelu. Za podatke kod kojih postoji velika *asimetričnost* (eng. skewness) i *spljoštenost* (eng. kurtosis) ML i GLS će imati pogreške. ML je manje osjetljiva na veličinu uzorka i spljoštenost od GLS, a GLS zahtijeva dobro specificirane modele te se može koristiti za manje uzorke (Olsson, Foss, Troye, & Howell, 2000).

Ako zbog pretpostavke o multivarijantnoj normalnosti nisu primjenjivi ML i GLS onda se može koristiti metoda asimptotske procjene neovisne o distribuciji (eng. Asymptotically Distribution Free, ADF) koja daje asimptotski nepristrane (eng. unbiased) procjene za hi-kvadrat, parametre te standardne pogreške. Ograničenje je što zahtijeva veliki uzorak, čak oko 5000 (Gold, Bentler,

& Kim, 2003). Drugo rješenje za rad s podacima koji ne zadovoljavaju kriterij normalnosti je primjena metode neopterećenih najmanjih kvadrata (eng. Unweighted Least Square, ULS). Veličina uzorka je u rasponu od 250-500, a spominje se i pravilo za računanje veličine uzorka prema broju varijabli (10 x broj varijabli) (Schumacker & Lomax, 2004). Olsson i ostali (2000) su usporedili ML, GLS i WLS (eng. Weighted Least Square) metode na lošim modelima kao i na podacima koji nisu normalno distribuirani te preporučuju da se ipak provede više metoda radi triangulacije. WLS je bolji na velikim uzorcima (više od 1000). Problem s metodom ULS je što ovisi o mjerenju (eng. scale) varijabli tako da u nekim slučajevima neće dati rezultat te se tada koristi SFLS (eng. scale-free least squares estimation, SFLS) koja ne ovisi o mjeri. Metode GLS i ML također ne ovise o mjeri, što znači da će u slučaju transformacija varijabli rezultati biti povezani.

4.2.1 Mjere sukladnosti modela

Važan dio nakon stvaranja i testiranja SEM modela je ispitivanje koeficijenata koji pokazuju sukladnost modela (eng. model fit), odnosno je li model u skladu s podacima i može li se modelom reproducirati podatke. Dobar model je onaj koji je dovoljno konzistentan s izmjerenim podacima. Postoji veliki broj indeksa sukladnosti (eng. fit indeks) ili mjera sukladnosti tako da se teško odlučiti za jedan, a dosta je kontroverzi oko toga. Na primjer neki istraživači ne vjeruju uopće u postavljanje granica za indekse te da je najvažnije promatrati χ^2 (hi-kvadrat) (Barrett, 2007). Pogrešno je odabrati samo jedan koji pokazuje ono što smo htjeli dobiti bez objašnjenja za ostale indekse koji se često objavljuju. Indekse sukladnosti možemo podijeliti na (Kenny, 2011): inkrementalne, apsolutne i komparativne. Inkrementalni indeksi se općenito kreću u intervalu od 0 do 1, gdje 0 predstavlja najgori model, a 1 najbolji (Kenny, 2011). Apsolutni indeksi pretpostavljaju da je 0 najbolji, a što je indeks veći to je model lošiji. Komparativni su korisni samo ako imamo dva ili više modela kako bi ih mogli usporediti te je općenito bolji onaj model koji ima manji indeks. Ponekad se komparativni indeksi spominju zajedno s apsolutnima, ali Kenny (2011) naglašava kako bi ipak bilo dobro posebno izdvojiti one indekse koji ne zahtijevaju postojanje više od jednog modela.

Za modele od 75 do 200 slučajeva hi-kvadrat je općenito prihvatljiva mjera, ali za veliki broj (iznad 400) slučajeva hi-kvadrat je gotovo uvijek statistički značajan, a što su veće korelacije u modelu, manja je mjera sukladnosti. Druga mjera je omjer χ^2/df gdje je *df* broj stupnjeva slobode. Međutim, problem je što nema standarda koja granica predstavlja dobar ili loš model tako da neki autori smatraju prihvatljivim modelom ako je omjer manji od 5 dok neki postavljaju gornju granicu prihvatljivosti na 3 ili 2 (Cangur & Ercan, 2015; Hooper, Coughlan,

& Mullen, 2008). Za ostale indekse ćemo umjesto prijevoda koristiti kratice, obzirom da ih po tome lakše raspoznajemo. Podaci koji nisu normalno distribuirani povećavaju hi-kvadrat (Kenny, 2011).

Drugi indeks koji se često pojavljuje je RMSEA (Root Mean Square Error of Approximation) koji nam kaže koliko dobro model s nepoznatim, ali optimalno odabranim procjenama parametara odgovara matrici kovarijanci dane populacije (Hooper i ostali, 2008). Granica za RMSEA je u novije vrijeme smanjena, ali okvirno se za interval (0.01, 0.05) smatra izvrsnom sukladnosti, (0.05, 0.08) je dobra sukladnost, (0.08, 0.10) osrednja (ni dobro ni loše), a sve iznad 0.10 je loše. Međutim, Kenny (2011) ističe kako su te granice za populaciju, što ne mora biti isto za uzorak tako da treba promatrati i vrijednost parametra PCLOSE što je skraćenica od „p of close fit“ ili p za blisku sukladnost. Parametar PCLOSE bi trebao biti veći od 0,05.

Indeks GFI (Goodnes of Fit Index) (Hooper i ostali, 2008) je osjetljiv na veličinu uzorka pa ga neki autori ne preporučuju. Preporučena donja granica je 0,09.

RMR (Root Mean square Residual) indeks odnosi se na kvadrat odstupanja matrice kovarijanci uzorka i hipotetskog modela (Hooper i ostali, 2008). Raspon za RMR se računa prema svakom indikatoru tako da ga je nezgodno interpretirati kad koristimo indikatore koji imaju različite skale (npr. Likertove skale od 1-5 i 1-7). SRMR (Standardised Root Mean square Residual) je standardiziran tako da su njegove vrijednosti iz intervala od nula do jedan, a vrijednosti manje od 0,05 smatraju se dobrim. Smatra se da su vrijednosti do 0,08 prihvatljive.

Indeks TLI (Tucker Lewis Indeks) je još poznat pod nazivom NNFI (Non-normed Fit Indeks) je poboljšanje Bentler-Bonett-ovog indeksa NFI (Normed Fit Index). Problem s NFI je što ovisi o dodavanju parametara pa je prema tome veći što se više dodaje parametara. TLI nema taj problem, a ovisi o prosječnim korelacijama podataka. Ako imamo više varijabli koje nisu korelirane onda TLI može pasti niže od 0,90, a preporučeno je da bude veći od 0,95.

AIC (Akaike Information Criterion) je primjer komparativne mjere sukladnosti, a ima smisla kao što je već rečeno samo ako imamo dva ili više modela jer onda možemo usporediti koji je bolji. Prednost je što se može izračunati i za točno identificirane modele kao i za pre-identificirane modele. AIC ovisi o broju parametara, a BIC (Bayesian Information Criterion) koji također spada u komparativne indekse vodi računa i o veličini uzorka (povećava „kaznu“ (eng. penalty) kako se povećava uzorak).

Računanje indeksa istraživači obično prepuštaju alatima koje koriste pri analizi. Ovdje je navedeno nekoliko tipičnih primjera indeksa koje ćemo promatrati kao orijentaciju. Premda neki autori odbacuju indekse i postavljene granice (Barrett, 2007), drugi preporučuju oprez pri strogom prihvaćanju granica kako se ne bi dogodila pogreška odbacivanja ispravnog modela (Marsh, Hau, & Wen, 2004).

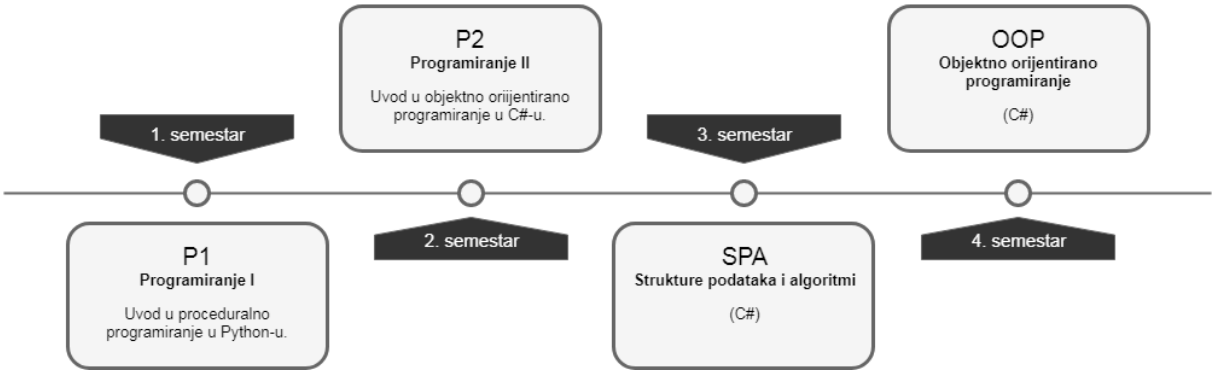
Nakon izrade modela, pokreće se procjena nepoznatih parametara prema odabranoj metodi. Ako je procjena bila uspješna onda se provjeravaju indeksi sukladnosti. Indeksi se mogu popraviti promjenom modela ili izbacivanjem ekstrema. Potrebno je provjeriti Mahalanobis udaljenost te ukloniti ekstremne vrijednosti. Međutim, pri tome treba biti oprezan jer se mogu ukloniti bitni podaci. Drugi način je promatranje indikatora promjene (eng. modification indices) koji sugeriraju za koliko će se promijeniti hi-kvadrat ako dodamo kovarijance među određenim elementima. Možemo eventualno pobrisati neke dijelove modela ili povezati kovarijancom varijable pogreške. Hopper i ostali (2008) ističu kako nije preporučljivo u potpunosti se osloniti na indikatore kako bi poboljšali model niti dodavati kovarijance među pogreškama obzirom da to možda pokazuje kako nešto drugo nije u redu s modelom. Ako postoji prihvatljivo objašnjenje zašto bi povezali pogreške onda je to prihvatljivo.

Pretpostavka glavnog dijela istraživanja je da će studenti izrađujući projekte pomoću OTTER okvira primjenjivati više koncepata objektno orijentiranog programiranja od onih koji su izrađivali projekte bez okvira. Kao što je već ranije spomenuto, OTTER se zapravo temelji na istoj tehnologiji (Windows Forms) kao i ostali projekti koji su izrađeni bez okvira. Svi ostali koji su koristili XNA, Unity ili su bili posebno napredni (primjer AlfonsRPG, Slika 3.7) su izbačeni iz analize. Primjer AlfonsRPG igre temelji se na ideji OTTER okvira, ali je student promijenio gotovo sve postojeće elemente i dodao dosta vlastitih te bi uključivanjem tog projekta zapravo išli u korist primjene OTTER-a, ali takav projekt premda izniman primjer onog što smo željeli postići ipak spada u iznimke. Prema pretpostavci koju očekujemo mogli bi postaviti SEM model i ispitati odgovara li podacima, ali kako se ne radi o testovima već o analizi projekata otvorenog tipa najprije ćemo obaviti PCA analizu nad podacima kako bi sami podaci sugerirali model te prema tome izraditi model pa ćemo stoga koristiti SEM eksplorativno.

4.3 Pilot istraživanje

Studenti prve godine na Prirodoslovno-matematičkom fakultetu u Splitu (PMFST) susreću se s istim problemima kao većina početnika u programiranju. Većina studenata koji upisuju uvodne

predmete iz programiranja su apsolutni početnici. Međusobno se razlikuju po smjeru kojeg su upisali: matematika (M), fizika (F), informatika (I), inženjerska fizika (iF), matematika i informatika (MI), informatika i tehnika (IT). Svima je zapravo prvi susret s programiranjem predmet *Programiranje I* (P1) gdje se upoznaju s osnovama proceduralnog programiranja u programskom jeziku Python (Krpan & Bilobrk, 2011). U drugom semestru, studenti prelaze na objektno – orijentiranu paradigmu te se upoznaju s programskim jezikom C#. Jedna od važnih stvari koja se može primijetiti na slici je primjena programskog jezika C# kod više predmeta i različitih semestara. Naime, manji broj jezika koje početnici uče može biti prednost, ali neki autori preporučuju više pod uvjetom da su početnici sposobni usvojiti ključne koncepte jezika (Hadjerrouit, 1998). Ako početnici nemaju dovoljno znanja iz konkretnih programskih jezika, onda se neće dobiti zadovoljavajući rezultati. Rješavanje problema ne mora ovisiti o konkretnom jeziku, ali početnici moraju negdje vježbati i nedovoljno poznavanje jezika može predstavljati problem (Kiper & Abernethy, 1996). Programiranje ili pisanje programa ovisi o paradigmi programiranja te neki autori ipak smatraju da bi početnici tijekom školovanja ipak trebali upoznati više paradigmi.



Slika 4.4. Pregled po semestrima

Studenti se susreću s početnom proceduralnom paradigmom, a počevši od drugog semestra počinju učiti objektno orijentiranu paradigmu. Studenti koji se obrazuju za buduće nastavnike informatike bi svakako trebali upoznati više paradigmi. Kao što je već prije spomenuto, nema nekog univerzalnog rješenja s kojom paradigmom zapravo treba početi. Na primjer, ako se sjetimo da prijelaz iz proceduralne u objektnu paradigmu može biti težak, onda bi se moglo razmisliti o tome da se odmah krene s objektno orijentiranom paradigmom, no s druge strane se isto tako događa da dio studenata ima problema s osnovnim algoritamskim konceptima te sintaksom (Hadjerrouit, 1998).

Možda iskustvo iz proceduralne paradigme predstavlja prepreku kod prijelaza, ali nastavnici toga moraju biti svjesni kako bi vodili početnike u daljnjoj izgradnji znanja. Studenti na PMFST tijekom predmeta P1 usvajaju osnovne koncepte programiranja te ne mogu zapravo postati eksperti u proceduralnom programiranju nakon samo jednog semestra iskustva što im u skladu s navedenim ne bi trebalo predstavljati toliku prepreku kod promjene paradigme. Kod izrade klasa u objektno orijentiranoj paradigmi (predmet P2) studenti pišu metode koje sadrže osnovne algoritamske koncepte slijeda, grananja i ponavljanja, a to su ključni elementi predmeta P1, odnosno proceduralnog pristupa kojeg su prošli u prethodnom semestru. U skladu s navedenim, studenti zapravo grade novo znanje uvođenjem novih koncepata objektno orijentirane paradigme na temelju postojećih osnovnih koncepata programiranja. Tijekom predmeta P2, studenti će upoznati sintaksu drugog jezika, ponoviti osnovne algoritamske strukture koje su radili na prethodnom predmetu te proći iste koncepte u novoj sintaksi. Obzirom da C# u prvom redu podržava objektno orijentiranu paradigmu onda je nemoguće izbjeći neke nove koncepte kao što su na primjer objekti i klase. Čim se otvori novi konzolski projekt, studenti vide imenski prostor, klasu *Program* i metodu *Main()*. Prema tome, studenti će naučiti prepoznati te koncepte u programu, napisati ih sami uz odgovarajuće upute, ali neće biti sposobni osmisliti dizajn aplikacije u kojoj bi za dani problem sami odredili koje su im klase potrebne. To je zadatak predmeta OOP u kojem ponovo prolaze koncepte klasa i objekata uz primjenu te upoznaju ostale ključne koncepte objektno orijentiranog programiranja koje smo identificirali ranije. Važno je i da se studenti upoznaju s konceptima odabranog programskog jezika što ranije (Hadjerrouit, 1998), što je u našem slučaju C# u drugom semestru.

Analizirali smo uspješnost studenata na temeljnim predmetima iz programiranja (Krpan, Mladenović, & Rosić, 2014). Prikupljeni su podaci za sve upisane studente tijekom tri akademske godine, a temeljni predmeti iz programiranja na PMFST koje smo promatrali su: Programiranje I, Programiranje II, Strukture podataka i algoritmi (SPA), Objektno orijentirano programiranje (OOP), Arhitektura računala (AR), Baze podataka (BP), Rješavanje problema (RP), Programiranje mrežnih aplikacija (PMA). Obzirom da se matematičke sposobnosti često smatraju važnima za programiranje, analizirani su podaci za dva temeljna matematička predmeta: Matematika I (M1) i Matematika II (M2) u prvom i drugom semestru. U tablici ispod (Tablica 4.1) nalazi se pregled prolaznosti za odabrane predmete. U stupcu s ukupnim brojem studenata uključeni su svi student koji su upisali predmet prvi put, ukupno za sve tri godine.

Tablica 4.1. Postotak prolaznosti

Predmet	2010/11	2011/12	2012/13	Sve godine	Broj studenata
Programiranje I	60.87%	53.67%	48.47%	**53.62%**	511
Programiranje II	51.08%	44.00%	39.09%	**44.03%**	511
Strukture podataka i algoritmi	86.96%	87.69%	72.73%	81.92%	177
Objektno orijentirano programiranje	89.32%	79.31%	73.85%	81.96%	255
Arhitektura računala	77.50%	75.00%	64.15%	73.06%	193
Baze podataka	100.00%	98.44%	83.56%	93.12%	189
Rješavanje problema	84.78%	58.33%	75.86%	78.16%	87
Programiranje mrežnih aplikacija	95.12%	82.61%	80.65%	86.44%	118
Matematika 1	50.00%	52.52%	21.15%	**39.55%**	397
Matematika 2	42.16%	39.57%	16.03%	**30.98%**	397

Očito je da je uspješnost, odnosno prolaznost na programerskim predmetima P1 i P2 najlošija, a slično je i na predmetima M1 i M2. Obzirom na nisku prolaznost među promatranim predmetima, provedena je korelacijska analiza konačnih ocjena (Cohen, Manion, & Morrison, 2007). Korelacijska analiza se koristi za provjeru povezanosti ili korelacije među varijablama. Prije korelacijske analize provjerena je distribucija podataka te je utvrđeno da distribucija nije normalna. Događa da uspješnost ljudi u nekom području ne mora biti normalno distribuirana već spada u „power law“ distribuciju (Bersin, 2014; O'Boyle Jr. & Aguinis, 2012). Takva distribucija se događa u situacijama kad ima najveći broj najboljih ili najlošijih dok broj ostalih postupno opada. Kod početnih predmeta iz programiranja i matematike je upravo „power law“ distribucija s tim da je najveći broj najlošijih.

Sve korelacije koje su u tablici (Tablica 4.2) označeni zvjezdicom (*) su statistički značajne za $p<0.05$, a to znači da postoji statistički značajna korelacija među svim ocjenama. Jednostavna interpretacija bi mogla biti da je većina studenata koji su položili P1 isto tako položila i P2 (pozitivna korelacija 0.802). Međutim, bilo bi pogrešno zaključiti da postoji uzročno posljedična veza obzirom da uspjeh u jednom predmetu ne uzrokuje uspjeh u drugom. Prema tome, postoje indicije da prijelaz na drugi jezik ne čini razliku.

Tablica 4.2. Analiza korelacija

Predmet	P1	P2	M1	M2
P1	1.000	0.802*	0.655*	0.626*
P2	0.802*	1.000	0.685*	0.731*

M1	0.655*	0.685*	1.000	0.747*
M2	0.626*	0.731*	0.746*	1.000

Predmeti koje smo promatrali se smatraju zahtjevnima iako se može reći da nakon prvog semestra prema rezultatima u tablicama to baš i nije tako. Činjenica je da studenti koji nisu položili P1 i P2 nisu bili u mogućnosti upisati naprednije predmete u idućim semestrima. P1 i P2 su morali biti upisani skupa jer studenti upisuju cijelu godinu. Usporedbom broja studenata vidljivo je da ih je na kasnijim predmetima bitno manje.

Studenti za svaki predmet ispunjavaju anketu za vrijeme trajanja nastave. Anketu neovisno provodi sveučilište. Posebno smo analizirali pitanje koje je vezano za percepciju ocjena koje očekuju iz nekoliko odabranih predmeta. Zanimljivo je kako studenti koji su ispunjavali anketu nisu očekivali negativne ocjene iako je veliki broj studenata na kraju dobio negativne ocjene (Krpan, Mladenović, & Rosić, 2014). Međutim, ograničenje istraživanja je u činjenici što su se anketa provodila na papiru sa studentima koji su se tom prilikom zatekli u učionicama na nastavi te je ankete ispunio prilično manji broj od stvarnog broja upisanih. Dio studenata je jednostavno prestao dolaziti jer su odustali od daljnjeg sudjelovanja u nastavi, ali nisu bili ispisani sa studija jer su naravno imali pravo upisati predmet ponovo. Dodatno ograničenje je anonimnost anketa tako da nije bilo moguće povezati očekivane ocjene sa stvarnim ocjenama studenata, a zbog malog broja ispunjenih anketa grupe upisanih i ispunjenih nisu bile usporedive. Na primjer, na predmetu P2 u anketama je 14 studenata očekivalo ocjenu izvrstan, a na kraju je 13 studenata dobilo tu ocjenu. Ne možemo znati da li se radi o istim studentima.

Korelacijska analiza pokazala je visoku pozitivnu povezanost između uvodnih predmeta iz programiranja (P1 i P2) s matematičkim predmetima na istoj godini (M1 i M2) što potvrđuje istraživanja i drugih istraživača. Mathews u doktorskoj disertaciji (Mathews, 2017) na temelju provedenog istraživanja zaključuje kako postoji korelacija između uspjeha u matematici s učenjem programiranja te kako ocjene iz matematičkih predmeta mogu biti jedan od prediktora uspješnosti u programiranju. Iz navedenog možemo zaključiti i kako bi trebalo detaljno analizirati kurikulume predmeta iz programiranja kako bi se ispitalo da li se možda u zadacima iz programiranja favorizira „matematički“ način razmišljanja, odnosno jesu li zadaci više matematički problemi.

Na predmetima nakon prve godine upisani su studenti koji su uspješno prošli prve prepreke, no dio njih i dalje nije uspješan. Razlog je moguće površno usvojeno znanje tijekom početnih predmeta ili su ti predmeti bili vrhunac njihovih sposobnosti.

Pristup poučavanja programiranja pomoću izrade igara razmatran je s ciljem povećavanja motivacije za učenje programiranja. Studenti smjera *Informatika* su prema redu predavanja imali više informatičkih predmeta u odnosu na ostale smjerove te smo razmatrali mogućnosti kako poboljšati rezultate na početnim predmetima iz programiranja. Problem u početku je nedostatak predznanja, a samim tim i mogućnosti izrade bilo kakvih složenijih stvari kao što su igre. Odabir programskog jezika Python ili C# za izradu igara povlači za sobom potrebu za uključivanjem dodatnih biblioteka i samim tim dodatan skup koncepata koje bi studenti trebali usvojiti. Nije bilo realno očekivati da će studenti koji već imaju problema s usvajanjem osnovnih koncepata programiranja uspjeti usvojiti još veći skup koncepata i još širu sintaksu, pored rješavanja problema koji se tiču same logike igre bez obzira na motivacijski utjecaj igara. Za efekt motivacije, studenti ipak moraju napraviti nešto što mogu vidjeti, a dostizanje takve faze u profesionalnim programskim jezicima kao što su Python i C# nije ni malo jednostavno dok se studenti bore s osnovama. Nastavnici su uočavali kako studenti već nakon početnih vježbi gube motivaciju te više ne sudjeluju u nastavi, odnosno rješavanju zadataka s vježbi. To je bilo vidljivo podjednako na predmetu *Programiranje I*, ali i na predmetu *Programiranje II*.

Studenti su se tijekom predmeta *Programiranje II* u drugom dijelu semestra počeli ponovno uključivati u rad nakon uvođenja izrade grafičkog korisničkog sučelja. Činilo se da im se svidjela ideja grafičkih elemenata i interakcije u obliku događaja. Međutim, i dalje ostaje problem složene sintakse jezika koji se uvođenjem novih elemenata ne smanjuje.

Vizualni programski jezici minimiziraju ili u potpunosti eliminiraju probleme sintakse te se početnici mogu usmjeriti na eventualno zanimljivije elemente te pri tome usvojiti osnovne koncepte programiranja. U skladu s tim, uveli smo u prvom semestru vizualni programski jezik Scratch te kasnije njegov dijalekt BYOB. BYOB (Build Your Own Blocks) je razvijen na Sveučilištu Berkeley kao modifikacija izvornog kôda jezika Scratch iz tog razdoblja (konkretno verzija 1.4). Nakon te početne verzije napravljen je „Snap!“ u JavaScript jeziku koji se temelji na BYOB-u, ali kako je u potpunosti izmijenjena implementacija onda su mu promijenili i naziv. Istraživanje u tom području primjene poučavanja pomoću izrade igara započeli smo 2009. godine s prvim verzijama Scratch-a i njegovih dijalekata te smo idućih 5 godina promatrali učinak na uspjeh studenata. Upotreba okruženja za programiranje koji su namijenjeni mlađim početnicima, odnosno učenicima u osnovnim školama na poučavanje

odraslih (preddiplomski studij) može se smatrati neuobičajenim, ali kako početnici u bilo kojoj dobi imaju slične probleme, postavlja se pitanje zašto ne? Ako bi programiranje učinili zabavnijim ili pristupačnijim početnicima pretpostavka je da bi možda bili motiviraniji za sudjelovanje te bi to možda moglo imati pozitivan utjecaj na njihov konačni uspjeh.

Studenti koji upisuju preddiplomski studij Informatike na diplomskoj razini imaju mogućnost nastavka studija na istom smjeru koji je nastavnički. Obzirom da se radi o potencijalnim budućim nastavnicima (u slučaju njihovog nastavka studija na diplomskom smjeru), onda je važno napraviti pozitivan korak u početnom programiranju. Dodatna prednost je što su vizualni programski jezici isto tako potencijalni alati koje će koristiti u nastavi. Studenti su na studiju Informatike akademske godine 2008./2009. u drugom semestru imali predmet *Seminar iz programiranja II* na kojem je planirana primjena naprednijih koncepata. Obzirom da su studenti nakon završetka prvog semestra naslijedili iste probleme i u idućem semestru nije bilo preporučljivo uvesti naprednije koncepte. Naime, čini se da studenti nakon početnih predmeta iz programiranja i dalje „ne znaju“ programirati (McCracken i ostali, 2001). U spomenuti predmet uveli smo Python zbog jednostavne sintakse primjerene početnicima, a s druge strane je prednost u odnosu na QBASIC što se primjenjuje u industriji. Međutim, osim što je pojedinim studentima Python izgledao zanimljivo, učenje dodatne sintakse nije pomoglo kod motivacije. Iduće akademske godine je plan studija reorganiziran tako da se umjesto tog predmeta uvodi novi u prvom semestru prve godine pod nazivom *Informatički projekt I (IP1)* predviđen za izradu jednostavnih programerskih projekata. Teško je bilo očekivati da bi studenti koji imaju problema sa sintaksom pored osnovnih koncepata iz programiranja mogli samostalno napraviti projekte posebno jer je P1 u istom semestru. Prema tome, uvodi se vizualni programski jezik kako bi se koncentrirali samo na koncepte programiranja umjesto na sintaksu jezika.

Studenti su u vizualnom jeziku učili osnove programiranja pomoću izrade jednostavnih igara. Svake vježbe sastojale su se od izrade jednostavne igre koju je bilo moguće napraviti u okviru jednog bloka sata. Nakon prve godine iskustva rada u Scratch-u verzije 1.4 (ak. god. 2009./2010.) bolji studenti su imali prigovore na okruženje smatrajući ga ograničavajućim. Navodili su sljedeće probleme:

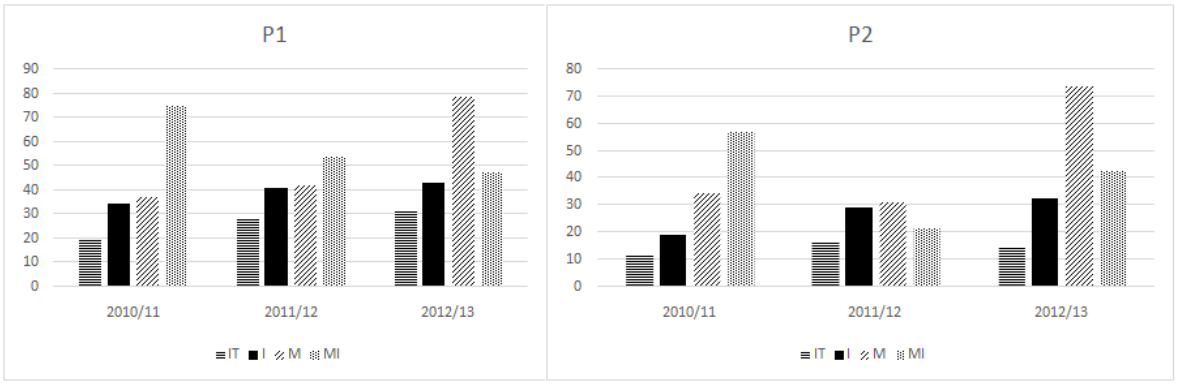
- ograničeni skup naredbi,
- nedostatak naredbi na koje su navikli iz drugih jezika (npr. petlja for),
- nemogućnost pretraživanja,
- nepreglednost kod složenijih programa,

- složenost računskih operacija.

Obzirom na prigovore, promijenjen je vizualni jezik u BYOB (Build Your Own Block). BYOB je podržavao sve funkcionalnosti Scratch-a iz te generacije, a dodatno je omogućavao izradu vlastitih blokova pomoću kojih je bilo moguće napraviti sve naredbe koje u Scratch-u nedostaju. Dodatno je postojala mogućnost kloniranja likova što bi studenti možda mogli povezati s idejom stvaranja objekata iz klase. Scratch od verzije 2.0 dobiva mogućnost izrade vlastitih blokova, no ti blokovi su i dalje prilično manjih mogućnosti u odnosu na BYOB. Oba jezika imaju iste načine rada s događajima (klik miša, pritisnuta tipka tipkovnice, unos korisnika i sl.).

4.3.1 Prikupljanje podataka

Proveli smo *ex post facto* istraživanje kod kojeg je odabran slučajan prigodan uzorak studenata (Kumar, 2012). Tijekom tri godine (2010 – 2013) ukupno je 727 studenata upisalo kolegij P1, a 784 studenta je upisalo kolegij P2. Odabrali smo studente koji su upisivali kolegije prvi put, obzirom da nisu imali predznanja iz samog sadržaja predmeta (kako to ne bi utjecalo na njihove rezultate). Nakon filtriranja podataka prema kriteriju prvog upisa, preostao je uzorak od 510 studenata. Obzirom da su predmeti na istoj godini (premda u različitim semestrima) i da studenti upisuju cijelu godinu (oba semestra), studenti koji upisuju P1 prvi put, sigurno su upisali i P2 prvi put. Usporedbom po studijskim programima (IT, I, M, MI, iF) pokazalo se da studenti koji su na matematičkim studijima (M i MI) ostvaruju bolji uspjeh od ostalih.



Slika 4.5. Studenti koji su položili P1 i P2

Promatrali smo nadalje studente studijskih grupa IT i I, obzirom da matematički studiji svakako ostvaruju bolji uspjeh, a studij inženjerske fizike (iF) upisivao je jako mali broj studenata. Studenti smjera I odabrani su u eksperimentalnu skupinu, a studenti smjera IT u kontrolnu, retrospektivno (Kumar, 2012). Eksperimentalna varijabla je primjena vizualnog jezika na kolegij IP1. Prikupljeni su podaci o prosjeku ocjena iz srednje škole, konačne ocjene iz predmeta P1, te konačne ocjene iz predmeta P2 kao i rezultati kolokvija i konačnog ispita iz P2.

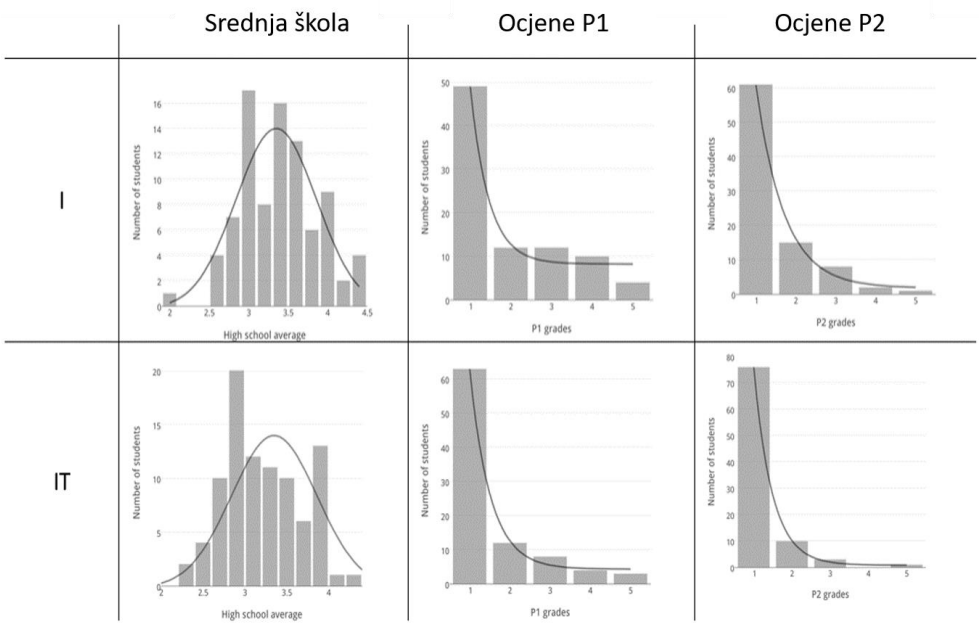
Obavljeno je daljnje filtriranje podataka eliminacijom podataka o sudionicima koji nisu imali potpune podatke. Preostaje uzorak od 452 studenta obzirom da za 57 studenata nije bilo moguće prikupiti prosjek ocjena iz srednje škole. Konačan uzorak za eksperimentalnu skupinu I i kontrolnu skupinu IT sadržavao je ukupno 202 studenta.

Studenti tijekom kolegija P1 uče proceduralno programiranje, te je jedna od pretpostavki bila da bi upotreba vizualnog programskog jezika trebala imati veći utjecaj na kolegij P2 gdje je okruženje u drugom dijelu semestra na sličan način vođeno događajima (eng. event-driven). BYOB je napravljen s namjerom proširenja, ali je isto tako predstavljen kao objektno orijentiran nudeći mogućnosti primjene složenih objektno orijentiranih koncepata kao što je primjerice nasljeđivanje. BYOB također sadrži blokove koji nisu nikad postojali u Scratchu, a kojima je bilo moguće dohvatiti pojedine atribute lika (npr. ime, dimenzije, sliku i sl.) te je također moguće spremati likove u varijable dok u Scratchu to nije bilo moguće. U programskom jeziku C# se na sličan način spremaju složeni objekti u varijable.

4.3.2 Analiza podataka i zaključak

Postavljene su nul-hipoteze za kontrolnu i eksperimentalnu skupinu:

H1: Ne postoji statistički značajna razlika u prosjeku ocjena iz srednje škole između kontrolne i eksperimentalne skupine.
H2: Ne postoji statistički značajna razlika u konačnim ocjenama na predmetu P1 između kontrolne i eksperimentalne skupine.
H3: Ne postoji statistički značajna razlika u konačnim ocjenama na predmetu P2 između kontrolne i eksperimentalne skupine.
Testirana je normalnost distribucije podataka za sve varijable primjenom Shapiro-Wilk W testa za sve tri promatrane godine zajedno i odvojeno po svakoj godini.



Slika 4.6. Distribucija ocjena

Promatrajući sve tri godine zajedno (u istom skupu podataka), svi podaci za prosjek ocjena iz srednje škole su bili normalno distribuirani ($p>0.05$) tako da je primijenjen T-test. Ocjene za P1 i P2 odgovaraju „power-law“ distribuciji (O'Boyle Jr. & Aguinis, 2012). Eksperimentalna skupina je ostvarila statistički značajno bolje rezultate ($p = 0.021$), te odbacujemo H1 i zaključujemo da postoji statistički značajna razlika između kontrolne (IT) i eksperimentalne skupine u prosjeku ocjena iz srednje škole.

Obzirom da ocjene iz P1 i P2 nisu normalno distribuirane, primijenjen je neparametrijski Mann-Whitney U test te su odbačene H2 i H3 ($p<0.05$).

U idućem koraku razdvojeni su podaci za svaku od tri akademske godine, obzirom i da su se neznatno razlikovali i sadržaji koji su obrađeni, ali također zbog prethodno utvrđene razlike među skupinama, ako se promatraju svi zajedno. Za svaku akademsku godinu testirane su iste hipoteze. Za svaku godinu posebno je prosjek ocjena iz srednje škole bio normalno distribuiran. Analizom podataka t-testom za prosjek ocjena iz srednje škole utvrđuje se sljedeće:

- za prvu godinu nisu utvrđeni statistički značajni rezultati ($p = 0.434$) te se odbacuje H1,
- za drugu godinu nisu utvrđeni statistički značajni rezultati ($p = 0.07$) te se odbacuje H1,
- za treću godinu nisu utvrđeni statistički značajni rezultati ($p = 0.12$) te se odbacuje H1.

Analizom testova i dobivenih rezultata zaključujemo da bi možda trebalo pokušati analizirati detaljnije svaku akademsku godinu, obzirom da je postojala neka inicijalna razlika koja se naknadno izgubila.

Ocjene po pojedinim godinama za P1 i P2 opet nisu bile normalno distribuirane te se primjenjuje Mann-Whitney W test sa sljedećim rezultatima:

- Prva godina: postoji statistički značajna razlika između IT1 i I1 u konačnim ocjenama iz P1 i P2 ($p = 0.045$ za P1, $p = 0.0002$ za P2).
- Druga godina: ne postoji statistički značajna razlika između IT2 i I2 u konačnim ocjenama iz P1 ($p = 0.045$), a postoji za P2 ($p = 0.012$).
- Treća godina: ne postoji statistički značajna razlika između IT3 i I3 u konačnim ocjenama iz P1 ($p = 0.073$), a postoji za P2 ($p = 0.002$).

Varijable prosjeka ocjena iz srednje škole i P1 smatraju se inicijalnim varijablama.
Prvi kolokvij iz P2 odnosio se na konzolske aplikacije pisane u programskom jeziku C#, a drugi kolokvij na grafičke aplikacije (Windows Forms). Obzirom da su grafičke aplikacije funkcionalno bliskije programima izrađenima u vizualnim programskim jezicima, detaljnije su analizirani podaci koji se odnose samo na te dijelove u akademskoj godini 2012/2013.
Nakon filtriranja podataka dobiven je uzorak za kontrolnu skupinu: IT=32 i eksperimentalnu skupinu: I=34. Ponovljenom analizom i testiranjem hipoteza utvrđuje se da ne postoji statistički značajna razlika u početnim uvjetima (prosjek ocjena iz srednje škole) između kontrolne i eksperimentalne skupine. Za rezultate prvog kolokvija (konzolske aplikacije) također se utvrđuje da ne postoji statistički značajna razlika između kontrolne i eksperimentalne skupine. Analizom drugog kolokvija koji se odnosi na grafičke aplikacije utvrđuje se da postoji statistički značajna razlika za $p = 0.049$, ali ipak obzirom na mali uzorak i vrijednost od p koja se mogla zaokružiti i na dvije decimale, ne možemo tvrditi da su ti rezultati značajni.
Većina istraživanja o GBL ima zajedničke elemente: povećanje interesa i motivacije za učenike različitih dobi. Obzirom da smo imali mogućnost oblikovati sadržaje za cijeli predmet u vizualnom programskom jeziku, istraživanje je moglo obuhvatiti dulje vremensko razdoblje. Istraživanjem smo ispitali utjecaj uvoda u GBL na konačnu ocjenu predmeta iz programiranja u kojem se učio C#. Tijekom prethodne analize, utvrđeno je kako je eksperimentalna skupina bila uspješnija na predmetu P2 u odnosu na kontrolnu ako se promatraju sve godine odvojeno. Međutim, razlike su minimalne, a studenti s matematikom u nazivu studija su i dalje bili bolji od ostalih. Uspješniji studenti su u vizualnim jezicima radili složenije igre, ali su brzo „prerasli“

okruženje koje ih je u određenom trenutku počelo ograničavati. Značajno je što su lošiji studenti sudjelovali u radu, nisu odustajali te im je bilo zabavno. Fokus istraživanja je bio na uspješnosti tako da su ostale napomene zapažanja.

Studenti koji su upisali studij informatike nakon diplomskog studija će moći raditi u školama kao učitelji informatike gdje će poučavati djecu programiranju. Ako ti studenti loše usvoje koncepte iz programiranja, posljedice će se odraziti na njihovim učenicima koji dalje u budućnosti mogu postati studenti.

Obzirom na sudjelovanje studenata na vježbama (rješavali su zadatke, ostajali su dovršiti rješenja čak i za vrijeme pauza) očekivani su bolji rezultati, odnosno jači utjecaj izrade igara u vizualnim jezicima na predmet P2, ali to se ipak nije dogodilo. Na primjer, očekivano je da će studenti prenijeti iskustva s događajima u vizualnim jezicima na grafičke aplikacije u C#-u, odnosno da će povezati kako su likovi u Scratchu zapravo objekti koji reagiraju na događaje (klikove mišem, pritisak tipke, …) isto kao što su to i kontrole na windows formama, da se kloniranje likova može promatrati kao stvaranje objekata iz klase i sl. Studenti ipak nisu prenosili naučene koncepte iz jednog okruženja i drugo na dovoljnoj razini da bi to imalo bitnog efekta na uspješnost učenja programiranja na predmetu P2.

Motiviranost i interes studenata tijekom rada na vježbama je ostavila pozitivan dojam na nastavnika tako da smo nastojali nastaviti na tom tragu i pomoći studentima pri prijenosu znanja iz vizualnih jezika u C#. Kako bi to mogli napraviti, započeli smo izradu i primjenu okvira OTTER kako bi studentima omogućili pojednostavljeno okruženje za izradu jednostavnih 2D igara sličnih onima koje su mogli raditi u Scratch-u i BYOB-u.

Pristup poučavanja programiranja u vizualnim programskim jezicima za studente na fakultetu nije baš uobičajen, pogotovo jer se radi o programskim jezicima koji su namijenjeni djeci. Zaključujemo da se mogu koristiti tijekom kraćeg vremena kao dodatno pojašnjenje nekih koncepata ili brzu izradu primjera. Iskustvo poučavanja u trajanju tijekom cijelog semestra pokazalo je kako bi trebalo nastaviti raditi na pozitivnim aspektima kao što je motivacija studenata, ali ipak uzevši u obzir i negativne stavove boljih studenata, a to znači prijelaz u tekstualno okruženje čim prije. Studenti koji studiraju za buduće nastavnike se moraju upoznati s takvim jezicima, ali ovdje je bio cilj da oni usvoje koncepte iz programiranja, a ne da nauče kako to primijeniti kao nastavnici.

Zaključak je kako bi studenti koji su apsolutni početnici mogli započeti prvo iskustvo s programiranjem u nekom od vizualnih jezika, ali bi im trebalo omogućiti što lakši prijelaz u neki od tekstualnih jezika koje će koristiti kasnije tijekom studija, kao što je to objektno

orijentirani programski jezik C#. Uspješniji studenti su smatrali kako programiranje u vizualnim jezicima nije „ozbiljno programiranje“. Povezivanje između vizualnih jezika bez obzira za koji uzrast su namijenjeni i „ozbiljnog“ C# jezika neće se dogoditi intuitivno i samostalno već je potrebno posredovati.

Studenti se obeshrabre pri susretu sa složenom sintaksom C#-a, a čak je i samo okruženje Microsoft Visual Studio prilično zahtjevno s mnoštvom opcija. Scratch i slični dijalekti imaju *nizak prag* kojeg učenici brzo prelaze, te *visoki strop* što im omogućuje izradu složenih projekata i napredovanje. Iz loših rezultata studenata i odustajanja na P2, složene i bogate sintakse jasno je da je početni prag za C# visok, razmak između prelaska praga ili usvajanja osnovnih koncepata do izrade projekata sličnih onima u Scratch-u – ogroman, a vremena za učenje je malo. Okruženje kao što je Greenfoot koji nudi kombinaciju vizualnog s tekstualnim za poučavanje programskog jezika Java omogućuje lakši spoj vizualnog i tekstualnog, ali ne može se koristiti za izradu profesionalnih aplikacija te prijelaz u neko drugo razvojno okruženje opet zahtijeva vrijeme učenja i prilagodbe. Korak kojeg početnici moraju napraviti za prijelaz iz Greenfoot-a u drugo razvojno okruženje se opet čini velik.

Obzirom da je C# odabrani jezik na našem fakultetu za više predmete, cilj je što prije prijeći u profesionalno okruženje i izrađivati programe u C# uz usvajanje OOP koncepata. Promjenom reda predavanja ukinut je predmet u kojem su studenti učili programiranje u vizualnim jezicima te dalje nije bilo moguće nastaviti s posredovanim prijenosom. Prema gore opisanom istraživanju, utjecaj je bio minimalan tako da je premješten fokus na predmet OOP u kojem studenti imaju probleme pri usvajanju OOP koncepata. Tijekom P2 su morali usvojiti sintaksu C#-a kako bi se na idućem predmetu mogli fokusirati na OOP koncepte, ali uočeno je kako studenti imaju problema s prijelazom na OO paradigmu te da sintaksa nije jedini problem.

4.4 Provedba analize objektno orijentiranih projekata

Kod istraživanja u obrazovanju se umjesto eksperimentima često služimo „ kvazieksperimentima“ jer je teže postići ekvivalentnost grupa potrebnu za eksperimentalno istraživanje. U takvim istraživanjima uzimaju se obrazovne skupine kao što su razredi, škole, studijske grupe i sl. kod kojih nije zajamčena ekvivalentnost jer se ne biraju slučajnim rasporedom ispitanika. Istraživanje ex post facto (od lat. post = poslije, factum = činjenica) je metoda koja se također koristi umjesto eksperimenta za ispitivanje hipoteza o uzroku i učinku nekog faktora u situacijama kad nije moguće ili etički namjerno manipulirati zavisnim i nezavisnim varijablama (Cohen i ostali, 2013). Postoje dva pristupa takvom istraživanju. U

prvom slučaju promatra se nezavisna varijabla te provjerava kako utječe na zavisnu, a kod drugog pristupa promatra se zavisna varijabla kako bi se utvrdilo kako se skupine razlikuju po nezavisnoj, odnosno što je uzrok rezultatima. U takvim su se situacijama nezavisne varijable već dogodile i istraživač retrospektivno ispituje učinke događaja kako bi uspostavio kauzalne veze među varijablama. Svako bi se istraživanje trebalo uklopiti u redovnu nastavu kao dio svakodnevne realizacije sadržaja. U skladu s tim, nastava na kolegiju odvijat će se prema predviđenom planu i programu. Pristup pripada naturalističkoj paradigmi obzirom da se odvija u prirodnom okruženju.

4.4.1 Opis istraživanja

Istraživanje je provedeno u prirodnom okruženju na temelju podataka prikupljenih tijekom tri godine izrade projekata otvorenog tipa na kolegiju Objektno orijentirano programiranje (OOP) na Prirodoslovno-matematičkom fakultetu u Splitu. U nastavnom planu i programu planiran je rad s objektno orijentiranim programskim jezikom C# u okruženju Microsoft Visual Studio (MVS) verzije koja je bila aktualna u odgovarajućoj akademskoj godini. Analizirana je izrada projekata otvorenog tipa koje su studenti izrađivali u sklopu redovne nastave u alatu OTTER prema broju koncepata i složenosti projekata.

Koncepti objektno orijentirane paradigme dolaze više do izražaja kad se rade složeniji projekti koji zahtijevaju više vremena od zadataka koji se inače mogu riješiti tijekom trajanja tjednih vježbi. Obzirom da je vrijeme ograničeno trajanjem kolegija i u situacijama kad studenti rade projekte sami, treba unaprijed pripremiti odgovarajući početni okvir u kojem će raditi (J. Kay i ostali, 2000). Izrada projekata omogućuje studentima povezivanje više znanja i vještina te eventualni prijenos znanja iz drugih kolegija i područja.

Ovim istraživanjem obuhvaćeno je projekata te rezultata praćenja prikupljenih tijekom više godina. Uzorak je prigodni na temelju upisa studenata u odgovarajućoj akademskoj godini. Kontrolne i eksperimentalne skupine definirane su prema akademskim godinama (nezavisni uzorci), a kontrolna skupina predstavlja studente kojima je održana nastava bez didaktičkog skrivanja, a eksperimentalna s metodom didaktičkog skrivanja i uz pomoć alata OTTER.

Premda su projekti otvorenog tipa, rad na projektu je dio ocjene iz kolegija tako da studenti moraju znati što se od njih očekuje. Cilj svakog projekta je na osnovi odabrane teme demonstrirati koncepte objektno orijentiranog programiranja tako da se projekti ocjenjuju prema definiranom kriteriju, odnosno koristi se *vrednovanje temeljeno na kriteriju* (eng. Criterion-based assessment). Kriteriji se odnose na objektno orijentirane koncepte dok je tema

slobodna kako bi studenti pokazali inicijativu i bili motivirani raditi ono što ih zanima. Na vježbama koje su prethodile početku izrade projekta studenti rješavaju jednostavne vježbe fokusirane na određene koncepte koji će im kasnije biti potrebni pri izradi projekta. Pored izrade projekta u konačnu ocjenu iz kolegija ulazi i usmena prezentacija projekta te rezultati kolokvija. Kontrolna skupina izrađivala je projekte u MVS okruženju u obliku grafičkog sučelja s obrascima dok studenti eksperimentalne skupine koriste unaprijed pripremljeni alat. Prilikom vrednovanja projekata u svrhu ovog istraživanja koristit će se analiza po konceptima koje su studenti koristili.

Za analizu kvantitativnih podataka korišten je odgovarajući programski alat za obradu projekata. Istraživanje se provodi u suradnji s više nastavnika na kolegiju, te je istraživač dio stvarnog okruženja.

Istraživanje je provedeno s ciljem boljeg razumijevanja upotrebe rješavanja problema u poučavanju preddiplomskih studenata objektno orijentiranom programiranju u formalnom okruženju. Prvi korak je pojasniti sam uzorak studenata i njihovo očekivano iskustvo, obzirom na nastavu koju su slušali do upisa kolegija OOP.

Studenti preddiplomskog studija na našim uvodnim kolegijima iz programiranja su većinom početnici bez prethodnog iskustva u programiranju. Prvi susret s programiranjem tijekom studija, a obično možemo reći i prvi susret s programiranjem uopće, je tijekom prvog semestra i kolegija Programiranje I (P1) gdje im se uvodi proceduralna paradigma i osnovni koncepti iz programiranja. Sljedeći korak ili kolegij je u drugom semestru: Programiranje II (P2) gdje im se uvodi objektno orijentirani programski jezik C#. Obzirom da su tijekom kolegija P1 učili o osnovnim konceptima na proceduralan način kroz jednostavniji jezik primjeren početnicima (npr. Python), fokus kolegija P2 je usvajanje složenije sintakse programskog jezika C# te blagi prijelaz na objektno orijentiranu paradigmu ponovnim uvođenjem osnovnih koncepata kao što su primjerice varijable, slijed, grananje, ponavljanje, strukture podataka i sl. Studenti su već prošli spomenute koncepte tijekom kolegija P1, ali im se spiralno pored tih koncepata u novoj sintaksi dodaju i novi koncepti. Primjerice, Python je jezik u kojem nije potrebno deklarirati varijable niti pretjerano paziti na tipove podataka dok u C#-u moramo razlikovati čak i više vrsta cijelih brojeva. Početni predmeti iz programiranja su često zasnovani na znanju, odnosno fokusirani na sadržaj ili deklarativno znanje o programiranju (Robins i ostali, 2003). To nije toliko neobično jer je prilično teško obuhvatiti cjelokupno znanje i vještine odjednom. Na primjer, tijekom kolegija P2, studenti uče o klasama, kako se pišu i od čega se sastoje, ali imaju poteškoća kod primjene ili prepoznavanja kad i kako bi klase trebali upotrijebiti u svojim

programima. Nastavnici moraju na neki način provjeriti znaju li studenti napisati klasu, ali to zna biti teško ispitati ako sami studenti nisu sigurni kad će klasu upotrijebiti. Kao posljedica toga može se dogoditi da studenti napišu rješenje koje daje željeni rezultat, ali bez upotrebe klase, svođenjem problema na ono što im je više poznato čak i kada je potrebno zbog toga napisati više naredbi. U skladu s navedenim, nastavnici ne mogu na takvom rješenju ispitati deklarativno znanje – zna li student napisati klasu ako već nije siguran gdje i kako je primijeniti, tako da ponekad zadaci postaju previše specifični (npr. Napisati klasu Pas).

U drugoj godini studija, studenti upisuju kolegij OOP, a cilj tog kolegija je usvojiti objektno orijentirane koncepte kao što su klase, nasljeđivanje, učahurivanje, polimorfizam i apstrakcija.

Ispitanici su bili studenti upisani na kolegij OOP. Nastavnici koji inače izvode nastavu su proveli istraživanje.

Preddiplomski kolegij OOP je podijeljen na dva dijela, predavanja i vježbe, a izvodi se tijekom 15 tjedana (dva sata predavanja i dva sata vježbi tjedno). Tijekom predavanja pored „klasičnog“ izlaganja sadržaja, studentima su predstavljeni primjeri upotrebe okvira. Svaki tjedan se također naglašavao određeni dio okvira kao nadopuna vježbama i trenutnim konceptima objektno orijentiranog programiranja koji su u fokusu.

Vježbe se mogu podijeliti na dvije logičke cjeline s kolokvijem koji se održava između njih. Prvi dio se sastoji od 10 dijelova, a svaka od njih je povezana sa specifičnim konceptom OOP-a.

1. ***Uvod u okvir*** (klase, object, metoda). Studentima se predstavlja okvir uz kratak opis arhitekture i osnovnih funkcionalnosti na primjeru jednostavne igre. Prvi zadatak je pomicanje zadanog lika po pozornici. Prvi cilj vježbe je demonstrirati razliku između koncepata objekta i klase prikazom dva različita lika (pas i mačka) na pozornici. Drugi cilj je pokazati da metode koje su pisane jednom (unutar klase) možemo i moramo pozivati za svakog lika posebno kako bi se pokazao učinak tih metoda na svaki lik.
2. ***Napredne metode*** (parametri, povratni tip, preopterećenje). Druga faza je usmjerena na napredne koncepte metoda – slanje ulaznih parametara, razumijevanje domene podataka te vraćanja rezultata. Studenti opet dobiju unaprijed pripremljeni scenarij (započetu igru) te moraju promijeniti postojeće metode kako bi izvršili postavljene zadatke. Ovdje se koristi mogućnost uvođenja preopterećenih metoda, a to pripada osnovnim OOP konceptima (polimorfizam). Studenti moraju preopteretiti postojeće

metode kako bi se mogle primijeniti u zamišljenim scenarijima te testirati implementacije tih metoda koje su napravili.

3. ***Klase i objekti*** (konstruktor, svojstva, modifikatori pristupa, static). Nakon što su se studenti upoznali s okruženjem, idući zadatak je napisati vlastitu klasu s osnovnim dijelovima: poljima, svojstvima i metodama (uključujući i konstruktor). Također im se uvodi koncept učahurivanja kroz svojstva klase i odgovarajućim get/set metodama. U ovoj fazi se koriste samo „private“ i „public“ modifikatori za pisanje svojstava. Studenti moraju definirati vlastitu klasu unutar okvira prema danim smjernicama. Klasa mora sadržavati najmanje jednu vrijednost i svojstvo, jednu metodu i preopterećeni konstruktor. Dodatno, studentima se upoznaju s konceptom statičke klase kroz primjer takve klase integrirane u sam okvir. Klasa se koristi za spremanje različitih vrijednosti koje se odnose na cijelu igru (širina likova, zadana brzina i sl.), a potrebno ih je definirati samo jednom. Obzirom da se te vrijednosti postavljaju samo jednom, a koriste mogu ih koristiti svi likovi, logična je upotreba statičke klase.
4. ***Nasljeđivanje*** (inheritance, new, override, protected). Sljedeći korak u vježbama je uvođenje koncepta nasljeđivanja. Studenti uče kako se definira klasa koja nasljeđuje od osnovne klase (eng. base class). Pored toga, uvodi im se novi modifikator pristupa „protected“ te redefiniranje elemenata osnovne klase pomoću premošćivanja (eng. override) razlikujući upotrebu ključnih riječi: new i override. Studenti dokazuju usvajanje spomenutih koncepata dovršavanjem pripremljenog zadatka u kojem moraju napisati osnovnu klasu s jednom metodom te definiranjem dviju novih klasa. Nove klase nasljeđuju od osnovne klase, ali svaka od njih ima različitu implementaciju metode iz osnovne klase. Klase u ovom slučaju predstavljaju različite likove u igri koji dijele slične karakteristike, odnosno svojstva (sadržana u osnovnoj klasi) te slično ponašanje predstavljeno metodom koja ima isti naziv u obje klase, ali drugačiju implementaciju.
5. ***Višestruko nasljeđivanje*** (base, abstract). Nastavljajući na prethodnu fazu, fokusiramo se na dodatne koncepte nasljeđivanja koji nisu mogli biti obrađeni u prethodnom dijelu. Ti koncepti uključuju apstraktne klase, višestruko nasljeđivanje, definiranje get/set te pristupanje elementima osnovne klase. Studenti mijenjaju početni okvir tako da glavna klasa okvira „Sprite“ postaje apstraktna te kao takva ne može imati instance. U skladu s navedenim, koristit će se kao predložak za buduće klase. Izrađuje se igra s dva svemirska broda (prema popularnim filmovima) te jedan od njih predstavlja lika kojeg kontrolira igrač, a drugi predstavlja „neprijateljski“ brod. Svaki brod pripada posebnoj klasi, ali dijele zajednička svojstva i metode iz osnovne klase.

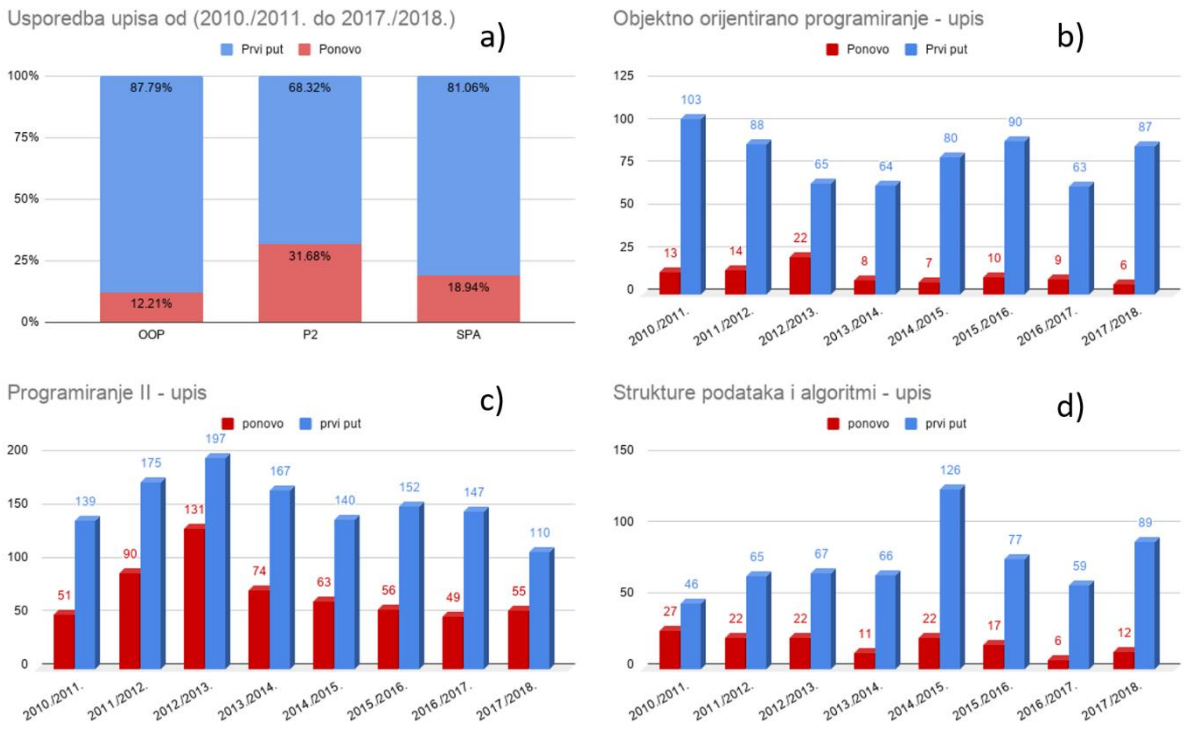
6. ***Događaji*** (event, eventHandler). Koncept događaja je važan za OOP jer je vezan za interakciju među objektima. U ovom dijelu se studenti detaljnije upoznaju s konceptom događaja, kako ih izazvati (eng. invoke) te kako upravljati (eng. handle) odnosno kako reagirati na događaje. Studenti uređuju okvir definiranjem načina detekcije događaja izazvanih mišem i tipkovnicom te odabrane događaje koriste dalje u scenariju igre. Metode i svojstva koji su povezani s otkrivanjem i upravljanjem događajima su učahureni unutar posebne klase koja je inače standardni dio okvira, ali u svrhu vježbe studente se vodi da sami implementiraju tu klasu kako bi bolje shvatili na koji način funkcionira. Zadatak je upravljati likovima na zaslonu, ali pomoću miša i tipkovnice umjesto da se likovi kreću prema napisanim naredbama u metodi za ponašanje lika.
7. ***Delegati*** (delegate). Koncept delegata je blisko vezan uz događaje obzirom da se moraju koristiti prilikom slanja reference na metodu u sklopu upravljanja događajem. To se odnosi na događaje koje pišu sami studenti (npr. kraj igre, kolizija i sl.). Zadatak je napisati događaj koji odgovara koliziji dva objekta u igri te implementirati metodu unutar klase koja odgovara na taj događaj. Nakon toga, studenti implementiraju program, odnosno igru u kojoj će se dogoditi taj događaj (kolizija likova) te izazvati reakcija na taj događaj.
8. ***Rad s iznimkama*** (exceptions, try, catch, finally). Pogreške koje se događaju tijekom izvođenja programa su normalna stvar u pocesu programiranja, a način kako predvidjeti i planirati reakciju na takve pogreške je važan dio učenja. Iz tog razloga je jedna vježba posvećena posebno radu s iznimkama (eng. exceptions). Studenti se upoznaju s pojmom iznimke u programu, različitim vrstama te njihovom povezanosti. Nakon toga uče definirati vlastite iznimke te kako upravljati tim iznimkama unutar projekta. Zadatak je dodati u postojeću klasu posebnu iznimku koja se događa kad korisnik unese neispravnu vrijednost parametra koji se odnosi na sliku lika (prazan tekst ili neispravna putanja).
9. ***Primjer jednostavnog projekta***. Tijekom posljednjeg dijela vježbi demonstrira se jednostavni primjer oblikovane igre. Studentima se objašnjava scenarij s klasama u obliku dijagrama klasa. Za svaku klasu je dan popis metoda i svojstava, a zadatak studenata je dovršiti dani primjer prateći zadane smjernice.
10. ***Oblikovanje projekta***. Nakon upoznavanja s osnovnim konceptima OOP, studenti se moraju upoznati s načinima korištenja tih koncepata u odgovarajućim slučajevima. Iz tog razloga, moraju se upoznati i s osnovama oblikovanja projekta i planiranja. U ovoj vježbi studentima se prikazuje nekoliko jednostavnih slučajeva uz primjere kako je svaki od njih planiran i oblikovan. Zadatak studenata je odlučiti o temi njihovog

konačnog projekta te osnovnim elementima oblikovanja tog projekta kao što su npr. broj klasa i njihova struktura, metode, svojstva, akcije korisnika/igrača, logika igre i sl.

Nakon dovršetka prvog dijela vježbi, studenti polažu i prvi kolokvij koji se u većini temelji na sintaksi jezika, odnosno načinu implementacija usvojenih koncepata. Premda su se upoznali s osnovnim OOP konceptima, potrebno je vrednovati njihovo poznavanje sintakse obzirom da to može biti ozbiljna prepreka pri učenju bilo koje programske paradigme. Nadalje, testiranjem OOP koncepata u ispitu može biti problematično jer ne postoji samo jedan ispravan pristup oblikovanju i implementaciji klasa. Iz tog razloga se vrednovanje OOP koncepata provodi tijekom usmenog dijela ispita na kraju semestra kad studenti predstavljaju svoje projekte te objašnjavaju svoje implementacije i načine oblikovanja te razloge zašto su nešto napravili na određeni način. Prema tome, posljednja četiri tjedna su rezervirana za izradu projekata. Studenti imaju vrijeme za izradu, ali isto tako i rok do kojeg moraju predati. Tema projekta je slobodna, ali u zadnjim generacijama je jedino ograničenje upotreba zadanog okvira. Premda će studenti većinu svojih projekata raditi van učionica, dolasci na posljednje vježbe su obavezni kako bi mogli dobiti povratne informacija, ali i pomoć nastavnika tijekom rada.

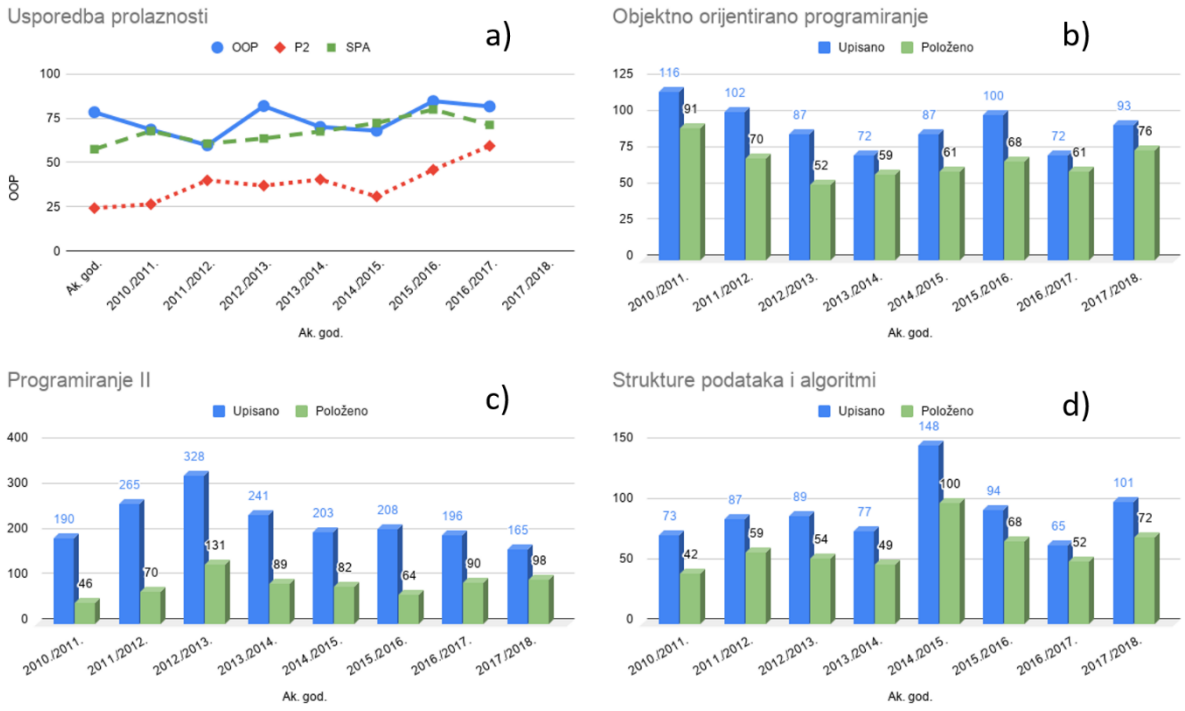
4.4.2 Podaci o uzorku

Analizirani su podaci posljednjih osam generacija na predmetima Programiranje II (P2), Strukture podataka i algoritmi (SPA) i Objektno orijentirano programiranje (OOP). Ukupno su obuhvaćeni podaci o 1796 upisa na P2, 734 upisa na SPA te 729 upisa na OOP. Na slici ispod (Slika 4.7) prikazan je odnos broja upisanih studenata prvi put i onih s ponovljenim upisom za ta tri predmeta. Vidljivo je da je broj ponavljača na OOP smanjen, ali treba napomenuti da dio studenata koji nije uspio položiti P2 nije nastavio studij niti upisao SPA i OOP.



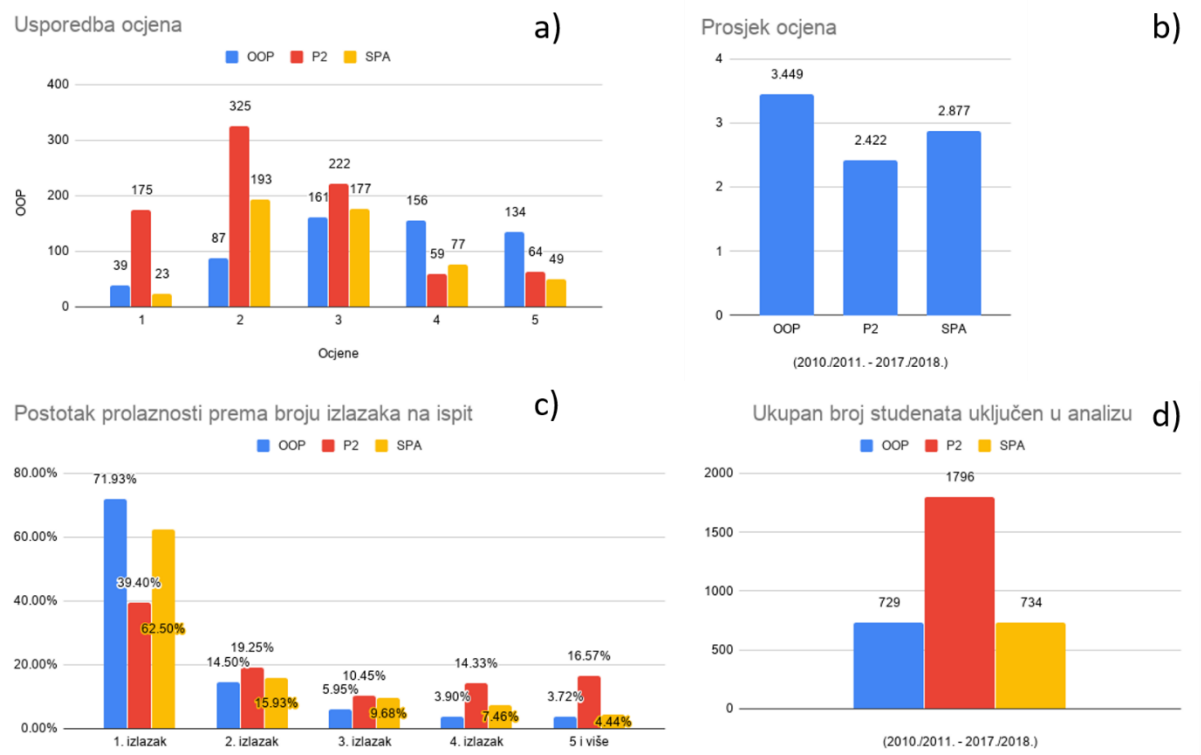
Slika 4.7. Analiza upisa prvi put i ponovljenog upisa

Druga analiza predstavlja odnos broja upisanih studenata i onih koji su položili ispit. Tu se može vidjeti nešto bolja prolaznost na OOP. Ovdje također moramo napomenuti da postotak prolaznosti ne mora značiti uspješnije poučavanje jer je ocjenjivanje relativno.



Slika 4.8. Usporedba prolaznosti

Nadalje, potrebno je napomenuti da su na predmete OOP i SPA upisani studenti koji su položili P1 i/ili P2, tako da je ostao ipak manji broj studenata koji su ipak pokazali određeno znanje iz početnog programiranja. Ako promatramo ocjene, prosjek malo ide u korist OOP, ali to nije mjerilo boljeg uspjeha. Zanimljivo je primijetiti kako većina studenata koja polaže ispite zapravo položi na prvom roku (Slika 4.9).



Slika 4.9. Analiza ocjena i prolaznosti po rokovima

4.4.3 Prikupljanje podataka o projektima

Prikupljeni su podaci, odnosno konačni projekti studenata, tijekom posljednjih pet godina, počevši s akademskom godinom 2013/14. Ukupno je prikupljeno 289 projekata, a tijekom tog vremena je na kolegij upisano 425 studenata od čega je 349 položilo kolegij. Neki studenti su radili spojene projekte (kombinirano s drugim kolegijima kao što je npr. Uvod u umjetnu inteligenciju) te takvi projekti nisu mogli biti usporedivi s ostalima, a neki studenti su predali nepotpune projekte ili nisu predali uopće. Analizirani su samo projekti koji su dovršeni te je zbog svega navedenog konačan broj 289.

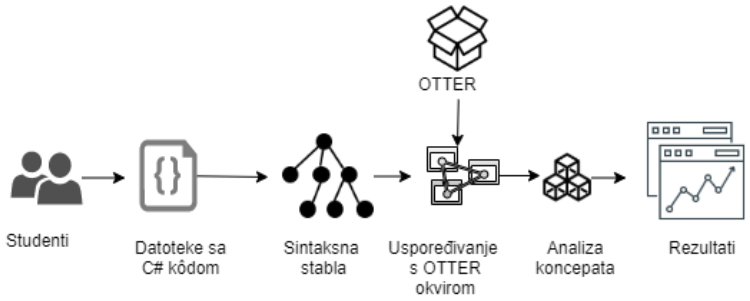
Tijekom prve dvije godine, upotreba okvira je bila proizvoljna, a vježbe su se temeljile na jednostavnim aplikacijama s grafičkim korisničkim sučeljem gdje su studenti koristili OOP koncepte. Na kraju semestra su studenti mogli birati da li žele izrađivati svoj projekt upotrebom okvira OTTER ili „klasične“ aplikacije s grafičkim korisničkim sučeljem.

Premda je većina (oko 90%) studenata odabrala izradu GUI aplikacije neki su se odlučili i za izradu jednostavnog projekta u obliku igre upotrebom OTTER okvira. Preliminarnom analizom projekata razvijenih upotrebom okvira uočeno je kako sadrže više OOP koncepata u odnosu na GUI projekte. Međutim nije bilo moguće donijeti precizne zaključke na malom uzorku studenata te smo odlučili preoblikovati vježbe (opisano u prethodnom dijelu) u kojima je upotreba okvira za izradu projekta postala obvezna. Rezultat te promjene je da su svi projekti prikupljeni u posljednje tri godine (203 ukupno) napravljeni u OTTER okviru. Na taj način smo dobili mnogo veći uzorak koji se mogao podijeliti na kontrolnu i eksperimentalnu skupinu, pri čemu je eksperimentalna skupina koristila OTTER, a kontrolna ne. Prema tome, moglo se ispitati da li postoji utjecaj okvira na usvajanje koncepata OOP tijekom upotrebe okvira.

Sam okvir je tijekom vremena prošao manje izmjene, ali su te izmjene bile manje (ispravljanje nekih nedostataka i poboljšanje performansi tako da nisu utjecale na proces učenja. Proces analize projekata opisan je u idućem dijelu.

4.4.3.1 Analiza projekata

Na slici prikazan je model analize studentskih projekata.



Slika 4.10. Model analize podataka

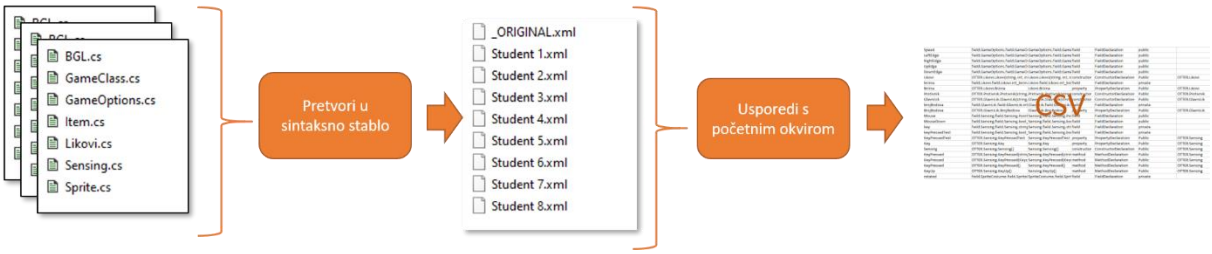
Opisat ćemo faze izvlačenja podataka:

1. ***Izdvajanje datoteka***. Studentski projekti su komprimirana C# rješenja (eng. solutions) koja su studenti predali na sustav e-učenja. Svako rješenje sastoji se od C# kôda (.cs datoteke) koje su potrebne za analizu, te ostalih datoteka koje će se ignorirati. Primjer ignoriranih datoteka su „.Designer.cs“ datoteke, multimedijske datoteke te ostale konfiguracijske datoteke ili datoteke s podacima.
2. ***Faza izdvajanja koncepata***. Prilagođeni analizator kôda izvlači koncepte prve razine kao što su strukture (eng. struct) i klase parsiranjem svake „cs“ datoteke i stvaranjem sintaksnog stabla. Iz svake klase ili strukture analizator izdvaja podčvorove kao što su:

svojstva (eng. properties), polja (eng. fields), konstruktori i metode. Neki od čvorova su pojednostavljeni te se smatraju terminalnima obzirom da njihovi potomci ne sadrže više bitnih informacija potrebnih za ovu analizu (npr. polja), dok ostali imaju više potomaka (eng. child nodes), npr. svojstva koja sadrže get i set pristupne čvorove (eng. accessor nodes). Nakon ove faze stvorene su XML datoteke koje predstavljaju strukturu svih studentskih projekata te također i originalnog OTTER okvira. Tijela metoda sadrže kôd, ali ne i koncepte koji su promatrani u ovoj analizi tako da su tijekom ove faze samo spremljeni za daljnju tekstualnu analizu. XML datoteka svakog projekta sastoji se od reprezentacije stabla u kojem su sve izvučene informacije.

3. ***Faza usporedbe***. Tijekom ove faze, skup studentskih projekata iz svake generacije uspoređuje se s originalnim okvirom kako bi se utvrdile sličnosti i razlike. Svaki čvor iz prethodne faze označen je na jedan od tri načina: novi, promijenjen, isti, korišten. Oznaka „novi“ označava čvorove koji ne postoje u originalnom okviru, a čim se otkrije novi čvor, automatski se njegovi preci (roditelj i svi njegovi preci) označavaju kao „promijenjeni“. Čvorovi su označeni kao „promijenjeni“ i u slučaju da se njihovo tijelo (eng. body), odnosno blok kôda koji im pripada razlikuje od originalnog (usporedbom teksta) ili ako se djeca promatranog projekta razlikuju od originalnog. U skladu s navedenim, oznakom „isti“ označavaju se čvorovi koji nemaju nikakvih promjena u odnosu na originalni okvir. Ukoliko projekt studenta ima „istih“ čvorova znači da je student pri izradi svog projekta koristio originalni okvir. Posljednji tip čvora je „korišten“ koji označava da je student primjenjivao neki od postojećih čvorova iz originalnog okvira u svom projektu. Studenti nisu morali primijeniti sve elemente koji se nalaze u okviru.
4. ***Faza analize koncepata***. Tijekom ove faze, analiziraju se svi rezultati prethodne faze te se identificiraju i prebrojavaju objektno-orijentirani koncepti. Važno je primijetiti da se samo čvorovi označeni oznakama „novi“, „promijenjen“ ili „korišten“ mogu analizirati tako da složenost početnog okvira i veliki broj koncepata koji je već uključen u okvir neće utjecati na rezultate.

Podaci za svaku generaciju studenata su izdvojeni u zasebnu CSV datoteku. Prosječan broj zapisa i identificiranih elemenata za analizu je bio oko 4000 po datoteci. Sve CSV datoteke su se zatim dodatno obradile kako bi se za svakog studenta dobio točan broj koncepata.



Slika 4.11. Pretvaranje u sintaksno stablo

Na slici 37 je prikazan samo dio podataka za zadnju generaciju studenata. Svaki redak predstavlja jedan projekt/studenta, a stupci su varijable koje su promatrane. Uzorak na slici nije reprezentativan, ali se ipak može primijetiti kako je prva promatrana generacija studenata više koristila premošćivanje (eng. override) metode *ToString()* kao primjer, dok studenti u novijoj generaciji to rijetko koriste već moraju zapravo pronaći smislenu primjenu u svojem jedinstvenom projektu.

2013/14

GUI	abstract	protected	staticClass	inherit	maxDepth	virtual	overrideToString
2	0	0	0	0	0	0	1
3	0	0	0	1	0	0	1
3	1	0	0	1	1	0	0
2	0	0	0	1	0	0	1
5	1	1	0	2	1	1	2
2	2	0	0	4	1	0	4
6	2	0	0	1	1	0	1
1	0	0	0	4	2	1	0
4	2	0	0	7	1	0	1
4	2	0	0	5	1	0	0
1	1	0	2	10	1	0	0
6	0	0	0	0	0	0	0
1	3	2	0	8	2	0	0

2017/18

GUI	abstract	protected	staticClass	inherit	maxDepth	virtual	overrideToString
1	1	0	0	6	2	0	0
1	0	0	1	5	2	0	0
1	0	0	0	4	2	0	0
1	0	0	1	4	2	0	0
1	0	0	0	3	2	0	0
1	1	0	0	6	2	0	0
1	2	0	1	3	2	0	0
1	2	1	1	6	3	0	0
1	1	1	0	6	2	0	0
2	1	0	0	6	2	1	0

Slika 4.12. Isječak iz datoteka s podacima za statističku analizu

Nakon prebrojavanja različitih koncepata iz projekata, provedbom Kolmogorov-Smirnov testa ispitano je da li podaci zadovoljavaju uvjete normalne distribucije. Rezultati pokazuju da svi podaci nemaju normalnu distribuciju što nas vodi do zaključka da treba koristiti neparametrijske statističke testove kao što su Mann-Whitney kako bi utvrdili postoje li razlike između dviju nezavisnih skupina ili Kruskal-Wallis test za provjeru postoje li razlike između tri ili više skupina. Identificirali smo ukupno 18 varijabli, zasnovanih na konceptima koji su izdvojeni iz projekata. Problem s tim rezultatima je velik broj varijabli koje potencijalno mogu sadržavati redundantne informacije. U skladu s tim, provedena je faktorska analiza (Cohen i ostali, 2013) kako bi statistički reducirali broj varijabli te izbjegli redundanciju u slučajevima kad su različite varijable povezane s istim konceptima.

4.4.4 Statistička obrada podataka

Faktorska analiza koristi se za identificiranje klastera ključnih varijabli, ali se također koristi za identificiranje redundantnih elemenata (Cohen i ostali, 2013). Važno je provesti analizu na

uzorku primjerene veličine jer analiza provedena na malim uzorcima nije pouzdana. To je metoda koja se koristi za grupiranje varijabli koje imaju nešto zajedničko. Grupirane varijable nazivamo faktorima. Eksplorativna faktorska analiza (EFA) se koristi za određivanje faktora kako bi se otkrile grupe varijable koje nisu bile poznate, a konfirmatorna faktorska analiza (CFA) se koristi za provjeru postojećeg modela, tako da je ovdje primjerena EFA jer želimo odrediti slične varijable te pri tome i reducirati broj tih varijabli. Na primjer, Hoegh & Moskal (2009) su koristili faktorsku analizu kako bi utvrdili da li pitanja u upitnicima odgovaraju konstruktima koje su predvidjeli, a na taj način mogu utvrditi konstruktnu valjanost (Kunkle & Allen, 2016).

Varijable koje se koriste u faktorskoj analizi bi trebale biti linearno povezane jer se u protivnom može dogoditi da broj faktora nakon analize bude skoro isti kao što je i početni broj varijabli. Preporučena veličina uzorka je više od 150 s najmanje 5-10 slučajeva za svaku promatranu varijablu, a vrijednosti Kaiser-Meyer-Olkin mjere primjerenosti uzorka (eng. Measure of Sampling Adequacy, KMO) bi trebale biti 0.6 ili više. Test „Bartlett Test of Sphericity“ bi trebao biti razine značajnosti 0.05.

Prva faza analize je primjena odabrane metode za računanje početnog *opterećenja faktora* (eng. factor loading) kao što je na primjer PCA, a nakon PCA faktori će i dalje ostati nekorelirani. Opterećenja faktora prikazuju snagu povezanosti varijable s faktorom. Može se dogoditi da neke varijable imaju visoko opterećenje za više faktora ili nemaju opterećenje za ni jednu (Hoegh & Moskal, 2009).

Druga faza analize je rotacija faktora, a odabrana je „varimax“ rotacija kako bi faktori ostali nekorelirani. Cijeli postupak analize se obično radi uz pomoć odgovarajućeg statističkog softvera kao što je na primjer SPSS. Rotacija tipa *varimax* se najčešće koristi kod istraživanja u obrazovanju (Wetzel, 2011).

Vrijednost KMO za analizirane podatke je 0.804, a značajnost za Bartlettov test $p<0.05$. Prema dobivenim rezultatima prva četiri faktora imaju najviše utjecaja na varijancu. *Ekstrakcijske sume kvadrata opterećenja* (eng. extraction sums of squared loadings) sadrže informaciju o postotku varijance. Postotak varijance nam kaže koliko varijance je objašnjeno sa svakim identificiranim faktorom. Prvi faktor ima najveći postotak, a svi faktori su nezavisni što znači da količina varijance u jednom faktoru nije povezana s nekim drugim faktorom. Ova prva analiza nam pruža grube rezultate.

Možemo zamisliti da su podaci o faktorima iscrtani u dvodimenzionalnom prostoru, no tu se može dogoditi da su neki faktori i varijable previše blizu jedni drugima pa prema tome diskriminacija faktora nije baš jasna. Ako se doda još jedna dimenzija *dubine rotacijom* (eng. depth by rotating) onda će one varijable koje su blizu postati još bliže dok će one udaljene biti više odmaknute kako bi lakše uočili razlike.

Tablica 4.3. Rezultati analize

Komponenta	Početni vektori			Ekstrakcijske sume kvadrata opterećenja			Rotacijske sume kvadrata opterećenja		
	Ukupno	% varijance	Kumulativni %	Ukupno	% varijance	Kumulativni %	Ukupno	% varijance	Kumulativni %
1	7.105	39.475	39.475	7.105	39.475	39.475	6.818	37.876	37.876
2	2.563	14.238	53.712	2.563	14.238	53.712	2.38	13.222	51.098
3	1.377	7.653	61.365	1.377	7.653	61.365	1.589	8.828	59.926
4	1.302	7.231	68.596	1.302	7.231	68.596	1.561	8.67	68.596
5	0.999	5.551	74.147						
6	0.869	4.825	78.972						
7	0.718	3.99	82.962						
8	0.643	3.575	86.537						
9	0.536	2.978	89.515						
10	0.429	2.382	91.897						
11	0.397	2.204	94.101						
12	0.355	1.973	96.074						
13	0.219	1.217	97.291						
14	0.209	1.159	98.45						
15	0.155	0.861	99.311						
16	0.084	0.465	99.776						
17	0.022	0.123	99.899						
18	0.018	0.101	100						

Rotacijske sume kvadrata opterećenja (eng. rotation sums of squared loadings) prikazuju rezultate rotacije, a rotirana matrica komponenti (eng. component matrix) određuje kojoj grupi pripada svaka varijabla. U sljedećoj tablici su izvučena samo prva četiri faktora (Tablica 4.4). Preporučuje se korištenje rotacijske sume.

Tablica 4.4. Rotacijske sume kvadrata opterećenja

Faktor	Ukupno	% varijance	Ukupni %
1	6.818	37.876	37.876
2	2.380	13.222	51.098
3	1.589	8.828	59.926
4	1.561	8.670	68.596

Postoji 11 varijabli u grupi koja pripada prvom faktoru: broj premošćenih (eng. overriden) metoda (osim *ToString()* metode), virtualnih elemenata, klasa koje nisu klasa *Form*, nasljeđivanje, apstraktnih elemenata, poziva konstruktora osnovne klase, jednostavnih *pristupnih* (eng. accessor) čvorova, metoda (osim metoda za upravljanje standardnim događajima), zaštićenih (eng. protected) elemenata, nepraznih konstruktora i složenih pristupnih čvorova (Tablica 4.5). Drugi faktor sadrži broj ostalih metoda za upravljanje događajima, broj Form klasa te broj premošćivanja metode *ToString()*. Pravilo "rule of thumb" prema Cohen-u je da bi svaki faktor trebao imati bar tri varijable. Najniža razina „rezanja" (eng."cut-off") za odabir varijabli je u ovom slučaju 0.4. Postavljanje više razine daje veću „snagu" faktorima, ali također i reducira njihov broj. Postavljanje točke rezanja smanjuje broj manje bitnih varijabli.

Tablica 4.5. Matrica komponenti

	Faktori			
	1	**2**	**3**	**4**
overrideOtherMethods	0.888			
virtual	0.882			
notGuiClass	0.872			
inherits	0.859			
abstract	0.834			
baseConstructorCall	0.783			0.432
simpleGetSet	0.703			
otherMethods	0.698		0.550	
protected	0.685			
nonEmptyConstructor	0.682			
complexGetSet	0.678			
eventHandlerMethods		0.827		
GUI		0.826		
overrideToString		0.626		
overload			0.868	
staticClass			0.516	
maxDepthOfInheritance				0.768
staticNodesInNonStaticClass				

Statistička analiza i identificiranje faktora izdvaja faktore, ali na istraživaču je postavljanje naziva faktora (ako je potrebno):

- Faktor 1: Objektno orijentirani koncepti,
- Faktor 2: Koncepti iz grafičkog sučelja.

Veći broj varijabli koje pripadaju prvom faktoru je i očekivan, obzirom da su te varijable identificirane kao objektno-orijentirani koncepti prikupljeni tijekom analiza studentskih

projekata te bi zapravo i trebale biti korelirane. S druge strane, drugi faktor sadrži upravo one varijable koje su identificirane kao manje bitne u kontekstu usvajanja objektno-orijentiranih koncepata. Također, prva dva faktora sadrže i većinu varijabli i varijance.

Provedba PCA dovela je do identifikacije dva faktora koji predstavljaju dva skupa koncepata primijenjenih u projektima. Obzirom da se PCA i EFA ne moraju poklapati, provedena je također i EFA uz iste postavke, ali upotrebom besplatnog alata JASP (JASP Team, 2019). Alat je odabran iz razloga što je besplatan i jednostavan za korištenje. Slika 4.13 prikazuje sliku zaslona računala u JASP-u. Posebno su istaknuta prva dva faktora koji se poklapaju s prethodnom analizom. Varijabla *Inherits* pojavljuje se u dva faktora, ali je puno veći utjecaj na prvi faktor.

Exploratory Factor Analysis

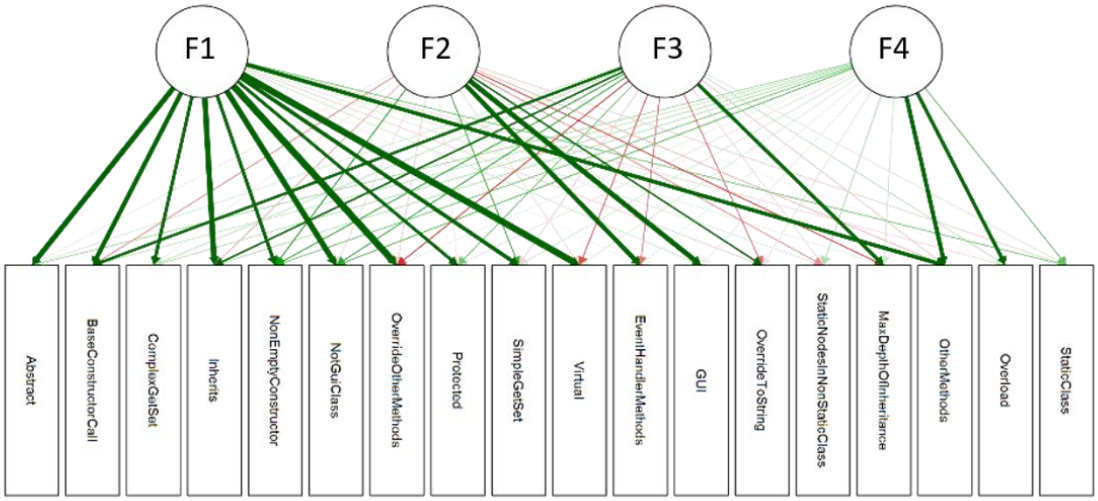
Factor Loadings ▼

	Factor 1	Factor 2	Factor 3	Factor 4
Abstract	0.802	.	.	.
BaseConstructorCall	0.731	.	0.570	.
ComplexGetSet	0.619	.	.	.
EventHandlerMethods	.	0.740	.	.
GUI	.	0.781	.	.
Inherits	0.818	.	0.472	.
MaxDepthOfInheritance	.	.	0.646	.
NonEmptyConstructor	0.628	.	.	.
NotGuiClass	0.836	.	.	.
OtherMethods	0.676	.	.	0.699
Overload	.	.	.	0.643
OverrideOtherMethods	0.945	.	.	.
OverrideToString	.	0.459	.	.
Protected	0.606	.	.	.
SimpleGetSet	0.672	.	.	.
StaticClass	.	.	.	.
StaticNodesInNonStaticClass	.	.	.	.
Virtual	0.922	.	.	.

Note. Applied rotation method is varimax.

Slika 4.13. Rezultati EFA pomoću JASP (JASP Team, 2019)

Povezanost faktora s varijablama možemo prikazati i grafički (Slika 4.14).



Slika 4.14. Grafički prikaz povezanosti faktora s varijablama (JASP Team, 2019)

S reduciranim brojem varijabli, Faktor 1 i Faktor 2, postavljamo nul-hipoteze:

H1: Ne postoji statistički značajna razlika u Faktoru 1 među generacijama.

H2: Ne postoji statistički značajna razlika u Faktoru 2 među generacijama.

Prema Kolmogorov-Smirnov testu, distribucija podataka nije normalna pa se u skladu s tim koristio Kruskal-Wallis test (KWT). KWT pokazuje kako postoji statistički značajna razlika u Faktoru 1 među 5 generacija ($p<0.05$). Usporedbom srednjih vrijednosti (eng. mean ranks) za svaku generaciju potvrđuje zapažanja nastavnika kako se u posljednjim generacijama koristi manje GUI elemenata (npr. Form, Button, …) te više objektno-orijentiranih koncepata koje su trebali usvojiti. Nakon uvođenja obavezne upotrebe isključivo OTTER okvira koji je u međuvremenu i nadograđen, može se vidjeti kako u generaciji 2015/16 drastično opada srednja vrijednost za GUI elemente. Međutim, posljednje dvije generacije su imale veću potrebu dodavati GUI elemente pored upotrebe OTTER okvira jer su se navikli na te funkcionalnosti no zbog toga se ne smanjuje srednja vrijednost za prvi faktor.

Tablica 4.6. Rangovi srednjih vrijednosti za prva dva faktora

Faktor	**Rangovi srednjih vrijednosti**				
	2013/2014 (N=43)	2014/2015 (N=40)	2015/2016 (N=66)	2016/2017 (N=56)	2017/2018 (N=81)
1	109.98	100.58	125.36	171.66	177.80
2	242.79	193.73	93.27	131.05	115.52

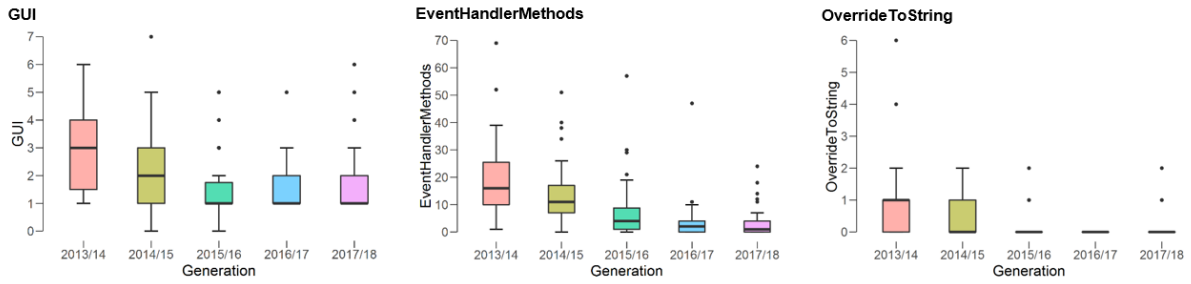
Potrebno je napomenuti da su u analizu za prve dvije generacije uključeni samo oni projekti koji su usporedivi. Naime, ukupan broj studenata koji su uključeni u OOP za prve dvije generacije je 144, ali samo 109 studenata je izradilo projekte za ispit. Pri tome je manji dio studenata odabrao igru kao temu projekta (Tablica 4.7).

Tablica 4.7. Projekti igre u prve dvije generacije

Vrsta igre	**2013/14** N=71	**2014/15** N=73
Studenti koji su radili igru	9	32
Studenti koji su radili u nekom okviru	7	8
Studenti koji su radili u naprednom okviru (Unity, XNA)	0	7

Dio studenata koji su radili igre kao projekt koristili su pored OTTER-a i neke druge okvire kao što su XNA ili Unity, no ti projekti nisu usporedivi s ostalima. Unity je posebno razvojno okruženje s konceptima koji su isključivo vezani za igre te ih nema u MS Visual Studio okruženju (na primjer položaj kamere, scene, i sl.). Također, XNA je poseban okvir namijenjen profesionalnom razvoju igara što OTTER ipak nije. OTTER se temelji na Windows Forms aplikacijama tako da su projekti izrađeni bez okvira i projekti izrađeni u OTTER-u zapravo zasnovani na istim konceptima programskog jezika C#.

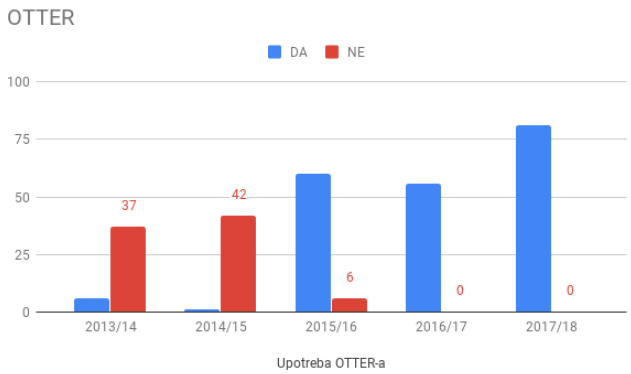
Faktori koji su identificirani faktorskom analizom grupiraju varijable. Faktor 2 koji se odnosi na koncepte grafičkog korisničkog sučelja sastoji se od samo tri varijable tako da ćemo ih ovdje prikazati grafički jer se vidi razlika u broju primijenjenih koncepata po generacijama (Slika 4.15) uz napomenu da slika nije dovoljna za donošenje zaključaka.



Slika 4.15. Faktor 2 (JASP Team, 2019)

Varijable kojima se vrijednosti kreću u malim intervalima nije moguće pregledno prikazati (npr. maksimalna dubina nasljeđivanja je iz intervala [0,3]).

Obzirom da se koristi MSVS kao integrirano razvojno okruženje (eng. integrated development environment, IDE), onda se kod stvaranja GUI elemenata kao što su Form, Button, TextBox i sl. ne može očekivati razumijevanje stvaranja objekata. Naime, studenti koriste povlačenje (eng. drag & drop) tih kontrola iz prozora s alatima (eng. toolbox), a pri stvaranju metoda koje se koriste za upravljanje događajima studenti mogu jednostavno dva puta kliknuti na odgovarajući događaj u prozoru „Properties“. Premda takvi postupci pojednostavljuju cijeli proces izrade programa te smanjuju mogućnosti sintaksnih pogrešaka, problem je što ne zahtijevaju razmišljanje o detaljima. Na primjer, za metode koje IDE automatski stvori nije potrebno razmišljati koji su parametri potrebni. Tijekom usmenog ispita studenti su morali objasniti svoju implementaciju projekta kao i odabire koncepata kod oblikovanja. Nastavnici su tijekom početnih faza primjene okvira počeli primjećivati kako su studenti koji su koristili okvir davali smislenija objašnjenja primijenjenih objektno-orijentiranih koncepata. To je bio jedan od najvažnijih razloga zašto je upotreba okvira postala obvezna. Dodatni razlog je što su studenti u prethodnim semestrima već imali prilike upoznati okruženje MSVS tako da si nastoje svesti problem na poznato područje, a također je vrlo jednostavno pronaći upute tipa „korak po korak“ online ili čak cijele projekte. U skladu s tim, studenti su mogli reproducirati gotovo rješenje te se uz pomoć objašnjenja čak i ciljano pripremiti za taj projekt. Međutim, prezentiranje tuđeg projekta na prvi pogled može izgledati dobro (nastavniku je teže detektirati da student nije sam izradio projekt), ali je i samim studentima u toj situaciji teško i razumjeti i objasniti detalje.

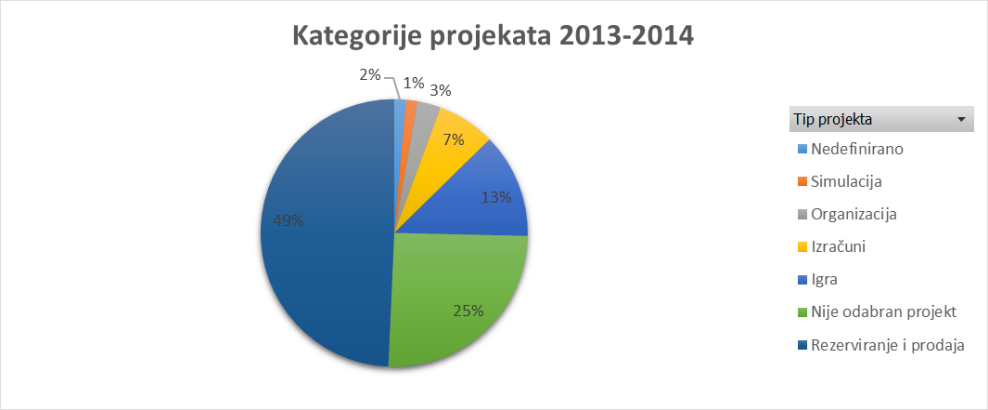


Slika 4.16. Upotreba OTTER okvira po generacijama

Promatranjem srednjih vrijednosti ne možemo dokazati razlike među grupama, a KWT test nam ne daju dovoljno informacija o razlikama među grupama, tako da je bilo potrebno analizirati pojedine parove odvojeno upotrebom Mann-Whitney testa. Detaljan prikaz usporedbi je u prilogu. Ukratko, rezultati su pokazali da postoji statistički značajna razlika upravo među

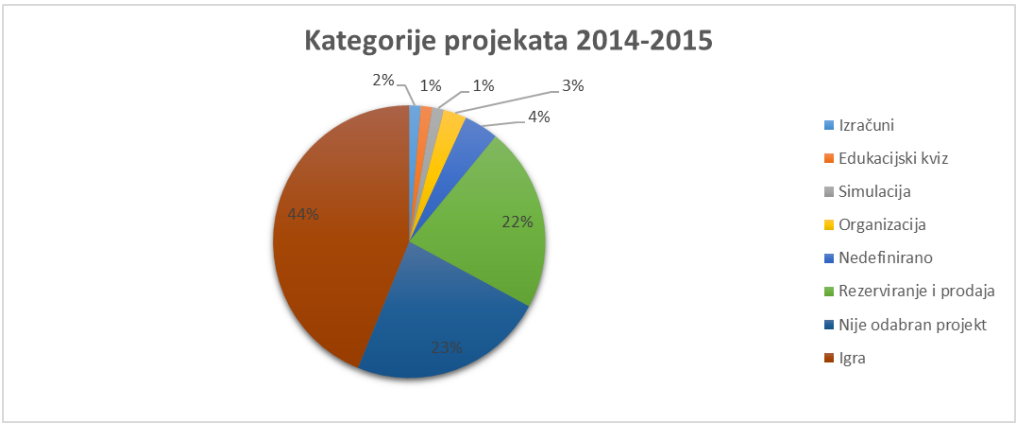
generacijama koje su koristile okvir u odnosu na generacije koje nisu koristile OTTER uopće ili su mogli birati. Sve generacije su bile različite prema parametrima Faktor 1 i Faktor 2, osim generacija 2016/2017 i 2017/2018.

Prve dvije generacije studenata analizirane su i „ručno“ kako bi se identificirale vrste projekata. U prvoj generaciji većina projekata je tipa *rezervacija i prodaja* (Slika 4.17). Na primjer, prodaja karata za kino, avion i sl. Dio studenata nije imao precizno definiranu temu projekata.



Slika 4.17. Kategorije projekata za ak. god. 2013/14

Sljedeća generacija studenata prebacuje interes na područje igara umjesto „poslovnih“ aplikacija (Slika 4.18).

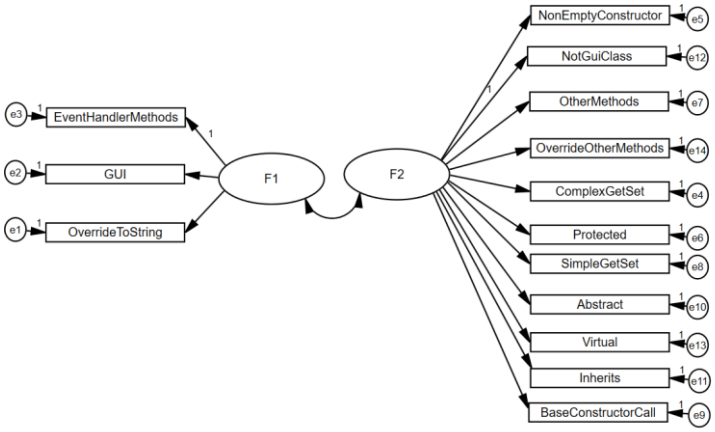


Slika 4.18. Kategorije projekata za ak. god. 2014/15

Problem kod ocjenjivanja prvih dviju generacija koje nisu koristi OTTER je utvrđivanje originalnosti studentskog rješenja, odnosno sprječavanje preuzimanja gotovih radova dostupnih online. Pri tome razlikujemo dvije vrste: preuzimanje i predstavljanje cijelog projekta kao vlastitog te preuzimanje i nadogradnja postojećeg projekta. Prva vrsta spada pod plagijat, dok druga ima sličnu svrhu kao i OTTER te studenti u principu sami objašnjavaju što je preuzeto, a što su nadogradili.

Pregledom varijabli uključenih u analizu uočavamo vrijednosti za *asimetričnost* (eng. skewness) i *spljoštenost* (eng. kurtosis) (Prilog 4) koje su za većinu varijabli veće od preporučenih 3.3 prema (Sposito, Hand, & Skarpness, 1983) što sugerira da većina podataka zapravo nema normalnu distribuciju.

Na temelju identificiranih faktora primjenom PCA analize izrađen je SEM model 1.



Slika 4.19. SEM model 1

Primjenom metode GLS dobivamo različite indekse: CMIN/DF iznosi 5,942 što je veće od 5, RMSEA je 0,131 uz PCLOSE <0,05 što znači da koeficijenti sukladnosti nisu u preporučenim intervalima da bi model bio sukladan. Vrijednost komparativnog indeksa AIC je 509,563, a hi-kvadrat je 451,563, *df* = 76. RMR iznosi 18,339, GFI 0,776 (treba biti veći od 0,90), a SRMR je 0,2597. Primjenom metode ULS u Amos-u dobije se nešto manje koeficijenata: RMR je 3,816, GFI je 0,981, a SRMR je 0,11.

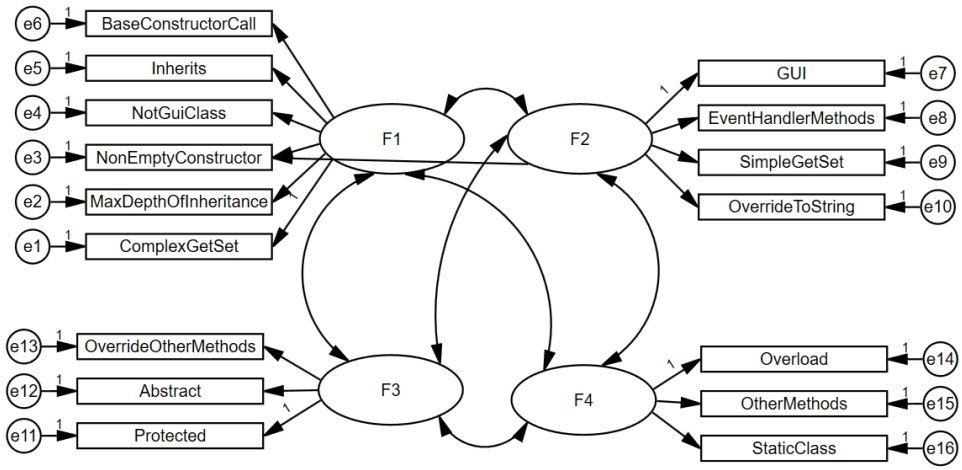
Uvođenjem dodatnih kovarijanci između pogrešaka dobivamo nešto „bolji“ model (AIC 405,592), koji ipak ne zadovoljava ostale kriterije sukladnosti za GLS, a za ULS dobivamo RMR = 1,999, GFI = 0,995 te SRMR = 0,097. Mahalanobis udaljenost za prvih nekoliko slučajeva sugerira da postoje ekstremne vrijednosti koje vjerojatno utječu na model.

Tablica 4.8. Mahalonobis udaljenost (dio)

Observation number	Mahalanobis d-squared
206	146,265
41	126,043
55	93,384
70	93,384
77	93,384
151	73,314

Provjerom osnovne statističke analize uočavamo da postoje ekstremne vrijednosti za neke varijable. Postavlja se pitanje smijemo li pobrisati te podatke iz analize, odnosno možemo li izbaciti potencijalno dobre projekte zbog ekstremnih vrijednosti. Detaljnijim proučavanjem konkretnih projekata zaključujemo da postoji velika vjerojatnost da su studenti preuzeli gotov projekt, a svi su predali isti kao grupni projekt. Projekt je sadržavao broj metoda koji je bio neusporedivo veći u odnosu na ostale projekte (103 metode) te broj preopterećenja (107). Nakon izbacivanja tih projekata preostaje 286 projekata. Za ULS metodu novi podaci poboljšavaju sukladnost modela pa je tako RMR = 0,867, GFI = 0,996 i SRMR = 0,081 što se može smatrati prihvatljivim modelom.

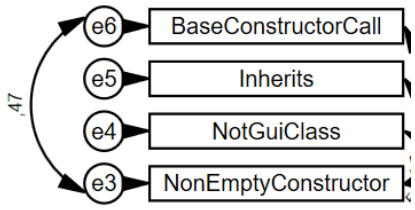
Međutim, promjena u podacima može biti značajna i za početnu PCA analizu i dobivanje inicijalnih faktora. Provedena je druga faktorska analiza na 286 projekata te su dobivena nova 4 faktora od kojih je napravljen SEM model tako da je novo stanje najbolje vidljivo iz slike modela (Slika 4.20). Statistička analiza je ista kao za prethodni model te su detaljnije tablice u prilogu.



Slika 4.20. SEM model 2

Premda je broj faktora promijenjen, opet se ističe razdvajanje objektno orijentiranih koncepata od koncepata vezanih za grafičko korisničko sučelje. Faktor F2 odnosi se na grafičko korisničko sučelje, ali ovaj put je dodatni indikator i *SimpleGetSet* odnosno oznaka za jednostavna svojstva koja se mogu automatski generirati pomoću razvojnog okruženja te samim tim ne zahtijevaju od studenata posebno razumijevanje. Vrijednost za hi-kvadrat je 347,187, *df*=97, ali koeficijenti: CMIN/df = 3,579, RMSEA = 0,095 za p<0,05 i dalje ukazuju da model za GLS ne odgovara postavljenim granicama iz literature premda su vrijednosti poboljšane u odnosu na prethodnu analizu. AIC koeficijent je 425,187.

Provjera indikatora promjene sugerira povezivanje kovarijancom pogrešaka e3 i e6 (Slika 4.21).



Slika 4.21. Promjena modela

S tom intervencijom u model dolazimo do poboljšanja koeficijenata (npr. AIC postaje 394,507). Izmjena modela ima smisla jer varijabla *BaseConstructorCall* označava poziv osnovnog konstruktora dok *NonEmptyConstructor* označava konstruktor kojem tijelo nije prazno. Indikator *Inherits* znači da je korišteno nasljeđivanje. Svi pripadaju istom faktoru, znači korelirani su što dalje znači da ima smisla povezati varijable. Hi-kvadrat je u tom slučaju 314,507, *df*=96. Za ULS metodu model nije izračunat, tako da je primijenjena metoda SFLS te su rezultati SRMR = 0.0889, RMR = 1.958, GFI = 0.943, a ako se povežu odabrane pogreške onda je SRMR = 0.0875, a RMR = 1.951, GFI = 0.945. Možemo zaključiti da je prema tim kriterijima model prihvatljiv (GFI > 0.90, stroži kriterij GFI > 0.95) dok je po SRMR na granici, ali opet treba napomenuti da su oba modela izrađena s ciljem boljeg predočavanja rezultata PCA (eksplorativna svrha) te se tim ne postavlja neka nova teorija već ispituje rezultat hipoteze.

5 ZAKLJUČAK

Učenje i poučavanje programiranja se općenito smatra teškim te često radovi na tu temu počinju od te pretpostavke. Problemi se mogu analizirati počevši od svih strana didaktičkog trokuta: učenika (koga poučavamo), sadržaja (što poučavamo), učitelja (kako poučavamo) i konteksta u kojem se nalazi trokut. Fokus istraživanja su studenti PMFST kao odrasli početnici u programiranju kojima treba pristupiti na odgovarajući način. Sadržaj poučavanja promatran u ovom radu odnosi se na koncepte objektno orijentirane paradigme, a način poučavanja je primjena didaktičkog skrivanja u projektnoj nastavi.

Obzirom da nema jedinstvenog rješenja koje bi riješilo sve probleme poučavanja programiranja, a stalno se mijenjaju i svi dijelovi trokuta: učenici postaju drugačiji, oblikuju se novi jezici i okruženja, osmišljavaju se novi pristupi poučavanju, javljaju se drugačije potrebe za poslovima u industriji, razvija se nova tehnologija, i sl., potrebno je kontinuirano istraživati i prilagođavati se. Intervencije oko sadržaja poučavanja često se odnose na jezik tako da kod učenika niže dobi

koriste posebni jezici za poučavanje koji su reducirani u odnosu na profesionalne te vizualni jezici koji eliminiraju probleme sintakse.

Prema primjerima dobre prakse iz literature vezanima uz korištenje vizualnih programskih jezika provedeno je pilot istraživanje u kojem su se analizirali podaci 727 studenta koji su učili programirati pomoću vizualnih programskih jezika kako bi se utvrdio učinak na njihov uspjeh iz programiranja. Osim pozitivnog dojma kojeg su nastavnici primijetili za vrijeme rada s vizualnim jezicima, zaključeno je kako takvi jezici općenito nisu primjereni za odrasle početnike, odnosno studente premda se mogu koristiti kao alat za brzu demonstraciju nekog koncepta. Vrijeme u kojem oni to prerastu je prilično brzo, a ne prenose naučene koncepte u druga okruženja te nakon početnih predmeta iz programiranja i dalje imaju probleme. Pozitivan učinak na uspjeh studenata je bio zanemariv te je mogao nastati slučajno. Prvi susret s programiranjem početnika na PMFST je proceduralna paradigma u Pythonu, te zatim objektno orijentirana paradigma u C#.

Učenje objektno orijentiranog programiranja je također teško jer se mijenja način razmišljanja. Premda se na prvi pogled čini da bi studentima trebao biti bliži jer se objektima modeliraju elementi vidljivi u stvarnim svijetu to ipak nije tako. Rasprava oko toga s kojom paradigmom krenuti još uvijek traje, ali bez obzira na to, studenti bi se trebali upoznati s više paradigmi. U svakom slučaju se mora dogoditi prijelaz iz jedne paradigme u drugu. To predstavlja problem, ali naši studenti nakon prvog semestra proceduralne paradigme još ne postaju eksperti tako da je to manje izraženo.

Obzirom da su vizualni elementi izvršavanja programa u vizualnim programskim jezicima ipak pozitivno djelovali na studente razmatrane su mogućnosti kako iskoristiti tu komponentu u profesionalnom programskom jeziku C#. Dodatno je aspekt interakcije likova na zaslonu računala na prvi pogled djelovao motivirajuće u odnosu na tekstualni ispis rezultata. Profesionalna okruženja i jezici su prilično složeni te stvaraju kognitivno opterećenje studentima koji onda nisu u stanju usvojiti osnovne koncepte programiranja. Uvođenje neke dodatne biblioteke koja bi omogućila izradu programa sličnog oblika kao u Scratch-u bi značilo još mnogo nove sintakse, a kako su te biblioteke najčešće namijenjene izradi igara, onda to nužno vodi i prema potrebi učenja koncepata isključivo vezanih za računalne igre. Prema tome bi se umjesto smanjivanja kognitivnog opterećenja ono povećalo, ali irelevantnim konceptima. Osmišljavanjem jednostavnog okvira s minimalnim funkcionalnostima koje su sasvim dovoljne za izradu projekata nije se bitno povećalo kognitivno opterećenje jer se u samom okviru koriste isključivo elementi grafičkog korisničkog sučelja kojeg studenti imaju stalno na raspolaganju

(bez uvođenja dodatnih biblioteka). Pri tome je bilo potrebno sakriti koncepte koje studenti uče tek kasnije (paralelno programiranje). Skriveni su svi složeni elementi koji nisu nužni za razumijevanje objektno orijentiranih koncepata čije usvajanje je cilj predmeta. Jedan od problema koje smo pri tome uočili je vraćanje u „sigurnu zonu“ odnosno poznato okruženje gdje su studenti izbjegavali primjenu okvira sve dok nastavnici nisu više počeli inzistirati na upotrebi okvira te vježbe prilagodili okviru.

OTTER nije namijenjen izradi igara već je oblikovan prateći koncepte Scratch-a kako bi studenti mogli projekte izrađene u tom okruženju stvoriti i u C#-u uz što manje problema. Nakon što su studenti morali koristiti OTTER, pokazalo se kao prednost što je OTTER bio jedinstven i posebno prilagođen njihovim predznanjem pa nisu imali mogućnost dohvaćanja gotovih rješenja online te su morali razmišljati i znati objasniti svoj dizajn projekta. Tijekom više godina primjene OTTER-a, prikupljeno je 289 studentskih projekata. Analiza projekata provedena je posebnim alatom razvijenim upravo za tu namjenu. Na taj se način osigurava objektivnost analize obzirom da su ljudi skloni pogreškama zbog zamora i sl. Dio projekata je analizirao čovjek, ali je to pored mogućnosti pogrešaka također i dugotrajan proces. Pomoću alata za analizu izdvojeni su elementi primijenjeni u okviru te prebrojani objektno orijentirani koncepti koje su studenti koristili uz napomenu da su iz analize izbačeni koncepti samog okvira koje studenti nisu koristili tako da složenost okvira ne utječe na rezultate pa samim tim ni činjenica da se okvir mijenjao tijekom vremena. Ista analiza je bila primjenjiva i na projekte izrađene bez okvira jer analizator ne računa početni okvir kao studentsko postignuće već broji samo nadograđene elemente.

Faktorskom analizom provedeno je grupiranje varijabli pri čemu su se izdvojili posebno koncepti vezani za grafičko korisničko sučelje te koncepti vezani za OOP. Potrebno je naglasiti da su koncepti vezani za grafičko korisničko sučelje koji su pobrojani u projektima te izdvojeni zasebno zapravo koncepti koji se obično generiraju automatski pomoću razvojnog okruženja tako da na primjer student može dobiti metodu s par klikova miša čak bez poznavanja sintakse.

Dodatno je provedena i SEM analiza kako bi se potvrdilo istraživanje, ali i utvrdilo može li se zaključiti nešto drugačije s drugom metodom. Tijekom SEM analize utvrđeno je da veliku ulogu imaju ekstremi te su određeni projekti izuzeti iz daljnje analize (ukupno tri projekta). Projekte je bilo potrebno detaljnije analizirati kako bi mogli utvrditi smijemo li ih izbaciti bez utjecaja na varijabilnost podataka. Nakon isključivanja ekstrema, prvi postavljeni model ima bolju sukladnost, ali ponavljanjem faktorske analize na 286 projekata dobije se nešto drugačiji raspored faktora, no ukupni rezultati ipak dovode do istog zaključka.

Pokazalo se da su studenti koji su koristili OTTER koristili više objektno orijentiranih koncepata koje su na usmenom dijelu ispita znali objasniti. Studenti koji nisu morali koristiti OTTER su imali manje smislenih objektno orijentiranih koncepata, npr. veliki broj metoda koje su stvorene su zapravo bile automatski generirane u okruženju te nisu znali objasniti koncepte na usmenom dijelu ispita.

Iz svega navedenog slijedi kako je opravdano korištenje metode didaktičkog skrivanja pri poučavanju objektno orijentiranog programiranja što je potvrđeno značajnim povećanjem broja primijenjenih koncepata iz objektno orijentiranog programiranja u realizaciji konačnih projekata. Unatoč povećanom opterećenju nastavnika projektni pristup pri uporabi naučenih koncepata dobar je pokazatelj broja usvojenih koncepata kod grupe studenata koji su koristili okvir. U budućim istraživanjima moglo bi se ispitati koliko se metoda didaktičkog skrivanja i projektni pristup može uspješno primijeniti na poučavanje programiranja drugih programskih paradigmi, ali i područja poput umjetne inteligencije ili unutar toga podatkovne znanosti.

LITERATURA

Abreu, F. B. e., & Melo, W. (1996). Evaluating the Impact of Object-Oriented Design on Software Quality. *Proceedings of the 3rd International Symposium on Software Metrics: From Measurement to Empirical Results*, 90–. Preuzeto od http://dl.acm.org/citation.cfm?id=525586.823874

Aivaloglou, E., & Hermans, F. (2016). How Kids Code and How We Know: An Exploratory Study on the Scratch Repository. *Proceedings of the 2016 ACM Conference on International Computing Education Research - ICER '16*, 53–61. https://doi.org/10.1145/2960310.2960325

Ala-Mutka, K. (2004). Problems in learning and teaching programming–a literature study for developing visualizations in the Codewitz-Minerva project. *Codewitz Needs Analysis*, 1–13.

Alimisis, D. (2009). Teacher education on robotics-enhanced constructivist pedagogical methods. *School of Pedagogical and Technological Education, Athens*.

Anderson, L. W., Krathwohl, D. R., Airasian, P. W., Cruikshank, K. A., Mayer, R. E., Pintrich, P. R., … Wittrock, M. C. (2001). *A taxonomy for learning teaching and assessing: A revision of Bloom`s taxonomy of educational objetives*. Longman.

Arlegui, J., Menegatti, E., Moro, M., & Pina, A. (2008). Robotics, computer science curricula and interdisciplinary activities. *Proceedings of the TERECoP Workshop Teaching with robotics, Conference SIMPAR*, 10–21.

Armstrong, D. J. (2006). The quarks of object-oriented development. *Communications of the ACM*, *49*(2), 123–128. https://doi.org/10.1145/1113034.1113040

Atmatzidou, S., Markelis, I., & Demetriadis, S. (2008). The use of LEGO Mindstorms in elementary and secondary education: Game as a way of triggering learning. *Proceedings of the TERECoP Workshop Teaching with robotics, Conference SIMPAR*.

Bakar, A., Inal, Y., & Cagiltay, K. (2006). Use of commercial games for educational purposes: Will today's teacher candidates use them in the future? *EdMedia+ Innovate Learning*, 1757–1762. Association for the Advancement of Computing in Education (AACE).

Barrett, P. (2007). Structural equation modelling: Adjudging model fit. *Personality and Individual Differences*, *42*(5), 815–824. https://doi.org/10.1016/j.paid.2006.09.018

Bau, D., Bau, D. A., Dawson, M., & Pickens, C. S. (2015). Pencil Code: Block Code for a Text World. *Proceedings of the 14th International Conference on Interaction Design and Children*, 445–448. https://doi.org/10.1145/2771839.2771875

Bayman, P., & Mayer, R. E. (1988). Using conceptual models to teach BASIC computer programming. *Journal of Educational Psychology*, *80*(3), 291–298. https://doi.org/10.1037/0022-0663.80.3.291

Bell, S. (2010). Project-Based Learning for the 21st Century: Skills for the Future. *The Clearing House: A Journal of Educational Strategies, Issues and Ideas*, *83*(2), 39–43. https://doi.org/10.1080/00098650903505415

Ben-Ari, M. (2001). Constructivism in computer science education. *Jl. of Computers in Mathematics and Science Teaching*, *20*(1), 45–73. https://doi.org/10.1145/274790.274308

Bennedsen, J. (2008). *Teaching and Learning Introductory Programming—A model-Based Approach* (PhD Thesis). University of Oslo, Norway.

Bennedsen, J., & Caspersen, M. E. (2007). Failure Rates in Introductory Programming. *SIGCSE Bull.*, *39*(2), 32–36. https://doi.org/10.1145/1272848.1272879

Bennedsen, J., & Schulte, C. (2007). What does objects-first mean?: An international study of teachers' perceptions of objects-first. *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research-Volume 88*, 21–29. Australian Computer Society, Inc.

Berglund, A., & Lister, R. (2010). Introductory programming and the didactic triangle. *Proceedings of the Twelfth Australasian Conference on Computing Education-Volume 103*, 35–44. Australian Computer Society, Inc.

Bers, M. U., & Horn, M. M. S. (2010). Tangible programming in early childhood: Revisiting developmental assumptions through new technologies. *High-tech tots: Childhood in a digital world*, 1–32.

Bers, M. U., Ponte, I., Juelich, K., Schenker, J., Viera, A., & Schenker, J. (2002). Teachers as designers: Integrating robotics in early childhood education. *Information Technology in childhood education*, *1*, 123–145.

Bersin, J. (2014). The Myth Of The Bell Curve: Look For The Hyper-Performers. Preuzeto 20. kolovoz 2019., od Forbes website: https://www.forbes.com/sites/joshbersin/2014/02/19/the-myth-of-the-bell-curve-look-for-the-hyper-performers/

Berthiaume, D. (2008). Teaching in the disciplines. *A handbook for teaching and learning in higher education*, 215.

Blackwell, A. F. (2002). What is programming? U J. Kuljis, L. Baldwin, & S. Scoble (Ur.), *PPIG 2002—14th Annual Workshop* (str. 20). Psychology of Programming Interest Group.

Bloom, B. S. (1956). *Taxonomy of educational objectives. Book 1*. Preuzeto od https://www.uky.edu/~rsand1/china2018/texts/Bloom%20et%20al%20-Taxonomy%20of%20Educational%20Objectives.pdf

Boaler, J. (1998). Open and Closed Mathematics: Student Experiences and Understandings. *Journal for Research in Mathematics Education*, *29*(1), 41. https://doi.org/10.2307/749717

Boaventura, F. M. B., & Sarinho, V. T. (2017). MEnDiGa: A Minimal Engine for Digital Games. *International Journal of Computer Games Technology*, *2017*, 1–13. https://doi.org/10.1155/2017/9626710

Bobrow, D. G. (1985). If Prolog is the Answer, What is the Question? Or What it Takes to Support AI Programming Paradigms. *IEEE Transactions on Software Engineering, SE-11*(11), 1401–1408. https://doi.org/10.1109/TSE.1985.231888

Böhm, C., & Jacopini, G. (1966). Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM*, *9*(5), 366–371. https://doi.org/10.1145/355592.365646

Bolshakova, E. (2005). PROGRAMMING PARADIGMS IN COMPUTER SCIENCE EDUCATION. *International Journal „Information Theories & Applications“*, *12*, 285–290.

Boshernitsan, M., & Downes, M. S. (2004). *Visual programming languages: A survey*. University of California.

Bruce, C., Buckingham, L., Hynd, J., McMahon, C., Roggenkamp, M., & Stoodley, I. (2004). Ways of experiencing the act of learning to program: A phenomenographic study of introductory programming students at university. *Transforming IT education: Promoting a culture of excellence*, 301–325.

Bruner, J., Goodnow, J., & Austin, G. (2017). *A study of thinking*. Routledge.

Bruner, J. S. (1966). *Toward a theory of instruction* (Sv. 59). London: Harvard University Press.

Brusilovsky, P., Kouchnirenko, A., Miller, P., & Tomek, I. (1994, lipanj 25). *Teaching Programming to Novices: A Review of Approaches and Tools.* Predstavljeno na ED-MEDIA 94- World Conference on Educational Multimedia and Hypermedia, Vancouver, British Columbia, Canada.

Burrows, S., Tahaghoghi, S. M. M., & Zobel, J. (2007). Efficient plagiarism detection for large code repositories. *Software: Practice and Experience*, *37*(2), 151–175. https://doi.org/10.1002/spe.750

Cangur, S., & Ercan, I. (2015). Comparison of Model Fit Indices Used in Structural Equation Modeling Under Multivariate Normality. *Journal of Modern Applied Statistical Methods*, *14*(1), 152–167. https://doi.org/10.22237/jmasm/1430453580

Carlisle, M. C. (2009). Raptor: A visual programming environment for teaching object-oriented programming. *Journal of Computing Sciences in Colleges*, *24*(4), 275–281.

Caspersen, M. E. (2007). *Educating novices in the skills of programming* (PhD Dissertation). Department of Computer Science, University of Aarhus.

Caspersen, M. E., & Bennedsen, J. (2007). Instructional Design of a Programming Course: A Learning Theoretic Approach. *Proceedings of the Third International Workshop on Computing Education Research*, 111–122. https://doi.org/10.1145/1288580.1288595

Chen, W.-K., & Cheng, Y. C. (2007). Teaching object-oriented programming laboratory with computer game programming. *IEEE Transactions on Education*, *50*(3), 197–203.

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, *20*(6), 476–493. https://doi.org/10.1109/32.295895

Clancy, M. (2004). Misconceptions and Attitudes that Interfere with Learning to Program. U Sally Fincher & M. Petre (Ur.), *Computer Science Education Research* (str. 85–100). https://doi.org/10.1201/9781482287325-18

Clements, D. H., & McMillen, S. (1996). Rethinking" concrete" manipulatives. *Teaching children mathematics*, *2*(5), 270–279.

Cohen, L., Manion, L., & Morrison, K. (2007). *Research methods in education* (6th ed). London ; New York: Routledge.

Cohen, L., Manion, L., & Morrison, K. (2013). *Research Methods in Education* (7. izd.). Routledge.

Connolly, T. M., Boyle, E. A., MacArthur, E., Hainey, T., & Boyle, J. M. (2012). A systematic literature review of empirical evidence on computer games and serious games.

Computers & Education, *59*(2), 661–686. https://doi.org/10.1016/j.compedu.2012.03.004

Cooper, S., Dann, W., Pausch, R., & Pausch, R. (2003). Teaching Objects-first in Introductory Computer Science. *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, 191–195. https://doi.org/10.1145/611892.611966

Curtis, D. (2002). The power of projects. *Educational Leadership*, *60*, 50–53.

Danielsiek, H., Paul, W., & Vahrenhold, J. (2012). Detecting and Understanding Students' Misconceptions Related to Algorithms and Data Structures. *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, 21–26. https://doi.org/10.1145/2157136.2157148

Dankovčíková, Z. (2017). *Custom Roslyn Tool for Static Code Analysis* (Masaryk University, Faculty of Informatics). Preuzeto od https://is.muni.cz/th/f6oc3/?lang=en

Dann, W., Cosgrove, D., Slater, D., Culyba, D., & Cooper, S. (2012). Mediated transfer: Alice 3 to java. *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, 141–146. ACM.

David R. Heil, G. P., Burger, S. E., & Burger, S. E. (2013). Understanding Integrated STEM Education: Report on a National Study. *2013 ASEE Annual Conference*. Predstavljeno na Atlanta, Georgia. Atlanta, Georgia: ASEE Conferences.

De Graaf, E., & Kolmos, A. (2003). Characteristics of problem-based learning. *International Journal of Engineering Education*, *19*(5), 657–662.

Decker, R., & Hirshfield, S. (1993). Top-down teaching: Object-oriented programming in CS 1. *ACM SIGCSE Bulletin*, *25*, 270–273. ACM.

De-Marcos, L., Domínguez, A., Saenz-De-Navarrete, J., & Pagés, C. (2014). An empirical study comparing gamification and social networking on e-learning. *Computers & Education*, *75*, 82–91. https://doi.org/10.1016/j.compedu.2014.01.012

Devlin, K. (2001). The Real Reason Why Software Engineers Need Math. *Commun. ACM*, *44*(10), 21–22.

Du Boulay, B. (1986). Some Difficulties of Learning to Program. *Journal of Educational Computing Research*, *2*(1), 57–73. https://doi.org/10.2190/3LFX-9RRF-67T8-UVK9

Egenfeldt-Nielsen, S. (2006). Overview of research on the educational use of video games. *Nordic Journal of Digital Literacy*, *1*(03), 184–214.

Egenfeldt-Nielsen, S. (2007). Third generation educational use of computer games. *Journal of Educational Multimedia and Hypermedia*, *16*(3), 263–281.

Elkin, M., Sullivan, A., & Bers, M. U. (2014). Implementing a robotics curriculum in an early childhood Montessori classroom. *Journal of Information Technology Education: Innovations in Practice*, *13*, 153–169.

Emanuelsson, P., & Nilsson, U. (2008). A Comparative Study of Industrial Static Analysis Tools. *Electron. Notes Theor. Comput. Sci.*, *217*, 5–21. https://doi.org/10.1016/j.entcs.2008.06.039

Farooq, M. S., Khan, S. A., Ahmad, F., Islam, S., & Abid, A. (2014). An Evaluation Framework and Comparative Analysis of the Widely Used First Programming Languages. *PLOS ONE*, *9*(2), 1–25. https://doi.org/10.1371/journal.pone.0088941

Fee, S. B., & Holland-Minkley, A. M. (2010). Teaching computer science through problems, not solutions. *Computer Science Education*, *20*(2), 129–144. https://doi.org/10.1080/08993408.2010.486271

Fincher, S. (1999). What are we doing when we teach programming? *FIE'99 Frontiers in Education. 29th Annual Frontiers in Education Conference. Designing the Future of Science and Engineering Education. Conference Proceedings (IEEE Cat. No.99CH37011*, *1*, 12A4/1-12A4/5 vol.1. https://doi.org/10.1109/FIE.1999.839268

Fleury, A. E., & Fleury, A. E. (2000). Programming in Java: Student-constructed rules. *ACM SIGCSE Bulletin*, *32*(1), 197–201. https://doi.org/10.1145/330908.331854

Floyd, R. W. (1979). The Paradigms of Programming. *Commun. ACM*, *22*(8), 455–460. https://doi.org/10.1145/359138.359140

Fowler, M. (2003). *UML Distilled: A Brief Guide to the Standard Object Modeling Language* (3. izd.). New Jersey: Addison-Wesley Professional.

Frank, M., Lavy, I., & Elata, D. (2003). Implementing the Project-Based Learning Approach in an Academic Engineering Course. *International Journal of Technology and Design Education*, *13*(3), 273–288. https://doi.org/10.1023/A:1026192113732

Futschek, G. (2013). Extreme didactic reduction in computational thinking education. *X World Conference on Computers in Education*, 1–6.

Gallage, P. (2019). Programming paradigms, Software run-time architecture and Development tools [Blog]. Preuzeto 27. kolovoz 2019., od https://thepafhelper.blogspot.com/2019/02/programming-paradigms-softwear-runtime.html

Gallagher, S. A., Stepien, W. J., & Rosenthal, H. (1992). The Effects of Problem-Based Learning On Problem Solving. *Gifted Child Quarterly*, *36*(4), 195–200. https://doi.org/10.1177/001698629203600405

Garner, S., Haden, P., & Robins, A. (2005). My program is correct but it doesn't run: A preliminary investigation of novice programmers' problems. *Proceedings of the 7th Australasian conference on Computing education-Volume 42*, 173–180.

Georgantaki, S., & Retalis, S. (2007). *Using Educational Tools for Teaching Object Oriented Design and Programming*.

Gibbs, C., & Coady, Y. (2010). Understanding Abstraction: A Means of Leveling the Playing Field in CS1? *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, 169–174. https://doi.org/10.1145/1869542.1869569

Gilmore, D. J. (1990). Methodological Issues in the Study of Programming. U *Psychology of Programming* (str. 9–19). Academic Press.

Gold, M. S., Bentler, P. M., & Kim, K. H. (2003). A Comparison of Maximum-Likelihood and Asymptotically Distribution-Free Methods of Treating Incomplete Nonnormal Data. *Structural Equation Modeling: A Multidisciplinary Journal*, *10*(1), 47–79. https://doi.org/10.1207/S15328007SEM1001_3

Gomes, I., Morgado, P., Gomes, T., & Moreira, R. (2009). *An overview on the static code analysis approach in software development* (str. 16) [Technical Report]. Portugal: Faculdade de Engenharia da Universidade do Porto.

Grandell, L., Peltomäki, M., Back, R.-J., & Salakoski, T. (2006). Why complicate things?: Introducing programming in high school using Python. *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*, 71–80. Australian Computer Society, Inc.

Gray, S., St. Clair, C., James, R., & Mead, J. (2007). Suggestions for Graduated Exposure to Programming Concepts Using Fading Worked Examples. *Proceedings of the Third International Workshop on Computing Education Research*, 99–110. https://doi.org/10.1145/1288580.1288594

Grüner, G. (1967). Die didaktische Reduktion als Kernstück der Didaktik. *Die Deutsche Schule*, *59*(7/8), 414–430.

Guzdial, M. (2004). Programming Environments for Novices. *Computer science education research*, 127–154.

Hadjerrouit, S. (1998). Java As First Programming Language: A Critical Evaluation. *SIGCSE Bull.*, *30*(2), 43–47. https://doi.org/10.1145/292422.292440

Hadjerrouit, S. (1999). A Constructivist Approach to Object-oriented Design and Programming. *Proceedings of the 4th Annual SIGCSE/SIGCUE ITiCSE Conference on Innovation and Technology in Computer Science Education*, 171–174. https://doi.org/10.1145/305786.305910

Harrison, R., Counsell, S., & Nithi, R. (1997). An overview of object-oriented design metrics. *Proceedings Eighth IEEE International Workshop on Software Technology and Engineering Practice Incorporating Computer Aided Software Engineering*, 230–235. https://doi.org/10.1109/STEP.1997.615494

Hartness, K. (2004). Robocode: Using Games to Teach Artificial Intelligence. *J. Comput. Sci. Coll.*, *19*(4), 287–291.

Haugland, S. W. (2000). Early childhood classrooms in the 21st century: Using computers to maximize learning. *Young Children*, *55*(1), 12–18.

Henderson, P. B. (1986). Anatomy of an Introductory Computer Science Course. *Proceedings of the Seventeenth SIGCSE Technical Symposium on Computer Science Education*, 257–264. https://doi.org/10.1145/5600.5708

Hesser, T. L., & Gregory, J. L. (2015). Exploring the Use of Faded Worked Examples as a Problem Solving Approach for Underprepared Students. *Higher Education Studies*, *5*(6), 36. https://doi.org/10.5539/hes.v5n6p36

Hoc, J.-M., Green, T. R. G., Samurcay, R., & Gilmore, D. J. (Ur.). (1990). *Psychology of Programming*. Academic Press.

Hoegh, A., & Moskal, B. M. (2009). Examining Science and Engineering Students' Attitudes Toward Computer Science. *Proceedings of the 39th IEEE International Conference on Frontiers in Education Conference*, 1306–1311. https://doi.org/10.1109/FIE.2009.5350836

Holland, S., Griffiths, R., & Woodman, M. (1997). Avoiding object misconceptions. *ACM SIGCSE Bulletin*, *29*, 131–134. ACM.

Hooper, D., Coughlan, J., & Mullen, M. R. (2008). Structural Equation Modelling: Guidelines for Determining Model Fit. *Electronic Journal of Business Research Methods*, *6*(1), 8.

Hu, M. (2004). Teaching Novices Programming with Core Language and Dynamic Visualisation. *Proceedings of the 17th Annual Conference of the National Advisory Committee on Computing Qualifications (NACCQ 2004)*, 94–103.

Hubwieser, P., & Mühling, A. (2011). What Students (Should) Know About Object Oriented Programming. *Proceedings of the Seventh International Workshop on Computing Education Research*, 77–84. https://doi.org/10.1145/2016911.2016929

Hunt, G. H., Wiseman, D. G., & Touzel, T. J. (2009). *Effective teaching: Preparation and implementation*. Charles C Thomas Publisher.

IEEE-CS, & ACM. (2001). *Computing Curricula 2001 Computer Science* (Izd. CC2001 Computer Science volume; str. 240).

JASP Team. (2019). JASP (Verzija 0.11.1). Preuzeto od https://jasp-stats.org/

Jenkins, T. (2002). On the difficulty of learning to program. *Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences*, *4*, 53–58.

Jiang, L., Su, Z., & Chiu, E. (2007). Context-based detection of clone-related bugs. *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 55. https://doi.org/10.1145/1287624.1287634

Kaasbøll, J. J. (1998). Exploring didactic models for programming. *NIK 98–Norwegian Computer Science Conference*, 195–203. Citeseer.

Kansanen, P., & Meri, M. (1999). The didactic relation in the teaching-studying-learning process. *Didaktik/Fachdidaktik as Science (-s) of the Teaching profession*, *2*(1), 107–116.

Kapser, C., & Godfrey, M. W. (2003). Toward a Taxonomy of Clones in Source Code: A Case Study. *Proceedings of the Conference on Evolution of Large Scale Industrial Software Architectures (ELISA 2003)*, 67–78.

Kasper, D., & Ünlü, A. (2013). On the Relevance of Assumptions Associated with Classical Factor Analytic Approaches. *Frontiers in Psychology*, *4*. https://doi.org/10.3389/fpsyg.2013.00109

Kay, A. C. (1993). The Early History of Smalltalk. *The Second ACM SIGPLAN Conference on History of Programming Languages*, 69–95. https://doi.org/10.1145/154766.155364

Kay, J., Barg, M., Fekete, A., Greening, T., Hollands, O., Kingston, J. H., & Crawford, K. (2000). Problem-Based Learning for Foundation Computer Science Courses. *Computer Science Education*, *10*(2), 109–128. https://doi.org/10.1076/0899-3408(200008)10:2;1-C;FT109

Kearsley, G., & Shneiderman, B. (1998). Engagement Theory: A Framework for Technology-Based Teaching and Learning. *Educational technology*, *38*(5), 20–23.

Kelleher, C., & Pausch, R. (2003). *Lowering the Barriers to Programming: A Survey of Programming Environments and Languages for Novice Programmers:* https://doi.org/10.21236/ADA457911

Kenny, D. A. (2011). Terminology and Basics of SEM. Preuzeto 20. listopad 2019., od Terminology and Basics of SEM website: http://davidakenny.net/cm/basics.htm

Kiper, J. D., & Abernethy, K. (1996). Language Choice for CS1 and CS2: Experiences from Two Universities. *Computer Science Education*, *7*(1), 35–51. https://doi.org/10.1080/0899340960070103

Kirriemuir, J., & McFarlane, A. (2004). *Literature review in games and learning*.

Knowles, M. S. (1980). *The Modern Practice of Adult Education: From Pedagogy to Androgogy* (2. izd.). New York: Cambridge Books.

Knowles, M. S., Holton, E. F., & Swanson, R. A. (2005). *The adult learner: The definitive classic in adult education and human resource development*. Elsevier.

Kölling, M. (1999a). The problem of teaching object-oriented programming, Part 1: Languages. *Journal of Object-oriented programming*, *11*(8), 8–15.

Kölling, M. (1999b). The problem of teaching object-oriented programming, Part 2: Environments. *Journal of Object-Oriented Programming*, *11*(9), 6–12.

Kölling, M. (2008). Greenfoot: A highly graphical ide for learning object-oriented programming. *ACM SIGCSE Bulletin*, *40*, 327–327. ACM.

Kölling, M. (2019). *The BlueJ Tutorial Version 4.0*. Preuzeto od https://www.bluej.org/tutorial/tutorial-v4.pdf

Kölling, M., Quig, B., Patterson, A., & Rosenberg, J. (2003). The BlueJ System and its Pedagogy. *Computer Science Education*, *13*(4), 249–268. https://doi.org/10.1076/csed.13.4.249.17496

Kölling, M., & Rosenberg, J. (2001). Guidelines for teaching object orientation with Java. *ACM SIGCSE Bulletin*, *33*(3), 33–36.

Koorsse, M. (2012). *An Evaluation of Programming Assistance Tools to Support the Learning of IT Programming: A Case Study in South African Secondary Schools* (PhD Thesis, Nelson Mandela Metropolitan University). Preuzeto od http://www.sciencedirect.com/science/article/pii/S0360131514002735

Kori, K., Pedaste, M., Leijen, Ä., & Tõnisson, E. (2016). The Role of Programming Experience in ICT Students' Learning Motivation and Academic Achievement. *International Journal of Information and Education Technology*, *6*(5), 331.

Kramer, J. (2007). Is Abstraction the Key to Computing? *Commun. ACM*, *50*(4), 36–42. https://doi.org/10.1145/1232743.1232745

Kramer, M., Hubwieser, P., & Brinda, T. (2016). A Competency Structure Model of Object-Oriented Programming. *2016 International Conference on Learning and Teaching in Computing and Engineering (LaTICE)*, 1–8. https://doi.org/10.1109/LaTiCE.2016.24

Kramer, Matthias, Tobinski, D. A., & Brinda, T. (2016). On the Way to a Test Instrument for Object-oriented Programming Competencies. *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*, 145–149. https://doi.org/10.1145/2999541.2999544

Krpan, D., & Bilobrk, I. (2011). Introductory programming languages in higher education. *MIPRO, 2011 Proceedings of the 34th International Convention*, 1331–1336. IEEE.

Krpan, D., Mladenović, S., & Rosić, M. (2014). Undergraduate Programming Courses, Students' Perception and Success. *International Conference on New Horizons in Education INTE, Procedia - Social and Behavioral Sciences, Elsevier*, 3868–3872.

Krpan, D., Mladenović, S., & Zaharija, G. (2014). Vizualni programski jezici u visokom obrazovanju. *16. CARNetova korisnička konferencija - CUC 2014 - Zbornik radova*. Predstavljeno na Zagreb. Zagreb.

Krpan, D., Mladenović, S., & Zaharija, G. (2019). The Framework for Project Based Learning of Object-Oriented Programming. *International Journal of Engineering Education*, *35*(5), 1366–1377.

Kumar, R. (2012). *Research methodology: A step-by-step guide for beginners*. Sage Publications Limited.

Kunkle, W. M. (2010). *The Impact of Different Teaching Approaches and Languages on Student Learning of Introductory Programming Concepts* (PhD Thesis).

Kunkle, W. M., & Allen, R. B. (2016). The Impact of Different Teaching Approaches and Languages on Student Learning of Introductory Programming Concepts. *ACM Trans. Comput. Educ.*, *16*(1), 3:1–3:26. https://doi.org/10.1145/2785807

Larmer, J. (2014). Project-Based Learning vs. Problem-Based Learning vs. X-BL. *Retrieved March*, *8*, 2024.

Leutenegger, S., & Edgington, J. (2007). A Games First Approach to Teaching Introductory Programming. *SIGCSE Bull.*, *39*(1), 115–118. https://doi.org/10.1145/1227504.1227352

Li, F. W., & Watson, C. (2011). Game-based concept visualization for learning programming. *Proceedings of the third international ACM workshop on Multimedia technologies for distance learning*, 37–42. ACM.

Linn, M. C., & Clancy, M. J. (1992). The Case for Case Studies of Programming Problems. *Commun. ACM*, *35*(3), 121–132. https://doi.org/10.1145/131295.131301

Lister, M. (2015). Gamification: The effect on student motivation and performance at the post-secondary level. *Issues and Trends in Educational Technology*, *3*(2). https://doi.org/10.2458/azu_itet_v3i2_Lister

Lister, R. (2011). COMPUTING EDUCATION RESEARCH: Programming, Syntax and Cognitive Load. *ACM Inroads*, *2*(2), 21–22. https://doi.org/10.1145/1963533.1963539

Lister, R., Berglund, A., Clear, T., Bergin, J., Garvin-Doxas, K., Hanks, B., … Whalley, J. L. (2006). Research Perspectives on the Objects-early Debate. *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education*, 146–165. https://doi.org/10.1145/1189215.1189183

Liu, C. (2000). *Smalltalk, objects, and design*. iUniverse.

Livovský, J., & Porubän, J. (2014). Learning object-oriented paradigm by playing computer games: Concepts first approach. *Central European Journal of Computer Science*, *4*(3), 171–182. https://doi.org/10.2478/s13537-014-0209-2

Low Stanić, A. (2014). *Provjera postavki teorije međugrupne prijetnje u višeetničkoj zajednici nakon sukoba* (Doktorat). Filozofski fakultet, Zagreb.

Maloney, J., Resnick, M., & Rusk, N. (2010). The Scratch programming language and environment. *ACM Transactions on Computing Education*, *10*(4), 1–15. https://doi.org/10.1145/1868358.1868363.http

Margulieux, L. E., Guzdial, M., & Catrambone, R. (2012). Subgoal-labeled Instructional Material Improves Performance and Transfer in Learning to Develop Mobile Applications. *Proceedings of the Ninth Annual International Conference on International Computing Education Research*, 71–78. https://doi.org/10.1145/2361276.2361291

Marsh, H. W., Hau, K.-T., & Wen, Z. (2004). In Search of Golden Rules: Comment on Hypothesis-Testing Approaches to Setting Cutoff Values for Fit Indexes and Dangers in Overgeneralizing Hu and Bentler's (1999) Findings. *Structural Equation Modeling: A Multidisciplinary Journal*, *11*(3), 320–341. https://doi.org/10.1207/s15328007sem1103_2

Mason, J. (1994). Enquiry in mathematics and mathematics education. *Constructing mathematical knowledge: Epistemology and mathematics education*, 190–200.

Mason, Raina, & Cooper, G. (2012). Why the bottom 10% just can't do it—Mental Effort Measures and Implication for Introductory Programming Courses. *Proceedings of the Fourteenth Australasian Computing Education Conference (ACE2012)*, *123*, 11. Melbourne, Australia.

Mason, Raina, & Cooper, G. (2013). Distractions in Programming Environments. *Proceedings of the 15th Australasian Conference on Computing Education - ACE'13*, 23–30.

Mason, Rainalee. (2012). *Designing introductory programming courses: The role of cognitive load* (PhD thesis). Southern Cross University. Lismore, NSW.

Mathews, D. K. (2017). Predictors of Success in Learning Computer Programming. *Open Access Dissertations*, *Paper 596*. Preuzeto od http://digitalcommons.uri.edu/oa_diss/596

Matijević, M. (2010). Između didaktike nastave usmjerene na učenika i kurikulumske teorije. *U: Zbornik radova Četvrtog kongresa matematike. Zagreb: Hrvatsko matematičko društvo i Školska knjiga*, 391–408.

Mayer, R. E. (2002). A taxonomy for computer-based assessment of problem solving. *Computers in Human Behavior*, *18*(6), 623–632. https://doi.org/10.1016/S0747-5632(02)00020-1

McAllister, G., & Alexander, S. (2008). Key aspects of teaching and learning in computing science. *A Handbook for Teaching and Learning in Higher Education*, 282.

McCartney, R., Boustedt, J., Eckerdal, A., Sanders, K., & Zander, C. (2013). Can First-year Students Program Yet?: A Study Revisited. *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*, 91–98. https://doi.org/10.1145/2493394.2493412

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., … Wilusz, T. (2001). A Multi-national, Multi-institutional Study of Assessment of Programming Skills of First-year CS Students. *SIGCSE Bull.*, *33*(4), 125–180. https://doi.org/10.1145/572139.572181

McGill, T. J., & Volet, S. E. (1997). A Conceptual Framework for Analyzing Students' Knowledge of Programming. *Journal of Research on Computing in Education*, *29*(3), 276–297. https://doi.org/10.1080/08886504.1997.10782199

McKeachie, W., & Svinicki, M. (2013). *McKeachie's teaching tips*. Preuzeto od https://www.amazon.de/McKeachies-Teaching-Tips-Strategies-University/dp/1133936792

McNerney, T. S. (2004). From turtles to Tangible Programming Bricks: Explorations in physical language design. *PERSONAL AND UBIQUITOUS COMPUTING*, *8*(5), 326–337. https://doi.org/10.1007/s00779-004-0295-6

Medeiros, R. P., Ramalho, G. L., & Falcão, T. P. (2019). A Systematic Literature Review on Teaching and Learning Introductory Programming in Higher Education. *IEEE Transactions on Education*, *62*(2), 77–90. https://doi.org/10.1109/TE.2018.2864133

Meerbaum-Salant, O., Armoni, M., & Ben-Ari, M. (2011). Habits of programming in scratch. *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, 168–172. https://doi.org/10.1145/1999747.1999796

Meerbaum-Salant, O., Armoni, M., & Ben-Ari, M. (2013). Learning Computer Science Concepts with Scratch. *Computer Science Education*, *23*(3), 239–264. https://doi.org/10.1080/08993408.2013.832022

Merriam, S. B. (2001). Andragogy and Self-Directed Learning: Pillars of Adult Learning Theory. *New Directions for Adult and Continuing Education*, *2001*(89), 3–14. https://doi.org/10.1002/ace.3

Mesch, F. (1994). Didactic reduction by theory, with special attention to measurement education. *Measurement*, *14*(1), 15–22. https://doi.org/10.1016/0263-2241(94)90039-6

Michaelson, G. (2018). Microworlds, Objects First, Computational Thinking and Programming. U M. S. Khine (Ur.), *Computational Thinking in the STEM Disciplines* (str. 31–48). https://doi.org/10.1007/978-3-319-93566-9_3

Micheli, E., Avidano, M., & Operto, F. (2008). Semantic and epistemological continuity in educational robots' programming languages. *Proceedings of the TERECoP Workshop Teaching with robotics, Conference SIMPAR*, 80–90. Preuzeto od http://www.dei.unipd.it/ emg/downloads/SIMPAR08-nuovo/TeachingWithRobotics/Micheli_et_al.pdf

Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological review*, *63*(2), 81–97.

Milne, I., & Rowe, G. (2002). Difficulties in learning and teaching programming—Views of students and tutors. *Education and Information technologies*, *7*(1), 55–66.

Mladenović, S., Krpan, D., & Mladenović, M. (2016). Using Games to Help Novices Embrace Programming: From Elementary to Higher Education. *International Journal of Engineering Education*, *32*(1), 521–531.

Monig, J., Ohshima, Y., & Maloney, J. (2015). Blocks at your fingertips: Blurring the line between blocks and text in GP. *Proceedings - 2015 IEEE Blocks and Beyond Workshop*, 51–53. https://doi.org/10.1109/BLOCKS.2015.7369001

Moors, L., Luxton-Reilly, A., & Denny, P. (2018). Transitioning from Block-Based to Text-Based Programming Languages. *2018 International Conference on Learning and Teaching in Computing and Engineering (LaTICE)*, 57–64.

Moursund, D. G. (1999). *Project-based learning using information technology* (1. izd.). Eugene, OR: International Society for Technology in Education.

Myers, B. A. (1990). Taxonomies of visual programming and program visualization. *Journal of Visual Languages & Computing*, *1*(1), 97–123. https://doi.org/10.1016/S1045-926X(05)80036-9

Nevison, C., & Wells, B. (2003). Teaching objects early and design patterns in Java using case studies. *ACM SIGCSE Bulletin*, *35*, 94–98. ACM.

Ng, E., & Bereiter, C. (1991). Three Levels of Goal Orientation in Learning. *Journal of the Learning Sciences*, *1*(3–4), 243–271. https://doi.org/10.1080/10508406.1991.9671972

Nilson, L. B. (2003). *Teaching at Its Best: A Research-Based Resource for College Instructors* (Sv. 2nd). https://doi.org/10.4103/0973-1075.76236

Noone, M., & Mooney, A. (2018). Visual and textual programming languages: A systematic review of the literature. *Journal of Computers in Education*, *5*(2), 149–174. https://doi.org/10.1007/s40692-018-0101-5

Nuutila, E., Törmä, S., & Malmi, L. (2005). PBL and Computer Programming—The Seven Steps Method with Adaptations. *Computer Science Education*, *15*(2), 123–142. https://doi.org/10.1080/08993400500150788

O'Boyle Jr., E., & Aguinis, H. (2012). The Best And The Rest: Revisiting The Norm Of Normality Of Individual Performance. *Personnel Psychology*, *65*(1), 79–119. https://doi.org/10.1111/j.1744-6570.2011.01239.x

Olsson, U. H., Foss, T., Troye, S. V., & Howell, R. D. (2000). The Performance of ML, GLS, and WLS Estimation in Structural Equation Modeling Under Conditions of Misspecification and Nonnormality. *Structural Equation Modeling: A Multidisciplinary Journal*, *7*(4), 557–595. https://doi.org/10.1207/S15328007SEM0704_3

Ormerod, T. (1990). Human Cognition and Programming. U *Psychology of Programming* (str. 9–19). Academic Press.

Otey, M. (1997). Visual Studio 97. Preuzeto 20. rujan 2019., od ITPro Today website: https://www.itprotoday.com/microsoft-visual-studio/visual-studio-97

Paas, F., Renkl, A., & Sweller, J. (2004). Cognitive Load Theory: Instructional Implications of the Interaction between Information Structures and Cognitive Architecture. *Instructional Science*, *32*(1/2), 1–8. https://doi.org/10.1023/B:TRUC.0000021806.17516.d0

Pair, C. (1990). Programming, Programming Languages and Programming Methods. U *Psychology of Programming* (str. 9–19). Academic Press.

Papanikolaou, K., Frangou, S., & Alimisis, D. (2008). Teachers as designers of robotics-enhanced projects: The TERECoP course in Greece. *Proceedings of the TERECoP Workshop Teaching with robotics, Conference SIMPAR*, 100–111.

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. New York: Basic Books.

Pattis, R. E. (1981). *Karel the Robot: A Gentle Introduction to the Art of Programming* (1st izd.). New York, NY, USA: John Wiley & Sons, Inc.

Pattis, R. E. (1993). The "procedures early" approach in CS 1: A heresy. *Proceedings of the Twenty-fourth SIGCSE Technical Symposium on Computer Science Education*, 122–126. https://doi.org/10.1145/169070.169362

Paul, W., & Vahrenhold, J. (2013). Hunting High and Low: Instruments to Detect Misconceptions Related to Algorithms and Data Structures. *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, 29–34. https://doi.org/10.1145/2445196.2445212

Pedroni, M. (2003). *Teaching introductory programming with the inverted curriculum approach* (Diploma thesis). Department Computer Science, ETH Zurich.

Perkins, D. N., Hancock, C., Hobbs, R., Martin, F., & Simmons, R. (1986). Conditions of Learning in Novice Programmers. *Journal of Educational Computing Research*, *2*(1), 37–55. https://doi.org/10.2190/GUJT-JCBJ-Q6QU-Q9PL

Perkins, D. N., & Salomon, G. (1988). Teaching for transfer. *Educational Leadership*, *46*(1), 22–32.

Perkins, D. N., & Salomon, G. (1992). Transfer of Learning. U *International Encyclopedia of Education* (2. izd., str. 13). Oxford, England: Pergamon Press.

Petre, M. (1990). Expert Programmers and Programming Languages. U *Psychology of Programming* (str. 9–19). Academic Press.

Petrovič, P., & Balogh, R. (2008). Educational Robotics Initiatives in Slovakia. *Proceedings of the TERECoP Workshop Teaching with robotics, Conference SIMPAR*, 122–131.

Piaget, J., & Inhelder, B. (2013). *The growth of logical thinking from childhood to adolescence: An essay on the construction of formal operational structures* (Sv. 84). Routledge.

Pohl, M., Rester, M., & Judmaier, P. (2009). Interactive Game Based Learning: Advantages and Disadvantages. U C. Stephanidis (Ur.), *Universal Access in Human-Computer Interaction. Applications and Services* (Sv. 5616, str. 92–101). https://doi.org/10.1007/978-3-642-02713-0_10

Poljak, V. (1990). *Didaktika* (8. izd.). Školska knjiga.

Powers, K. D., & Powers, D. T. (1999). Making sense of teaching methods in computing education. *FIE'99 Frontiers in Education. 29th Annual Frontiers in Education Conference. Designing the Future of Science and Engineering Education. Conference Proceedings (IEEE Cat. No.99CH37011*, *1*, 11B3/30-11B3/35 vol.1.

Powers, K., Ecott, S., & Hirshfield, L. M. (2007). Through the looking glass: Teaching CS0 with Alice. *SIGCSE 2007: 38th SIGCSE Technical Symposium on Computer Science Education*, *1*, 213–217. https://doi.org/10.1145/1227310.1227386

Prensky, M. (2001). Digital natives, digital immigrants part 1. *On the horizon*, *9*(5), 1–6.

Prensky, M. (2005). Engage me or enrage me. *Educase Review*, *40*(5), 61–64.

Proulx, V. K., Raab, J., & Rasala, R. (2002). Objects from the beginning-with GUIs. *ACM SIGCSE Bulletin*, *34*(3), 65–69.

Psycharis, S., Makri-Botsari, E., & Xynogalas, G. (2008). The use of Educational Robotics for the teaching of Physics and its relation to self-esteem. *Proceedings of the TERECoP Workshop Teaching with robotics, Conference SIMPAR*, 132–142.

Pulver, K. (2019). Otter—A 2D C# Framework built on SFML 2. Preuzeto 30. kolovoz 2019., od Otter 2D website: http://otter2d.com/

Punia, S. K., Kumar, P., & Gupta, A. (2016). A Review of Software Quality Metrics for Object-Oriented Design. *International Journal of Advanced Research in Computer Science and Software Engineering*, *6*(8), 9.

Raadt, M., Watson, R., & Toleman, M. (2002). *Language Trends in Introductory Programming Courses*.

Ragkhitwetsagul, C., Krinke, J., & Clark, D. (2018). A comparison of code similarity analysers. *Empirical Software Engineering*, *23*(4), 2464–2519. https://doi.org/10.1007/s10664-017-9564-7

Ragonis, N., & Ben-Ari, M. (2005). On Understanding the Statics and Dynamics of Object-oriented Programs. *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, 226–230. https://doi.org/10.1145/1047344.1047425

Randolph, J. J. (2008). *Multidisciplinary Methods in Educational Technology Research and Development*.

Resnick, M. (1997). *Turtles, termites, and traffic jams: Explorations in massively parallel microworlds*. Mit Press.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., … Kafai, Y. (2009). Scratch: Programming for All. *Communications of the ACM*, *52*(11), 60–67. https://doi.org/10.1145/1592761.1592779

Richardson, K. (1998). *Models of Cognitive Development*. Psychology Press.

Robins, A. (2010). Learning edge momentum: A new account of outcomes in CS1. *Computer Science Education*, *20*(1), 37–71. https://doi.org/10.1080/08993401003612167

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and Teaching Programming: A Review and Discussion. *Computer Science Education*, *13*(2), 137–172. https://doi.org/10.1076/csed.13.2.137.14200

Rogalski, J., & Samurcay, R. (1990). Acquisition of Programming Knowledge and Skills. U *Psychology of Programming* (str. 9–19). Academic Press.

Rogers, C., & Portsmore, M. (2004). Bringing engineering to elementary school. *Journal of STEM Education: innovations and research*, *5*(3/4), 17.

Rosenfeld, M., & Rosenfeld, S. (1999). *Understanding the Surprise in PBL: An Exploration into the Learning Styles of Teachers and Their Students*. Predstavljeno na 8th Conference of EARLI (European Association for Research in Learning and Instruction), Goteburg, Sweden. Preuzeto od http://www.shermanrosenfeld.com/blog/2012/12/02/111/

Rossum, G. (1999). Computer programming for everybody. *Proposal to the Corporation for National Research Initiatives*.

Roy, C. K., Cordy, J. R., & Koschke, R. (2009). Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, *74*(7), 470–495. https://doi.org/10.1016/j.scico.2009.02.007

Sajaniemi, J., & Hu, C. (2006). *Teaching Programming: Going beyond "Objects First"* (Izd. Report A-2006-1; str. 10). UNIVERSITY OF JOENSUU, Department of Computer Science.

Samuel, M. S. (2017). An Insight into Programming Paradigms and Their Programming Languages. *Journal of Applied Technology and Innovation*, *1*(1), 37–57.

Sanders, K., & Thomas, L. (2007). Checklists for Grading Object-oriented CS1 Programs: Concepts and Misconceptions. *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, 166–170. https://doi.org/10.1145/1268784.1268834

Schmidt, H. G. (1983). Problem-based learning: Rationale and description. *Medical education*, *17*(1), 11–16.

Schreiber, J. B., Nora, A., Stage, F. K., Barlow, E. A., & King, J. (2006). Reporting Structural Equation Modeling and Confirmatory Factor Analysis Results: A Review. *The Journal of Educational Research*, *99*(6), 323–338. https://doi.org/10.3200/JOER.99.6.323-338

Schumacker, R. E., & Lomax, R. G. (2004). *A beginner's guide to structural equation modeling, 2nd ed.* Mahwah, NJ, US: Lawrence Erlbaum Associates Publishers.

Schunk, D. H. (2012). *Learning theories: An educational perspective* (6th ed). Boston: Pearson.

Seybert, H. (2011). Internet use in households and by individuals in 2011. *Eurostat statistics in focus*, *66*, 1–8.

Shabalina, O., Malliarakis, C., Tomos, F., & Mozelius, P. (2017). Game-based learning for learning to program: From learning through play to learning through game development. *11th European Conference on Games Based Learning 2017, Graz, Austria, 5-6 October 2017*, *11*, 571–576. Academic Conferences and Publishing International Limited.

Shaik, A. (2012). Object Oriented Software Metrics and Quality Assessment: Current State of the Art. *International Journal of Computer Applications*, *37*, 10.

Shreeve, M. W. (2008). Beyond the didactic classroom: Educational models to encourage active student involvement in learning. *Journal of Chiropractic Education*, *22*(1), 23–28.

Shriver, B. D. (1986). From the Editor in Chief: Software paradigms. *IEEE Software*, *3*(1), 2–2. https://doi.org/10.1109/MS.1986.232421

Simon, M. K., & Goes, J. (2013). Ex Post Facto Research. Preuzeto 07. veljača 2018., od Dissertation and Scholarly Research: Recipes for Success website: http://www.dissertationrecipes.com/wp-content/uploads/2011/04/Ex-Post-Facto-research.pdf

Singh, G. (2013). Metrics for measuring the quality of object-oriented software. *ACM SIGSOFT Software Engineering Notes*, *38*(5), 1. https://doi.org/10.1145/2507288.2507311

Slaughter, T. (2009). Creating a Successful Academic Climate for Urban Students. *Techniques: Connecting Education and Careers (J1)*, *84*(1), 16–19.

Smith, M. K. (2002). Malcolm Knowles, informal adult education, self-direction and andragogy. Preuzeto 19. kolovoz 2019., od The encyclopedia of informal education (INFED) website: http://infed.org/mobi/malcolm-knowles-informal-adult-education-self-direction-and-andragogy/

Smyth, P. (2018, ožujak 12). An Introduction to Programming Paradigms. Preuzeto 20. kolovoz 2019., od GC Digital Fellows website: https://digitalfellows.commons.gc.cuny.edu/2018/03/12/an-introduction-to-programming-paradigms/

Snyder, A. (1986). Encapsulation and Inheritance in Object-oriented Programming Languages. *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, 38–45. https://doi.org/10.1145/28697.28702

Soares, A., Fonseca, F., & Martin, N. L. (2015). TEACHING INTRODUCTORY PROGRAMMING WITH GAME DESIGN AND PROBLEM-BASED LEARNING. *Issues in Information Systems*, *16*(3), 128–137.

Sole, A. D. (2016). *Roslyn Succinctly*. Preuzeto od https://www.syncfusion.com/ebooks/roslyn/project-roslyn-the-net-compiler-platform

Spigariol, L. (2016). A pedagogical proposal for teaching object-oriented programming: Implementation through the educational software Wollok. *IEEE CACIDI 2016 - IEEE Conference on Computer Sciences*, 1–6. https://doi.org/10.1109/CACIDI.2016.7785976

Sposito, V. A., Hand, M. L., & Skarpness, B. (1983). On the efficiency of using the sample kurtosis in selecting optimal lpestimators. *Communications in Statistics-simulation and Computation*, *12*(3), 265–272.

Stefik, M., & Bobrow, D. G. (1985). Object-Oriented Programming: Themes and Variations. *AI Magazine*, *6*(4), 40. https://doi.org/10.1609/aimag.v6i4.508

Strauss, W., & Howe, N. (2000). Millennials rising: The next great generation. *New York: Vintage*.

Stroustrup, B. (1988). What is object-oriented programming? *IEEE Software*, *5*(3), 10–20. https://doi.org/10.1109/52.2020

Sturgis, P. (2019). Structural Equation Modelling (SEM): What it is and what it isn't. Preuzeto 01. studeni 2019., od National Centre for Research Methods online learning resource website: https://www.ncrm.ac.uk/resources/online/SEM2016/

Sweller, J. (1994). Cognitive load theory, learning difficulty, and instructional design. *Learning and Instruction*, *4*(4), 295–312. https://doi.org/10.1016/0959-4752(94)90003-5

Sweller, J., Ayres, P., & Kalyuga, S. (2011). *Cognitive Load Theory* (1. izd.). Springer-Verlag New York.

Sweller, J., van Merrienboer, J. J. G., & Paas, F. G. W. C. (1998). Cognitive Architecture and Instructional Design. *Educational psychology review*, *10*(3), 251–296. https://doi.org/10.1023/A:1022193728205

Taylor, C., Zingaro, D., Porter, L., Webb, K. C., Lee, C. B., & Clancy, M. (2014). Computer science concept inventories: Past and future. *Computer Science Education*, *24*(4), 253–276. https://doi.org/10.1080/08993408.2014.970779

Tchoshanov, M. A. (2013). *Engineering of Learning: Conceptualizing e-Didactics.* Moscow: UNESCO Institute for Information Technologies in Education.

Thomas, J. W. (2000). A Review of Research on Project-Based Learning. Preuzeto 01. rujan 2019., od http://www.bobpearlman.org/BestPractices/PBL_Research.pdf

TIOBE. (2019). TIOBE Index | TIOBE - The Software Quality Company. Preuzeto 20. kolovoz 2019., od TIOBE Index website: https://www.tiobe.com/tiobe-index/

Trigwell, K., Prosser, M., & Waterhouse, F. (1999). Relations between teachers' approaches to teaching and students' approaches to learning. *Higher education*, *37*(1), 57–70.

Tseng, K. H., Chang, C. C., Lou, S. J., & Chen, W. P. (2013). Attitudes towards science, technology, engineering and mathematics (STEM) in a project-based learning (PjBL) environment. *International Journal of Technology and Design Education*, *23*(1), 87–102. https://doi.org/10.1007/s10798-011-9160-x

Tucker, A. B. (2007). *Programming Languages: Principles and Paradigms*. McGraw-Hill Higher Education.

Turville, K., Meredith, G., & Smith, P. (2012). Understanding novice programmers: Their perceptions and motivations. *ASCILITE-Australian Society for Computers in Learning in Tertiary Education Annual Conference, 2012*.

Utting, I., Cooper, S., Kölling, M., Maloney, J., & Resnick, M. (2010). Alice, Greenfoot, and Scratch – A Discussion. *Trans. Comput. Educ.*, *10*(4), 1–11. https://doi.org/10.1145/1868358.1868364

van Merriënboer, J. J. G., & De Croock, M. B. M. (1992). Strategies for Computer-Based Programming Instruction: Program Completion vs. Program Generation. *Journal of Educational Computing Research*, *8*(3), 365–394. https://doi.org/10.2190/MJDX-9PP4-KFMT-09PM

van Merriënboer, J. J. G., & Sweller, J. (2005). Cognitive Load Theory and Complex Learning: Recent Developments and Future Directions. *Educational Psychology Review*, *17*(2), 147–177. https://doi.org/10.1007/s10648-005-3951-0

Van Merriënboer, J. J., & Kirschner, P. A. (2018). *Ten steps to complex learning: A systematic approach to four-component instructional design* (3. izd.). New York, NY, USA: Routledge, Taylor & Francis.

Van Roy, Peter. (2009). Programming Paradigms for Dummies: What Every Programmer Should Know. *New Computational Paradigms for Computer Music*, 9–47.

Vihavainen, A., Airaksinen, J., & Watson, C. (2014). A Systematic Review of Approaches for Teaching Introductory Programming and Their Influence on Success. *Proceedings of the Tenth Annual Conference on International Computing Education Research*, 19–26. https://doi.org/10.1145/2632320.2632349

Vilner, T., Zur, E., & Gal-Ezer, J. (2007). Fundamental Concepts of CS1: Procedural vs. Object Oriented Paradigm—A Case Study. *SIGCSE Bull.*, *39*(3), 171–175. https://doi.org/10.1145/1269900.1268835

Wang, A. I., & Wu, B. (2009). An Application of a Game Development Framework in Higher Education. *International Journal of Computer Games Technology*, *2009*, 1–12. https://doi.org/10.1155/2009/693267

Watson, C., & Li, F. W. B. (2014). *Failure rates in introductory programming revisited*. 39–44. https://doi.org/10.1145/2591708.2591749

Weintrop, D. (2015). Blocks, text, and the space between: The role of representations in novice programming environments. *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 301–302. https://doi.org/10.1109/VLHCC.2015.7357237

Weintrop, D., & Wilensky, U. (2015). To Block or Not to Block, That is the Question: Students' Perceptions of Blocks-based Programming. *Proceedings of the 14th International Conference on Interaction Design and Children*, 199–208. https://doi.org/10.1145/2771839.2771860

Werner, L., Denner, J., Bliesner, M., & Rex, P. (2009). Can Middle-schoolers Use Storytelling Alice to Make Games?: Results of a Pilot Study. *Proceedings of the 4th International Conference on Foundations of Digital Games*, 207–214. https://doi.org/10.1145/1536513.1536552

Wetzel, A. (2011). *Factor Analysis Methods and Validity Evidence: A Systematic Review of Instrument Development Across the Continuum of Medical Education* (PhD Thesis). Virginia Commonwealth University, Richmond.

William F. McComas, W. F. M. (eds.). (2014). *The Language of Science Education: An Expanded Glossary of Key Terms and Concepts in Science Teaching and Learning*. SensePublishers.

Williams, K. (2003). Literacy and Computer Literacy: Analyzing the NRC's "Being Fluent with Information Technology". *Journal of Literacy and Technology*, *3*(1), 1–20.

Williams, M. K. (2017). John Dewey in the 21st Century. *Journal of Inquiry & Action in Education*, *9*(1), 91–102.

Winslow, L. E. (1996). Programming Pedagogy –A Psychological Overview. *ACM SIGCSE Bulletin*, *28*(3), 17–22. https://doi.org/10.1145/234867.234872

Wong, Y. S., Yatim, M. H. M., & Tan, W. H. (2014). Use computer game to learn Object-Oriented programming in computer science courses. *2014 IEEE Global Engineering Education Conference (EDUCON)*, 9–16. https://doi.org/10.1109/EDUCON.2014.6826059

Xinogalos, S. (2009). A proposal for teaching Object-Oriented Programming to undergraduate students. *International Journal of Teaching and Case Studies*, *2*(1), 41–55.

Xinogalos, S. (2015). Object-Oriented Design and Programming: An Investigation of Novices' Conceptions on Objects and Classes. *Trans. Comput. Educ.*, *15*(3), 13:1–13:21. https://doi.org/10.1145/2700519

Yan, L. (2009). Teaching object-oriented programming with games. *ITNG 2009 - 6th International Conference on Information Technology: New Generations*, 969–974. https://doi.org/10.1109/ITNG.2009.13

Yang, T.-C., Hwang, G.-J., Yang, S. J. H., & Hwang, G.-H. (2015). A Two-Tier Test-based Approach to Improving Students' Computer-Programming Skills in a Web-Based Learning Environment. *Educational Technology & Society*, *18*(1), 198–210.

Ziegler, U., & Crews, T. (1999). An integrated program development tool for teaching and learning how to program. *The Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education - SIGCSE '99*, 276–280. https://doi.org/10.1145/299649.299786

PRILOZI

Popis priloga

Prilog 1 – Analiza projekata za ak. god. 2013/14 i 2014/15

Rb.	Naziv projekta	Tip igre	Br. Klasa	Korištene strukture	Nasljeđivanje	Ugnježđivanje	Preopterećene metode	Premošćivanje	Događaji I delegati	Komentari vlastitoga koda	Br. Nadogradnji	Br. Promjena - Prilagodbi	Br. Nadogradnji	okvir za izradu igre
1	AlfonsRPG	RPG	19	NE	16	0	3	23	0	3	17	1	18	WFA - Game SDK
2	Brick Breaker	Arkadna igra	2	DA	1	1	4	0	0	1	1	1	2	WFA - Game SDK
3	Tower Defense	Strategy	6	DA	2	1	5	1	0	3	5	1	6	WFA - Game SDK
4	Dr!ve	2D simulacija vožnje	4	DA	2	1	6	1	0	3	4	1	5	WFA - Game SDK
5	RACe	2D simulacija vožnje	1	DA	0	1	4	0	0	1	0	1	1	WFA - Game SDK
6	Super Mario	Platformska igra	2	DA	0	1	5	0	0	2	1	1	2	WFA - Game SDK
7	Tank Wars	Strategy/shooter	5	DA	3	1	7	0	0	3	4	1	5	WFA - Game SDK
8	Wizzard battle	RPG	11	DA	7	1	5	0	0	3	10	1	11	WFA - Game SDK
9	Donkey Mario	Platformska igra	10	NE	7	0	2	9	0	1	10	0	10	Microsoft.Xna.Framework
10	Space Invaders	Shooting game	6	NE	1	0	0	8	0	2	6	0	6	Microsoft.Xna.Framework
11	Samurai	RPG	6	NE	4	0	0	4	0	2	6	0	6	Unity 3D
12	Poludjele sapunice	Casual game	5	NE	0	0	0	0	0	1	5	0	5	Unity 2D
13		Casual game	5	NE	0	0	0	0	0	1	5	0	5	Unity 2D
14	Šah	Thinking	9	DA	6	0	0	12	0	2	9	0	9	Samo WFA
15	PMF igra												0	
16	Pacman s interneta	Pacman	17	DA	13	0	0	2	3	1	0	0	0	Samo WFA
17		Pacman	18	DA	13	0	0	2	3	1	0	0	0	Samo WFA
18		Pacman	19	DA	13	0	0	2	3	1	0	0	0	Samo WFA
19	Snake	Snake	4	DA	0	0	0	0	0	1	6	0	6	Samo WFA
20		Snake	4	DA	0	0	0	0	0	1	6	0	6	Samo WFA
21	Snake 2	Snake	4	DA	0	0	0	0	0	1	5	0	5	Samo WFA
22	Labirint miš	Maze	7	DA	3	0	0	0	0	1	6	1	7	Samo WFA
23	Puzzle slike	Puzzle	3	DA	0	0	0	0	0	1	3	0	3	Samo WFA
24		Puzzle	4	DA	0	0	0	0	0	1	3	0	3	Samo WFA
25	Color lines												0	
26	Brick		5	DA	3	0	0	10	0	1	4	1	5	Microsoft.Xna.Framework
27	Kockice	Luck	11	DA	1	0	0	0	0	1	11	0	11	Samo WFA
28	Minolovac	Thinking	10	DA	3	0	2	0	0	2	10	0	10	Samo WFA
29	Labirint	Maze	6	DA	4	0	0	3	0	1	6	0	6	Samo WFA
30		Maze	7	DA	4	0	0	3	0	1	6	0	6	Samo WFA
31	Sudoku	Puzzle	3	DA	1	0	0	0	0	4	3	0	3	Samo WFA
32	Memory	Memory	6	DA	2	0	0	0	0	1	6	0	6	Samo WFA
33	Memory cars	Memory	6	DA	2	0	0	0	0	1	6	0	6	Samo WFA
34	Puzzle 8 blokova												0	
35	Trešeta	Cards	3	DA	0	0	1	0	0	2	3	0	3	Samo WFA
36	Kviz	Quiz	9	DA	3	0	2	0	0	1	9	0	9	Samo WFA
37	Tic tac toe	Puzzle	4	NE	2	0	0	2	0	2	4	0	4	Samo WFA
38	Ultimate tic tac toe	Puzzle	3	DA	1	0	0	0	0	1	3	0	3	Samo WFA
39	Tica												0	
40	White tiles	Refleksne igre	5	0	2	0	1	0	0	1	4	1	5	Samo WFA
41		Refleksne igre	5	0	2	0	1	0	0	1	4	1	5	Samo WFA

Ručna analiza projekata

Rb	Spol	Tip projekta	Preuzet gotov projekt	Okvir	Godina
1	Ž	Nedefinirano			2013 - 2014
2	Ž	Nije odabran projekt			2013 - 2014
3	M	Rezerviranje i prodaja			2013 - 2014
4	Ž	Nije odabran projekt			2013 - 2014
5	Ž	Rezerviranje i prodaja			2013 - 2014
6	Ž	Rezerviranje i prodaja			2013 - 2014
7	Ž	Rezerviranje i prodaja			2013 - 2014
8	Ž	Organizacija			2013 - 2014
9	Ž	Nije odabran projekt			2013 - 2014
10	Ž	Rezerviranje i prodaja			2013 - 2014
11	M	Rezerviranje i prodaja			2013 - 2014
12	M	Rezerviranje i prodaja			2013 - 2014
13	Ž	Nije odabran projekt			2013 - 2014
14	Ž	Rezerviranje i prodaja			2013 - 2014
15	Ž	Rezerviranje i prodaja			2013 - 2014
16	M	Nije odabran projekt			2013 - 2014
17	M	Igra	NE	WFA - Game SDK	2013 - 2014
18	M	Igra	NE		2013 - 2014
19	Ž	Rezerviranje i prodaja			2013 - 2014
20	Ž	Rezerviranje i prodaja			2013 - 2014
21	Ž	Izračuni			2013 - 2014
22	Ž	Nije odabran projekt			2013 - 2014
23	Ž	Rezerviranje i prodaja			2013 - 2014
24	Ž	Nije odabran projekt			2013 - 2014
25	M	Igra	NE	WFA - Game SDK	2013 - 2014
26	Ž	Rezerviranje i prodaja			2013 - 2014
27	Ž	Rezerviranje i prodaja			2013 - 2014
28	M	Igra	NE		2013 - 2014
29	M	Rezerviranje i prodaja			2013 - 2014
30	Ž	Rezerviranje i prodaja			2013 - 2014
31	Ž	Nije odabran projekt			2013 - 2014
32	M	Igra	NE	WFA - Game SDK	2013 - 2014
33	Ž	Rezerviranje i prodaja			2013 - 2014
34	M	Izračuni			2013 - 2014
35	M	Nije odabran projekt			2013 - 2014
36	Ž	Izračuni			2013 - 2014
37	Ž	Nije odabran projekt			2013 - 2014
38	M	Nije odabran projekt			2013 - 2014
39	Ž	Rezerviranje i prodaja			2013 - 2014
40	M	Rezerviranje i prodaja			2013 - 2014
41	Ž	Rezerviranje i prodaja			2013 - 2014

Rb	Spol	Tip projekta	Preuzet gotov projekt	Okvir	Godina
42	M	Igra	NE	WFA - Game SDK	2013 - 2014
43	M	Rezerviranje i prodaja			2013 - 2014
44	Ž	Nije odabran projekt			2013 - 2014
45	Ž	Izračuni			2013 - 2014
46	Ž	Rezerviranje i prodaja			2013 - 2014
47	Ž	Nije odabran projekt			2013 - 2014
48	M	Nije odabran projekt			2013 - 2014
49	Ž	Nije odabran projekt			2013 - 2014
50	Ž	Rezerviranje i prodaja			2013 - 2014
51	Ž	Rezerviranje i prodaja			2013 - 2014
52	M	Izračuni			2013 - 2014
53	Ž	Rezerviranje i prodaja			2013 - 2014
54	Ž	Rezerviranje i prodaja			2013 - 2014
55	M	Igra	NE	WFA - Game SDK	2013 - 2014
56	Ž	Rezerviranje i prodaja			2013 - 2014
57	M	Rezerviranje i prodaja			2013 - 2014
58	M	Rezerviranje i prodaja			2013 - 2014
59	M	Rezerviranje i prodaja			2013 - 2014
60	M	Nije odabran projekt			2013 - 2014
61	Ž	Rezerviranje i prodaja			2013 - 2014
62	M	Rezerviranje i prodaja			2013 - 2014
63	Ž	Nije odabran projekt			2013 - 2014
64	Ž	Rezerviranje i prodaja			2013 - 2014
65	M	Igra	NE	WFA - Game SDK	2013 - 2014
66	M	Simulacija			2013 - 2014
67	M	Nije odabran projekt			2013 - 2014
68	M	Organizacija			2013 - 2014
69	Ž	Rezerviranje i prodaja			2013 - 2014
70	M	Igra	NE	WFA - Game SDK	2013 - 2014
71	Ž	Rezerviranje i prodaja			2013 - 2014
1	Ž	Rezerviranje i prodaja			2014 - 2015
2	M	Igra	NE	Unity 3D	2014 - 2015
3	Ž	Rezerviranje i prodaja			2014 - 2015
4	Ž	Igra	DA		2014 - 2015
5	Ž	Igra	DA		2014 - 2015
6	Ž	Igra	NE	Microsoft.Xna.Framework	2014 - 2015
7	Ž	Organizacija			2014 - 2015
8	Ž	Rezerviranje i prodaja			2014 - 2015
9	M	Nije odabran projekt			2014 - 2015
10	Ž	Igra	DA		2014 - 2015
11	M	Rezerviranje i prodaja			2014 - 2015
12	Ž	Rezerviranje i prodaja			2014 - 2015
13	Ž	Igra	DA		2014 - 2015
14	M	Igra	NE	WFA - Game SDK	2014 - 2015
15	M	Nije odabran projekt			2014 - 2015

Rb	Spol	Tip projekta	Preuzet gotov projekt	Okvir	Godina
16	Ž	Igra	NE		2014 - 2015
17	Ž	Simulacija			2014 - 2015
18	M	Nije odabran projekt			2014 - 2015
19	M	Igra	NE	Microsoft.Xna.Framework	2014 - 2015
20	M	Rezerviranje i prodaja			2014 - 2015
21	Ž	Igra	DA		2014 - 2015
22	Ž	Igra	DA	Microsoft.Xna.Framework	2014 - 2015
23	Ž	Igra	DA		2014 - 2015
24	Ž	Igra	DA		2014 - 2015
25	Ž	Igra	DA		2014 - 2015
26	Ž	Organizacija			2014 - 2015
27	Ž	Rezerviranje i prodaja			2014 - 2015
28	M	Nije odabran projekt			2014 - 2015
29	Ž	Igra	NE		2014 - 2015
30	M	Igra	NE		2014 - 2015
31	Ž	Edukacijski kviz			2014 - 2015
32	M	Nije odabran projekt			2014 - 2015
33	M	Nije odabran projekt			2014 - 2015
34	Ž	Izračuni			2014 - 2015
35	Ž	Igra	DA		2014 - 2015
36	M	Igra	DA		2014 - 2015
37	Ž	Rezerviranje i prodaja			2014 - 2015
38	M	Nije odabran projekt			2014 - 2015
39	M	Rezerviranje i prodaja			2014 - 2015
40	M	Igra	NE	Microsoft.Xna.Framework	2014 - 2015
41	M	Igra	NE	Unity 2D	2014 - 2015
42	Ž	Igra	DA		2014 - 2015
43	Ž	Igra	DA		2014 - 2015
44	Ž	Igra	NE		2014 - 2015
45	M	Nije odabran projekt			2014 - 2015
46	Ž	Igra	NE		2014 - 2015
47	Ž	Nije odabran projekt			2014 - 2015
48	Ž	Rezerviranje i prodaja			2014 - 2015
49	Ž	Nije odabran projekt			2014 - 2015
50	M	Rezerviranje i prodaja			2014 - 2015
51	Ž	Nije odabran projekt			2014 - 2015
52	Ž	Nije odabran projekt			2014 - 2015
53	Ž	Nedefinirano			2014 - 2015
54	M	Nije odabran projekt			2014 - 2015
55	Ž	Igra	NE		2014 - 2015
56	M	Igra	NE	Unity 2D	2014 - 2015
57	Ž	Nije odabran projekt			2014 - 2015
58	M	Nije odabran projekt			2014 - 2015
59	M	Nedefinirano			2014 - 2015
60	M	Igra	NE		2014 - 2015

Rb	Spol	Tip projekta	Preuzet gotov projekt	Okvir	Godina
61	Ž	Nije odabran projekt			2014 - 2015
62	M	Rezerviranje i prodaja			2014 - 2015
63	Ž	Nije odabran projekt			2014 - 2015
64	Ž	Rezerviranje i prodaja			2014 - 2015
65	Ž	Igra	DA		2014 - 2015
66	Ž	Igra	DA		2014 - 2015
67	M	Igra	NE		2014 - 2015
68	Ž	Rezerviranje i prodaja			2014 - 2015
69	Ž	Igra	DA		2014 - 2015
70	Ž	Igra	DA		2014 - 2015
71	Ž	Rezerviranje i prodaja			2014 - 2015
72	M	Rezerviranje i prodaja			2014 - 2015
73	Ž	Nedefinirano			2014 - 2015

Prilog 2 – Statistička analiza podataka

Usporedba 1. i 2. generacije

Test Statistics[a]

	GUI	abstract	protected	staticClass	inherits	maxDepthOfInheritance	virtual	overrideToString	overrideOtherMethods	overload	notGuiClass	simpleGetSet	complexGetSet	nonEmptyConstructor	eventHandlerMethods	otherMethods	baseConstructorCall	staticNodeInNonStaticClass	REGR factor score 1 for analysis 1	REGR factor score 2 for analysis 1
Mann-Whitney U	674,500	732,000	825,500	859,000	812,000	735,000	845,000	530,500	842,500	748,500	813,500	581,500	754,500	727,500	620,000	792,500	856,000	727,000	748,000	494,000
Wilcoxon W	1494,500	1552,000	1771,500	1805,000	1632,000	1555,000	1791,000	1350,500	1662,500	1568,500	1633,500	1401,500	1574,500	1547,500	1440,000	1738,500	1676,000	1673,000	1568,000	1314,000
Z	-1,737	-1,309	-0,497	-0,034	-0,446	-1,318	-0,224	-3,312	-0,182	-1,516	-0,433	-2,544	-1,102	-1,219	-2,189	-0,617	-0,040	-1,592	-1,021	-3,336
Asymp. Sig. (2-tailed)	0,082	0,190	0,619	0,973	0,656	0,187	0,823	0,001	0,855	0,130	0,665	0,011	0,271	0,223	0,029	0,537	0,968	0,111	0,307	0,001

a. Grouping Variable: generation

Usporedba 1. i 3. generacije

Test Statistics[a]

	GUI	abstract	protected	staticClass	inherits	maxDepthOfInheritance	virtual	overrideToString	overrideOtherMethods	overload	notGuiClass	simpleGetSet	complexGetSet	nonEmptyConstructor	eventHandlerMethods	otherMethods	baseConstructorCall	staticNodeInNonStaticClass	REGR factor score 1 for analysis 1	REGR factor score 2 for analysis 1
Mann-Whitney U	611,500	1280,000	1117,000	1262,500	1004,000	688,000	1239,500	545,500	1400,000	1286,000	972,000	458,000	1249,000	890,000	443,500	730,000	704,500	203,500	1250,000	161,000
Wilcoxon W	2822,500	2226,000	2063,000	2208,500	1950,000	1634,000	2185,500	2756,500	2346,000	2232,000	1918,000	2669,000	2195,000	3101,000	2654,500	1676,000	1650,500	1149,500	2196,000	2372,000
Z	-5,457	-0,920	-2,416	-1,939	-2,603	-4,855	-1,568	-6,717	-0,132	-0,996	-2,799	-5,972	-1,131	-3,304	-6,060	-4,278	-4,526	-7,760	-1,048	-7,800
Asymp. Sig. (2-tailed)	0,000	0,358	0,016	0,052	0,009	0,000	0,117	0,000	0,895	0,319	0,005	0,000	0,258	0,001	0,000	0,000	0,000	0,000	0,295	0,000

a. Grouping Variable: generation

Usporedba 1. i 4. generacije

Test Statistics[a]

	GUI	abstract	protected	staticClass	inherits	maxDepthOfInheritance	virtual	overrideToString	overrideOtherMethods	overload	notGuiClass	simpleGetSet	complexGetSet	nonEmptyConstructor	eventHandlerMethods	otherMethods	baseConstructorCall	staticNodeIsInNonStaticClass	REGR factor score 1 for analysis 1	REGR factor score 2 for analysis 1
Mann-Whitney U	597,000	1028,000	734,500	1063,500	502,000	291,500	942,500	392,000	993,000	1131,500	706,500	694,500	867,500	1193,000	195,500	689,500	313,500	830,500	688,000	124,000
Wilcoxon W	2193,000	1974,000	1680,500	2009,500	1448,000	1237,500	1888,500	1988,000	1939,000	2077,500	1652,500	2290,500	1813,500	2139,000	1791,500	1635,500	1259,500	1776,500	1634,000	1720,000
Z	-4,545	-1,328	-3,896	-1,990	-5,009	-6,926	-2,432	-7,183	-1,591	-0,631	-3,538	-3,606	-2,502	-0,078	-7,152	-3,638	-6,367	-3,125	-3,643	-7,624
Asymp. Sig. (2-tailed)	0,000	0,184	0,000	0,047	0,000	0,000	0,015	0,000	0,112	0,528	0,000	0,000	0,012	0,938	0,000	0,000	0,000	0,002	0,000	0,000

a. Grouping Variable: generation

Usporedba 1. i 5. generacije

Test Statistics[a]

	GUI	abstract	protected	staticClass	inherits	maxDepthOfInheritance	virtual	overrideToString	overrideOtherMethods	overload	notGuiClass	simpleGetSet	complexGetSet	nonEmptyConstructor	eventHandlerMethods	otherMethods	baseConstructorCall	staticNodeIsInNonStaticClass	REGR factor score 1 for analysis 1	REGR factor score 2 for analysis 1
Mann-Whitney U	743,000	1412,500	855,000	1361,000	692,500	264,000	1741,000	633,000	1425,500	1335,000	911,500	1180,500	1060,500	1612,500	236,500	890,000	415,000	1195,500	873,000	176,000
Wilcoxon W	4064,000	2358,500	1801,000	2307,000	1638,500	1210,000	5062,000	3954,000	2371,500	2281,000	1857,500	4501,500	2006,500	4933,500	3557,500	1836,000	1361,000	2141,500	1819,000	3497,000
Z	-5,819	-1,844	-5,101	-3,068	-5,556	-8,673	-0,004	-7,608	-1,766	-2,370	-4,387	-2,951	-3,697	-0,683	-7,988	-4,475	-7,025	-3,323	-4,560	-8,219
Asymp. Sig. (2-tailed)	0,000	0,065	0,000	0,002	0,000	0,000	0,996	0,000	0,077	0,018	0,000	0,003	0,000	0,494	0,000	0,000	0,000	0,001	0,000	0,000

a. Grouping Variable: generation

Usporedba 2. i 3. generacije

Test Statistics[a]

	GUI	abstract	protected	staticClass	inherits	maxDepthOfInheritance	virtual	overrideToString	overrideOtherMethods	overload	notGuiClass	simpleGetSet	complexGetSet	nonEmptyConstructor	eventHandlerMethods	otherMethods	baseConstructorCall	staticNodeIsInNonStaticClass	REGR factor score 1 for analysis 1	REGR factor score 2 for analysis 1
Mann-Whitney U	862,000	1003,500	1101,500	1172,500	773,500	520,500	1184,000	1004,000	1284,000	988,000	726,500	976,000	995,000	980,500	672,500	670,000	602,500	403,000	1066,000	483,000
Wilcoxon W	3073,000	1823,500	1921,500	1992,500	1593,500	1340,500	2004,000	3215,000	2104,000	1808,000	1546,500	3187,000	1815,000	3191,500	2883,500	1490,000	1422,500	1223,000	1886,000	2694,000
Z	-3,312	-2,245	-1,797	-1,898	-3,604	-5,532	-1,234	-3,313	-0,266	-2,729	-3,920	-2,253	-2,308	-2,231	-4,233	-4,243	-4,771	-6,090	-1,656	-5,455
Asymp. Sig. (2-tailed)	0,001	0,025	0,072	0,058	0,000	0,000	0,217	0,001	0,790	0,006	0,000	0,024	0,021	0,026	0,000	0,000	0,000	0,000	0,098	0,000

a. Grouping Variable: generation

Usporedba 2. i 4. generacije

Test Statistics[a]

	GUI	abstract	protected	staticClass	inherits	maxDepthOfInheritance	virtual	overrideToString	overrideOtherMethods	overload	notGuiClass	simpleGetSet	complexGetSet	nonEmptyConstructor	eventHandlerMethods	otherMethods	baseConstructorCall	staticNodeIsInNonStaticClass	REGR factor score 1 for analysis 1	REGR factor score 2 for analysis 1
Mann-Whitney U	820,000	734,000	736,500	987,500	354,500	194,500	908,000	784,000	939,500	884,000	493,500	1072,000	683,000	934,500	340,000	634,500	271,500	958,000	597,000	513,000
Wilcoxon W	2416,000	1554,000	1556,500	1807,500	1174,500	1014,500	1728,000	2380,000	1759,500	1704,000	1313,500	1892,000	1503,000	1754,500	1936,000	1454,500	1091,500	1778,000	1417,000	2109,000
Z	-2,394	-3,104	-3,291	-1,949	-5,740	-7,324	-2,050	-4,352	-1,444	-2,284	-4,699	-0,358	-3,459	-1,390	-5,835	-3,614	-6,374	-1,360	-3,887	-4,511
Asymp. Sig. (2-tailed)	0,017	0,002	0,001	0,051	0,000	0,000	0,040	0,000	0,149	0,022	0,000	0,721	0,001	0,165	0,000	0,000	0,000	0,174	0,000	0,000

a. Grouping Variable: generation

Usporedba 2. i 5. generacije

Test Statistics[a]

	GUI	abstract	protected	staticClass	inherits	maxDepthOfInheritance	virtual	overrideToString	overrideOtherMethods	overload	notGuiClass	simpleGetSet	complexGetSet	nonEmptyConstructor	eventHandlerMethods	otherMethods	baseConstructorCall	staticNodeIsInNonStaticClass	REGR factor score 1 for analysis 1	REGR factor score 2 for analysis 1
Mann-Whitney U	1060,000	999,000	875,000	1259,500	438,000	153,000	1591,000	1195,500	1353,500	955,000	615,000	1458,500	809,500	1504,500	405,500	814,500	350,000	1380,000	792,000	689,000
Wilcoxon W	4381,000	1819,000	1695,000	2079,500	1258,000	973,000	4912,000	4516,500	2173,500	1775,000	1435,000	2278,500	1629,500	2324,500	3726,500	1634,500	1170,000	2200,000	1612,000	4010,000
Z	-3,490	-3,678	-4,454	-3,023	-6,565	-8,987	-0,265	-4,091	-1,573	-4,152	-5,579	-0,891	-4,629	-0,642	-6,790	-4,445	-7,052	-1,480	-4,562	-5,130
Asymp. Sig. (2-tailed)	0,000	0,000	0,000	0,003	0,000	0,000	0,791	0,000	0,116	0,000	0,000	0,373	0,000	0,521	0,000	0,000	0,000	0,139	0,000	0,000

a. Grouping Variable: generation

Usporedba 3. i 4. generacije

Test Statistics[a]

	GUI	abstract	protected	staticClass	inherits	maxDepthOfInheritance	virtual	overrideToString	overrideOtherMethods	overload	notGuiClass	simpleGetSet	complexGetSet	nonEmptyConstructor	eventHandlerMethods	otherMethods	baseConstructorCall	staticNodeIsInNonStaticClass	REGR factor score 1 for analysis 1	REGR factor score 2 for analysis 1
Mann-Whitney U	1581,000	1752,500	1498,500	1835,500	1099,500	1421,000	1679,500	1736,000	1577,000	1732,000	1517,000	1120,000	1524,000	1169,500	1389,500	1708,500	1161,000	641,000	1256,000	1238,000
Wilcoxon W	3792,000	3963,500	3709,500	4046,500	3310,500	3632,000	3890,500	3332,000	3788,000	3328,000	3728,000	3331,000	3735,000	3380,500	2985,500	3304,500	3372,000	2237,000	3467,000	3449,000
Z	-1,641	-0,521	-2,008	-0,107	-3,885	-2,480	-1,081	-1,865	-1,494	-0,694	-1,723	-3,753	-1,733	-3,510	-2,382	-0,718	-3,562	-6,316	-3,042	-3,134
Asymp. Sig. (2-tailed)	0,101	0,602	0,045	0,915	0,000	0,013	0,280	0,062	0,135	0,488	0,085	0,000	0,083	0,000	0,017	0,473	0,000	0,000	0,002	0,002

a. Grouping Variable: generation

Usporedba 3. i 5. generacije

Test Statistics[a]

	GUI	abstract	protected	staticClass	inherits	maxDepthOfInheritance	virtual	overrideToString	overrideOtherMethods	overload	notGuiClass	simpleGetSet	complexGetSet	nonEmptyConstructor	eventHandlerMethods	otherMethods	baseConstructorCall	staticNodeIsInNonStaticClass	REGR factor score 1 for analysis 1	REGR factor score 2 for analysis 1
Mann-Whitney U	2600,000	2425,500	1827,000	2378,500	1618,500	1838,000	2340,500	2610,500	2259,000	2281,000	2039,500	1397,000	1912,000	1834,500	1777,500	2638,000	1665,500	989,500	1645,000	2063,000
Wilcoxon W	4811,000	4636,500	4038,000	4589,500	3829,500	4049,000	5661,500	5931,500	4470,000	4492,000	4250,500	3608,000	4123,000	4045,500	5098,500	4849,000	3876,500	4310,500	3856,000	4274,000
Z	-0,361	-1,024	-3,546	-1,660	-4,151	-3,930	-1,897	-0,660	-1,723	-1,671	-2,494	-4,979	-3,052	-3,291	-3,551	-0,137	-3,964	-6,714	-4,004	-2,376
Asymp. Sig. (2-tailed)	0,718	0,306	0,000	0,097	0,000	0,000	0,058	0,509	0,085	0,095	0,013	0,000	0,002	0,001	0,000	0,891	0,000	0,000	0,000	0,018

a. Grouping Variable: generation

Usporedba 4. i 5. generacije

Test Statistics[a]

	GUI	abstract	protected	staticClass	inherits	maxDepthOfInheritance	virtual	overrideToString	overrideOtherMethods	overload	notGuiClass	simpleGetSet	complexGetSet	nonEmptyConstructor	eventHandlerMethods	otherMethods	baseConstructorCall	staticNodeIsInNonStaticClass	REGR factor score 1 for analysis 1	REGR factor score 2 for analysis 1
Mann-Whitney U	1980,000	2141,000	1995,000	2034,000	2182,000	2086,500	1786,000	2184,000	2217,000	1759,500	2132,000	1944,000	2060,500	2075,000	2070,500	2046,000	2247,000	2249,000	2214,000	1888,000
Wilcoxon W	5301,000	3737,000	3591,000	3630,000	5503,000	3682,500	5107,000	3780,000	3813,000	3355,500	3728,000	3540,000	3656,500	5396,000	5391,500	3642,000	5568,000	3845,000	3810,000	5209,000
Z	-1,533	-0,619	-1,254	-1,463	-0,381	-1,064	-2,927	-1,451	-0,234	-2,456	-0,602	-1,422	-0,929	-0,852	-0,891	-0,974	-0,093	-0,091	-0,236	-1,664
Asymp. Sig. (2-tailed)	0,125	0,536	0,210	0,143	0,703	0,287	0,003	0,147	0,815	0,014	0,547	0,155	0,353	0,394	0,373	0,330	0,926	0,928	0,813	0,096

a. Grouping Variable: generation

Prilog 3 – podaci

projekt	GUI	abstract	protected	statička klasa	inherit	maks. dubina	virtual	override ToString	override ostalo	overload	klase osim GUI	get/set	složeni get/set	neprazan konstr.	otter	event handlers	ostale metode	poziv base konstr.	statički el. van static klase	generacija
P1	2	0	0	0	0	0	0	1	0	0	1	12	0	3	F	8	3	0	0	1
P2	3	0	0	0	1	0	0	1	0	0	2	14	0	4	F	18	4	1	1	1
P3	3	1	0	0	1	1	0	0	0	2	2	26	0	3	F	12	5	0	0	1
P4	2	0	0	0	1	0	0	1	0	0	2	11	1	2	F	6	2	1	1	1
P5	5	1	1	0	2	1	1	2	1	3	3	12	2	8	F	69	15	0	0	1
P6	2	2	0	0	4	1	0	4	4	0	5	15	1	3	F	9	12	0	0	1
P7	6	2	0	0	1	1	0	1	1	0	2	26	0	8	F	15	16	1	0	1
P8	1	0	0	0	4	2	1	0	1	4	7	18	0	8	T	4	35	0	1	1
P9	4	2	0	0	7	1	0	1	7	0	8	5	1	5	F	37	12	0	0	1
P10	4	2	0	0	5	1	0	0	2	0	8	19	3	8	F	8	5	5	0	1
P11	1	1	0	2	10	1	0	0	0	0	14	0	0	3	T	22	15	10	0	1
P12	6	0	0	0	0	0	0	0	0	0	1	5	1	6	F	28	0	0	0	1
P13	1	3	2	0	8	2	0	0	0	12	10	40	0	10	T	4	37	5	0	1
P14	4	1	0	0	15	1	1	0	2	2	17	6	2	12	F	24	23	14	0	1
P15	3	2	0	0	4	1	0	0	1	0	5	10	1	5	F	28	4	4	3	1
P16	2	0	0	0	0	0	0	2	0	0	2	18	0	2	F	14	4	0	0	1
P17	4	0	0	0	4	1	0	1	0	3	6	32	2	4	F	22	10	1	1	1
P18	4	1	0	0	1	1	0	1	0	0	2	16	2	6	F	37	2	1	0	1
P19	3	0	0	0	1	1	0	1	0	0	2	16	0	2	T	8	6	0	0	1
P20	4	2	0	0	4	1	0	0	4	0	5	10	2	5	F	20	5	4	0	1
P21	1	0	0	0	4	1	0	0	0	9	4	7	4	8	F	10	53	4	0	1
P22	5	1	1	0	2	1	1	2	1	3	3	12	2	8	F	69	15	0	0	1
P23	3	0	0	0	1	0	0	1	0	11	1	10	0	7	F	21	57	0	0	1

projekt	GUI	abstract	protected	statička klasa	inherit	maks. dubina	virtual	override ToString	override ostalo	overload	klase osim GUI	get/set	složeni get/set	neprazan konstr.	otter	event handlers	ostale metode	poziv base konstr.	statički el. van static klase	generacija
P24	1	1	1	0	4	1	0	1	0	0	6	26	2	2	F	52	2	0	0	1
P25	3	0	0	0	1	0	0	1	0	2	2	24	0	6	F	12	6	0	0	1
P26	3	0	0	0	0	0	0	2	0	0	2	16	0	5	F	14	9	0	0	1
P27	3	0	0	0	1	1	0	0	0	0	3	28	2	5	F	23	0	0	0	1
P28	4	0	0	0	0	0	0	2	0	0	2	30	0	7	F	17	6	0	0	1
P29	1	0	0	0	1	1	1	1	1	0	2	10	0	1	F	16	5	0	0	1
P30	1	0	0	0	1	1	0	0	0	0	2	10	0	2	T	3	1	1	0	1
P31	6	0	0	0	1	1	0	2	0	0	3	2	0	9	F	37	10	0	0	1
P32	1	2	0	0	3	1	0	1	3	0	4	10	0	2	F	10	9	3	0	1
P33	1	0	0	0	0	0	0	0	1	0	1	16	0	2	T	1	4	0	0	1
P34	2	1	0	0	1	1	0	0	0	0	2	22	0	3	F	13	3	0	0	1
P35	3	0	0	0	1	0	0	1	0	0	3	16	2	5	F	27	6	1	1	1
P36	3	2	0	0	2	1	0	1	1	0	3	18	0	5	F	18	5	2	1	1
P37	3	0	0	0	2	2	0	1	2	0	4	20	4	3	F	20	4	0	0	1
P38	2	0	0	0	0	0	0	1	0	0	1	16	2	3	F	10	7	0	0	1
P39	4	3	10	0	4	1	1	1	12	0	6	49	0	5	F	39	20	0	0	1
P40	2	2	0	0	2	1	0	1	2	0	5	16	0	6	F	16	5	4	0	1
P41	1	2	0	0	6	1	0	6	6	0	8	0	0	4	F	3	27	6	0	1
P42	3	0	0	0	2	0	0	1	0	0	3	10	0	4	F	11	1	2	2	1
P43	1	1	1	0	4	1	0	1	0	0	6	26	2	2	F	52	2	0	0	1
P44	3	0	0	0	2	1	1	1	6	0	3	23	1	8	F	16	18	8	0	2
P45	3	0	0	0	2	0	0	0	1	0	2	5	0	3	F	17	2	1	1	2
P46	5	0	0	0	0	0	0	0	0	0	1	4	0	8	F	38	1	0	2	2
P47	3	0	0	0	1	1	0	0	0	0	2	19	3	3	F	10	2	0	0	2
P48	4	3	8	0	3	1	1	1	9	0	5	40	0	5	F	26	12	0	0	2

projekt	GUI	abstract	protected	statička klasa	inherit	maks. dubina	virtual	override ToString	override ostalo	overload	klase osim GUI	get/set	složeni get/set	neprazan konstr.	otter	event handlers	ostale metode	poziv base konstr.	statički el. van static klase	generacija
P49	1	0	0	0	0	0	0	0	0	1	0	0	0	0	T	1	9	0	0	2
P50	1	0	0	0	0	0	0	0	0	0	3	0	0	3	F	5	7	0	10	2
P51	4	0	0	0	3	0	0	1	0	0	5	23	3	6	F	18	7	3	3	2
P52	1	0	0	0	1	1	0	0	0	0	2	4	0	2	F	6	9	0	0	2
P53	7	1	0	0	2	1	0	0	1	0	4	14	0	8	F	40	24	0	1	2
P54	4	0	0	0	2	1	0	0	0	0	3	8	0	5	F	51	1	2	0	2
P55	2	13	10	1	25	0	34	0	107	0	30	83	13	19	F	11	103	23	3	2
P56	3	1	0	0	3	1	0	0	0	0	4	7	1	4	F	13	4	3	1	2
P57	2	0	0	0	0	0	0	1	0	0	2	13	0	2	F	16	2	0	0	2
P58	1	0	0	0	0	0	0	0	0	0	1	3	1	2	F	25	0	0	0	2
P59	3	0	0	0	1	0	0	1	0	0	2	13	1	4	F	15	2	1	1	2
P60	1	0	0	0	0	0	0	1	0	0	2	0	0	4	F	16	8	0	0	2
P61	3	0	0	0	1	0	0	1	0	0	2	14	0	4	F	12	5	1	1	2
P62	0	1	0	0	4	1	2	0	10	0	5	3	1	4	F	0	22	0	0	2
P63	3	0	2	0	4	1	1	0	3	0	5	4	0	4	F	6	7	4	0	2
P64	2	2	0	0	2	1	0	0	2	2	3	17	1	2	F	12	9	0	0	2
P65	2	0	0	0	2	0	0	0	0	2	2	5	3	4	F	7	10	0	0	2
P66	3	1	0	0	2	1	0	0	0	0	3	10	0	4	F	9	10	2	2	2
P67	1	0	0	0	1	1	0	0	0	0	2	2	2	1	F	7	4	0	0	2
P68	3	0	2	0	3	1	1	0	2	0	4	26	0	5	F	13	30	1	0	2
P69	3	0	0	0	1	0	0	1	0	0	2	16	0	4	F	19	4	1	1	2
P70	2	13	10	1	25	0	34	0	107	0	30	83	13	19	F	11	103	23	3	2
P71	1	0	0	0	0	0	0	0	0	0	2	0	0	2	F	10	8	0	12	2
P72	1	1	0	0	2	1	0	2	0	0	3	0	0	1	F	7	9	0	0	2
P73	0	0	0	0	5	0	0	0	0	0	5	0	0	0	F	0	17	0	1	2

projekt	GUI	abstract	protected	statička klasa	inherit	maks. dubina	virtual	override ToString	override ostalo	overload	klase osim GUI	get/set	složeni get/set	nepraz an konstr.	otter	event handlers	ostale metode	poziv base konstr.	statički el. van static klase	generacija
P74	0	1	1	0	9	1	4	0	10	2	10	8	0	7	F	0	23	8	0	2
P75	4	0	0	0	1	1	0	2	0	0	3	32	0	7	F	17	6	1	0	2
P76	1	1	0	0	5	1	0	0	0	0	9	38	0	6	F	8	27	0	0	2
P77	2	13	10	1	25	0	34	0	107	0	30	83	13	19	F	11	103	23	3	2
P78	2	0	0	0	2	0	0	0	0	2	2	5	3	4	F	7	10	0	0	2
P79	2	3	0	0	2	1	0	0	4	0	4	26	0	6	F	6	9	2	0	2
P80	1	0	0	0	0	0	0	0	0	0	3	0	0	3	F	4	7	0	10	2
P81	3	0	1	0	4	1	0	0	3	0	5	6	0	4	F	7	9	4	0	2
P82	4	2	4	0	2	1	0	2	2	0	3	22	0	6	F	20	8	2	0	2
P83	2	0	0	1	0	0	0	0	0	0	2	0	0	3	F	8	6	0	0	2
P84	1	3	0	0	6	2	0	0	4	0	8	9	0	16	F	23	43	11	0	2
P85	1	2	0	0	2	1	0	0	2	0	4	15	3	5	F	6	8	4	0	2
P86	1	1	1	0	4	1	0	1	0	0	6	24	2	2	F	34	2	0	0	2
P87	1	0	0	0	0	0	0	0	0	0	2	0	1	0	T	7	15	0	4	3
P88	5	4	3	0	2	1	0	0	3	0	3	6	0	9	F	14	2	5	4	3
P89	1	0	1	0	2	1	1	0	0	0	2	3	1	1	T	9	6	2	1	3
P90	1	1	2	0	5	2	0	0	0	0	5	7	1	5	T	0	17	5	4	3
P91	1	0	0	0	1	1	0	0	0	0	1	7	1	1	T	3	8	1	0	3
P92	1	1	0	1	3	1	3	0	2	0	5	8	4	3	T	2	14	3	3	3
P93	1	0	0	0	1	1	0	0	0	0	1	0	0	0	T	4	7	1	1	3
P94	1	1	0	0	3	2	0	0	0	1	5	4	2	2	T	4	29	3	11	3
P95	1	1	0	0	0	0	0	0	0	0	1	0	0	0	T	0	6	0	1	3
P96	1	0	0	0	0	0	0	0	0	0	2	0	1	0	T	7	15	0	4	3
P97	2	0	1	0	3	2	1	0	1	0	3	8	2	1	T	17	7	3	1	3
P98	3	7	0	0	1	1	0	0	6	1	5	4	2	6	T	8	40	0	6	3

projekt	GUI	abstract	protected	statička klasa	inherit	maks. dubina	virtual	override ToString	override ostalo	overload	klase osim GUI	get/set	složeni get/set	nepraz an konstr.	otter	event handlers	ostale metode	poziv base konstr.	statički el. van static klase	generacija
P99	5	1	0	1	2	1	0	0	0	2	5	1	0	10	F	29	11	2	0	3
P100	1	1	0	0	4	2	0	0	0	0	4	12	0	3	T	1	6	4	1	3
P101	1	4	2	1	3	2	0	0	3	0	4	3	0	1	T	0	12	3	2	3
P102	3	2	0	1	3	2	3	1	3	4	6	9	5	6	T	30	35	3	6	3
P103	1	0	0	0	3	2	0	0	0	0	3	4	0	1	T	1	10	3	1	3
P104	1	0	0	0	4	1	0	0	0	0	4	2	0	1	T	0	10	4	2	3
P105	1	0	0	0	0	0	0	0	0	1	2	0	2	2	T	1	22	0	8	3
P106	2	0	1	0	3	2	1	0	1	0	3	8	2	1	T	16	7	3	1	3
P107	1	0	0	0	0	0	0	2	0	0	5	52	0	2	T	57	25	0	0	3
P108	2	0	0	0	4	3	1	0	1	2	5	4	0	3	T	3	12	4	2	3
P109	0	0	0	0	0	0	0	0	0	2	5	1	0	4	F	9	11	0	0	3
P110	2	1	0	0	4	1	0	1	0	0	7	3	3	4	T	21	27	4	7	3
P111	2	0	0	0	4	1	0	0	0	0	4	13	1	3	T	3	13	4	6	3
P112	1	0	0	0	2	1	1	0	2	0	5	12	0	3	F	6	10	0	22	3
P113	1	1	0	0	8	2	0	0	0	0	8	8	2	6	T	3	16	7	6	3
P114	2	2	1	0	8	2	0	0	2	0	9	0	0	7	T	6	16	8	3	3
P115	2	1	0	0	6	2	0	1	0	1	7	8	0	3	T	5	20	6	8	3
P116	2	3	3	0	8	2	0	0	0	0	9	6	2	10	T	13	13	8	3	3
P117	1	3	0	2	6	2	1	0	1	1	11	0	0	0	T	2	25	6	4	3
P118	1	8	6	1	5	2	0	0	6	5	7	5	0	1	T	0	21	5	2	3
P119	2	0	0	0	4	3	1	0	1	2	5	4	0	3	T	3	12	4	2	3
P120	1	0	0	0	2	1	0	0	1	0	3	3	1	1	T	0	11	2	2	3
P121	3	1	1	1	4	2	0	0	0	1	6	2	4	5	T	8	27	4	2	3
P122	1	3	4	0	5	2	0	0	0	0	7	7	3	7	T	5	16	5	6	3
P123	1	0	0	0	4	1	0	0	2	1	4	4	2	1	T	0	14	4	4	3

projekt	GUI	abstract	protected	statička klasa	inherit	maks. dubina	virtual	override ToString	override ostalo	overload	klase osim GUI	get/set	složeni get/set	nepraz an konstr.	otter	event handlers	ostale metode	poziv base konstr.	statički el. van static klase	generacija
P124	1	3	1	0	5	3	1	0	1	0	7	4	0	2	T	1	17	6	5	3
P125	0	0	0	0	0	0	0	0	0	2	5	1	0	4	F	9	11	0	0	3
P126	1	1	0	0	2	1	0	0	0	5	3	12	2	7	T	0	25	7	3	3
P127	1	0	0	0	2	1	0	0	0	1	3	5	1	1	T	1	12	3	2	3
P128	1	0	0	0	3	2	0	0	1	4	3	2	0	2	T	4	8	3	1	3
P129	1	1	0	0	4	2	0	0	0	0	4	4	0	2	T	1	10	3	2	3
P130	2	1	0	0	3	1	0	0	1	0	4	3	2	2	T	9	16	3	2	3
P131	1	0	0	0	4	1	0	0	2	1	4	4	2	1	T	0	14	4	4	3
P132	4	1	1	0	5	2	2	0	4	0	7	9	4	4	T	9	20	5	4	3
P133	1	0	0	0	3	1	0	0	0	0	3	0	0	0	T	0	8	3	4	3
P134	1	5	7	0	9	2	0	0	4	0	11	16	3	10	T	5	25	9	8	3
P135	3	2	0	0	7	1	0	0	5	1	8	19	4	6	T	8	14	3	5	3
P136	1	5	7	0	9	2	0	0	4	0	11	16	3	10	T	5	25	9	8	3
P137	1	1	0	0	4	2	0	0	0	0	4	12	0	3	T	1	6	4	1	3
P138	1	3	3	0	7	2	0	0	0	0	8	6	1	5	T	0	16	7	6	3
P139	1	1	1	0	4	2	0	0	0	2	5	2	2	3	T	1	21	4	9	3
P140	1	0	1	0	3	2	1	0	1	0	4	8	2	2	T	6	10	3	3	3
P141	1	0	0	0	7	2	0	0	0	0	8	0	0	1	T	6	10	7	4	3
P142	1	2	2	0	4	2	0	0	0	0	5	9	1	3	T	4	13	4	2	3
P143	1	3	1	1	5	3	0	0	0	3	7	1	3	2	T	19	18	5	3	3
P144	1	0	0	0	5	2	0	0	0	0	5	0	0	0	T	3	14	5	6	3
P145	1	1	0	0	3	1	0	0	0	0	5	0	1	0	T	0	12	6	6	3
P146	1	0	0	0	2	1	1	0	2	0	5	0	0	3	F	6	10	0	16	3
P147	1	2	1	1	4	2	1	0	3	1	7	6	0	3	T	1	16	5	4	3
P148	1	2	1	0	3	2	1	0	3	2	5	11	1	6	T	9	12	5	1	3

projekt	GUI	abstract	protected	statička klasa	inherit	maks. dubina	virtual	override ToString	override ostalo	overload	klase osim GUI	get/set	složeni get/set	neprazan konstr.	otter	event handlers	ostale metode	poziv base konstr.	statički el. van static klase	generacija
P149	1	2	1	0	3	2	1	0	3	2	5	11	1	6	T	9	12	5	1	3
P150	1	0	0	0	3	2	2	0	0	0	5	11	2	4	T	10	12	3	4	3
P151	1	0	0	0	1	0	0	0	0	6	10	14	0	9	T	8	73	0	14	3
P152	1	1	0	0	4	2	0	0	0	0	4	8	0	1	T	0	9	4	2	3
P153	2	0	0	2	1	1	0	0	0	0	4	8	2	2	T	9	27	1	1	4
P154	1	1	1	0	5	2	2	0	2	0	6	12	2	5	T	3	8	5	0	4
P155	1	0	0	0	1	1	0	0	0	0	1	7	1	1	T	2	12	1	0	4
P156	1	1	1	0	4	2	1	0	2	0	4	6	0	4	T	0	7	4	0	4
P157	1	0	0	0	2	1	0	0	1	1	2	4	2	2	T	8	13	2	0	4
P158	1	1	0	0	7	2	0	0	0	0	7	6	0	6	T	0	7	7	0	4
P159	2	1	1	0	5	2	0	0	0	0	5	10	0	5	T	2	9	5	0	4
P160	2	1	2	0	5	2	2	0	6	0	5	20	0	6	T	7	6	5	3	4
P161	1	1	3	0	4	2	0	0	0	0	4	6	0	2	T	0	7	5	2	4
P162	1	1	0	0	9	3	0	0	2	0	9	12	4	7	T	0	26	9	0	4
P163	1	1	6	0	7	2	1	0	6	0	7	17	5	7	T	0	15	7	0	4
P164	1	1	0	0	4	2	0	0	1	0	4	5	1	3	T	0	9	4	2	4
P165	1	2	1	0	7	2	0	0	2	1	7	6	0	5	T	1	17	7	1	4
P166	1	1	2	0	6	3	1	0	2	1	6	5	3	6	T	0	11	6	0	4
P167	2	0	0	0	5	1	0	0	0	0	5	1	1	1	T	4	3	5	2	4
P168	1	0	1	1	2	2	0	0	1	0	3	8	2	2	T	4	15	2	0	4
P169	1	1	0	0	5	2	0	0	2	0	5	4	2	5	T	0	11	6	1	4
P170	1	1	0	0	10	2	0	0	1	1	10	31	1	9	T	0	15	10	1	4
P171	1	1	2	0	7	2	2	0	0	0	7	12	0	12	T	0	10	12	0	4
P172	1	1	0	0	5	2	0	0	2	0	5	4	2	5	T	0	11	6	1	4
P173	2	1	0	0	4	3	1	0	1	7	6	6	1	5	T	4	30	4	1	4

projekt	GUI	abstract	protected	statička klasa	inherit	maks. dubina	virtual	override ToString	override ostalo	overload	klase osim GUI	get/set	složeni get/set	nepraz an konstr.	otter	event handlers	ostale metode	poziv base konstr.	statički el. van static klase	generacija
P174	2	0	0	0	5	1	0	0	0	0	5	1	1	1	T	4	3	5	2	4
P175	2	0	0	0	4	1	0	0	2	1	4	5	5	3	T	3	41	4	0	4
P176	2	1	4	1	5	2	0	0	3	1	6	10	4	5	T	4	8	5	1	4
P177	3	2	6	0	8	2	0	0	0	1	9	15	5	9	T	4	18	8	6	4
P178	3	1	0	0	7	2	0	0	0	2	8	8	0	7	T	3	16	7	3	4
P179	3	1	4	1	6	3	0	0	1	0	7	10	2	7	T	11	16	6	0	4
P180	1	1	0	0	7	2	0	0	0	0	8	10	0	9	T	1	15	6	0	4
P181	1	2	4	0	7	2	1	0	1	0	8	8	4	7	T	1	11	7	0	4
P182	1	1	1	0	4	2	0	0	0	0	4	4	0	4	T	0	8	4	0	4
P183	2	0	0	0	2	1	0	0	2	0	2	6	2	3	T	2	12	2	0	4
P184	3	0	2	0	5	2	2	0	2	0	6	18	4	2	T	47	6	5	0	4
P185	3	0	0	1	1	1	0	0	0	1	4	8	0	6	T	11	14	1	0	4
P186	2	1	1	0	5	2	1	0	2	0	6	14	8	4	T	4	11	5	2	4
P187	1	1	6	0	7	2	1	0	6	0	7	17	5	7	T	0	15	7	0	4
P188	1	3	1	0	5	3	1	0	1	5	7	6	1	4	T	1	34	6	4	4
P189	1	1	1	0	5	2	1	0	1	1	5	6	0	5	T	0	14	5	0	4
P190	1	2	5	0	8	2	0	0	2	0	8	14	2	6	T	3	20	9	1	4
P191	1	0	0	0	4	1	0	0	0	0	4	34	0	3	T	2	15	4	0	4
P192	3	2	5	0	6	2	0	0	0	0	7	11	3	6	T	5	13	6	2	4
P193	1	0	0	0	1	1	0	0	0	0	1	7	1	1	T	2	12	1	0	4
P194	1	1	1	0	4	2	0	0	1	0	5	4	3	2	T	0	9	4	3	4
P195	1	2	6	0	14	2	0	0	2	1	15	20	6	11	T	4	50	14	2	4
P196	1	1	0	0	7	2	0	0	0	0	8	10	0	9	T	1	15	6	0	4
P197	2	1	1	0	4	2	0	0	0	5	5	6	0	5	T	4	10	4	2	4
P198	1	1	1	0	5	2	1	0	1	1	5	6	0	5	T	0	14	5	0	4

projekt	GUI	abstract	protected	statička klasa	inherit	maks. dubina	virtual	override ToString	override ostalo	overload	klase osim GUI	get/set	složeni get/set	nepraz an konstr.	otter	event handlers	ostale metode	poziv base konstr.	statički el. van static klase	generacija
P199	2	1	1	0	5	2	1	0	2	0	6	14	8	4	T	4	11	5	2	4
P200	1	1	0	0	10	2	0	0	1	1	10	31	1	9	T	0	15	10	1	4
P201	1	0	0	0	2	1	0	0	1	1	2	4	2	2	T	8	13	2	0	4
P202	1	1	0	0	4	2	0	0	0	0	4	6	0	2	T	0	8	4	1	4
P203	3	0	0	1	0	0	0	0	0	0	1	0	0	3	T	10	3	0	0	4
P204	1	3	1	1	3	1	1	0	5	0	5	5	1	2	T	1	16	2	0	4
P205	2	3	0	1	11	2	2	0	4	1	13	24	0	8	T	10	33	11	3	4
P206	5	2	8	0	20	2	8	0	0	1	21	21	6	22	T	9	20	20	5	4
P207	2	1	0	0	5	2	0	0	2	0	5	12	2	4	T	5	11	5	0	4
P208	1	1	0	0	4	2	2	0	0	1	5	8	2	3	T	1	19	4	0	4
P209	1	1	0	0	6	2	0	0	1	3	8	21	2	3	T	1	26	6	2	5
P210	1	0	0	1	5	2	0	0	1	0	6	13	1	4	T	3	12	5	0	5
P211	1	0	0	0	4	2	0	0	0	1	4	5	3	3	T	3	16	4	0	5
P212	1	0	0	1	4	2	0	0	0	4	6	12	0	2	T	0	23	4	0	5
P213	1	0	0	0	3	2	0	0	0	1	3	4	0	1	T	0	11	3	0	5
P214	1	1	0	0	6	2	0	0	0	0	7	12	1	4	T	0	7	6	0	5
P215	1	2	0	1	3	2	0	0	2	1	10	0	0	5	T	5	16	3	2	5
P216	1	2	1	1	6	3	0	0	0	0	7	0	0	4	T	1	9	6	1	5
P217	1	1	1	0	6	2	0	0	0	1	6	12	0	6	T	0	14	6	0	5
P218	2	1	0	0	6	2	1	0	5	3	8	18	7	6	T	6	33	6	4	5
P219	1	1	4	0	4	2	0	0	0	0	4	23	1	3	T	0	8	4	2	5
P220	2	1	2	0	4	2	0	0	1	0	4	7	1	5	T	2	11	4	0	5
P221	1	2	6	0	7	2	0	0	5	3	8	17	3	5	T	0	15	7	0	5
P222	1	0	1	0	3	2	0	1	1	4	6	13	0	5	T	0	24	3	4	5
P223	2	1	5	0	5	2	0	0	2	0	6	0	3	6	T	4	11	5	5	5

projekt	GUI	abstract	protected	statička klasa	inherit	maks. dubina	virtual	override ToString	override ostalo	overload	klase osim GUI	get/set	složeni get/set	nepraz an konstr.	otter	event handlers	ostale metode	poziv base konstr.	statički el. van static klase	generacija
P224	1	0	0	1	4	1	0	0	2	4	8	13	3	2	T	6	22	4	11	5
P225	2	1	0	0	5	2	0	0	4	1	5	10	8	6	T	5	18	5	0	5
P226	1	1	1	0	4	2	0	0	0	1	4	4	2	2	T	0	13	4	0	5
P227	2	0	5	0	8	2	0	0	0	0	9	3	4	18	T	4	15	16	1	5
P228	2	1	1	0	6	2	1	0	0	0	7	10	1	9	T	3	16	10	1	5
P229	1	1	1	1	3	2	0	0	1	0	5	8	2	3	T	0	10	3	1	5
P230	1	1	1	1	5	2	0	0	1	3	7	11	3	5	T	2	18	5	0	5
P231	1	1	1	0	5	2	0	0	2	0	5	6	2	2	T	3	9	5	0	5
P232	1	0	0	0	3	2	0	0	2	0	5	16	0	3	T	1	1	3	0	5
P233	1	1	1	0	5	2	0	0	2	0	5	6	2	2	T	3	9	5	0	5
P234	3	1	0	0	4	2	0	0	0	1	6	17	0	5	T	5	18	4	1	5
P235	1	2	3	0	5	3	0	0	0	1	5	6	2	2	T	4	10	5	0	5
P236	1	1	0	0	3	2	0	0	0	0	4	9	1	2	T	0	8	3	0	5
P237	2	1	1	0	3	2	0	0	0	1	3	2	2	3	T	1	12	4	0	5
P238	5	4	0	1	13	2	0	0	4	0	17	59	5	20	T	12	21	13	1	5
P239	1	1	4	0	4	2	0	0	0	1	4	8	0	2	T	3	13	4	1	5
P240	1	1	2	1	6	2	1	0	1	0	8	11	2	3	T	1	16	6	1	5
P241	2	1	2	0	5	2	0	0	0	0	5	13	1	4	T	3	15	5	2	5
P242	1	0	0	0	5	2	0	0	2	2	5	8	4	2	T	0	10	6	1	5
P243	1	1	1	0	4	2	0	0	2	1	4	10	2	4	T	1	13	5	3	5
P244	1	2	0	0	6	2	0	0	2	1	7	17	1	6	T	0	15	8	0	5
P245	3	3	1	1	6	2	0	0	0	1	9	15	1	10	T	5	15	6	0	5
P246	1	1	0	0	7	2	0	0	0	0	8	6	0	1	T	0	10	7	4	5
P247	1	0	1	0	4	1	0	0	0	1	4	8	0	3	T	0	15	4	4	5
P248	1	1	0	0	5	2	0	0	2	2	6	4	2	5	T	0	9	5	0	5

projekt	GUI	abstract	protected	statička klasa	inherit	maks. dubina	virtual	override ToString	override ostalo	overload	klase osim GUI	get/set	složeni get/set	neprazan konstr.	otter	event handlers	ostale metode	poziv base konstr.	statički el. van static klase	generacija
P249	1	0	0	0	1	1	0	0	0	0	1	2	0	1	T	0	19	1	0	5
P250	1	2	1	1	3	2	0	0	2	1	5	10	4	5	T	12	7	5	3	5
P251	1	2	1	2	9	3	0	0	0	10	14	20	1	9	T	4	58	9	0	5
P252	1	0	0	0	2	1	0	0	4	0	2	11	1	1	T	0	11	2	1	5
P253	1	1	0	0	7	2	1	0	2	0	8	11	2	6	T	0	8	7	0	5
P254	2	2	6	0	7	2	0	0	3	1	7	14	4	8	T	3	17	7	2	5
P255	1	0	0	0	1	1	0	0	0	1	3	1	0	1	T	0	12	1	4	5
P256	1	3	5	0	7	2	1	0	3	3	8	15	4	4	T	0	19	7	3	5
P257	1	1	1	0	4	2	0	0	2	0	4	12	2	4	T	0	12	4	1	5
P258	1	1	2	0	6	2	2	0	6	2	7	15	6	6	T	0	18	6	0	5
P259	1	1	1	0	5	2	0	0	1	1	5	13	1	3	T	0	7	5	0	5
P260	6	1	1	0	3	2	0	2	0	1	5	38	2	10	T	18	12	3	1	5
P261	2	1	2	0	6	2	0	0	0	0	6	10	2	6	T	4	22	6	3	5
P262	1	1	7	0	7	2	0	0	1	1	7	35	7	5	T	4	34	8	2	5
P263	1	0	0	0	2	1	0	0	1	0	3	7	1	2	T	0	15	2	1	5
P264	1	1	1	0	6	2	0	0	2	1	6	7	3	3	T	0	6	6	0	5
P265	1	0	5	0	6	2	0	0	3	0	6	16	4	6	T	0	9	6	0	5
P266	2	2	3	0	7	2	0	0	0	1	7	8	2	6	T	3	7	7	0	5
P267	1	1	0	1	5	2	0	0	0	0	6	16	0	4	T	0	22	5	0	5
P268	1	0	0	2	4	1	0	0	0	2	5	16	14	8	T	2	20	4	1	5
P269	1	1	4	0	4	2	0	0	0	0	4	10	0	4	T	0	16	4	5	5
P270	2	1	3	0	7	2	0	0	2	0	7	8	2	3	T	7	11	7	2	5
P271	2	2	3	0	7	2	1	0	2	0	8	11	3	8	T	5	13	7	1	5
P272	4	1	0	1	2	1	0	0	0	4	7	26	4	13	T	24	25	3	5	5
P273	1	1	1	0	4	2	1	0	2	0	4	6	0	4	T	0	8	4	0	5

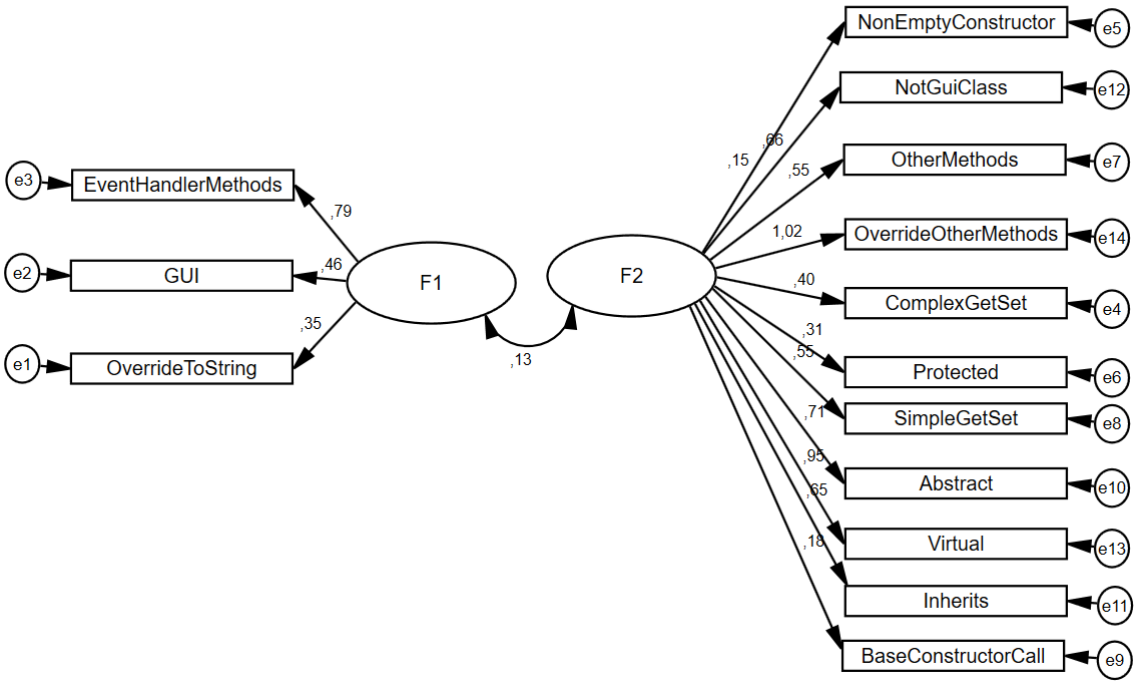
projekt	GUI	abstract	protected	statička klasa	inherit	maks. dubina	virtual	override ToString	override ostalo	overload	klase osim GUI	get/set	složeni get/set	nepraz an konstr.	otter	event handlers	ostale metode	poziv base konstr.	statički el. van static klase	generacija
P274	1	2	4	0	7	2	1	0	1	0	8	9	2	7	T	0	11	7	0	5
P275	1	1	4	1	7	2	0	0	1	0	8	19	1	4	T	3	22	7	0	5
P276	1	1	1	1	6	2	0	1	0	1	7	8	0	3	T	14	14	6	0	5
P277	1	1	1	0	3	2	0	0	2	0	3	6	2	2	T	3	8	3	1	5
P278	1	1	2	1	2	2	0	0	1	0	3	2	2	2	T	7	9	2	0	5
P279	2	1	4	0	5	2	1	0	5	3	7	16	5	5	T	6	33	5	2	5
P280	3	1	0	0	7	2	0	0	2	0	7	26	2	8	T	1	13	7	0	5
P281	1	1	8	0	7	2	0	0	2	0	7	20	4	7	T	1	9	7	0	5
P282	1	0	0	0	2	1	0	0	0	0	1	4	0	1	T	5	16	1	0	5
P283	2	3	0	1	9	2	2	0	2	10	11	1	3	6	T	11	32	8	0	5
P284	1	2	0	0	12	3	0	0	0	3	12	17	1	6	T	4	32	18	0	5
P285	1	0	3	0	2	1	0	0	1	1	2	9	1	2	T	0	6	2	0	5
P286	1	2	5	0	7	2	0	0	5	3	8	16	4	5	T	0	14	7	0	5
P287	1	1	3	0	4	2	0	0	2	2	5	10	2	3	T	3	25	4	1	5
P288	2	4	8	1	5	3	0	0	4	0	7	1	5	9	T	3	23	8	0	5
P289	1	1	5	0	4	2	0	0	0	1	4	10	0	3	T	0	11	4	0	5

Prilog 4 – Statistika varijabli za sve projekte

		GUI	Abstract	Protected	Static Class	Inherits	MaxDepthOf Inheritance	Virtual	Override ToString	Override Other Methods	Overload
N	Valid	289	289	289	289	289	289	289	289	289	289
	Missing	0	0	0	0	0	0	0	0	0	0
Mean		1,78	1,12	1,17	0,16	4,24	1,44	0,64	0,23	2,36	0,84
Median		1,00	1,00	0,00	0,00	4,00	2,00	0,00	0,00	1,00	0,00
Std. Deviation		1,205	1,665	2,082	0,414	3,511	0,798	3,503	0,633	10,898	1,750
Variance		1,453	2,771	4,333	0,171	12,330	0,636	12,272	0,401	118,767	3,062
Skewness		1,615	4,369	2,296	2,547	2,689	-0,379	9,091	4,395	9,286	3,544
Std. Error of Skewness		0,143	0,143	0,143	0,143	0,143	0,143	0,143	0,143	0,143	0,143
Kurtosis		2,574	26,699	5,135	6,064	12,696	-0,572	84,476	28,487	87,365	15,500
Std. Error of Kurtosis		0,286	0,286	0,286	0,286	0,286	0,286	0,286	0,286	0,286	0,286
Range		7	13	10	2	25	3	34	6	107	12
Minimum		0	0	0	0	0	0	0	0	0	0
Maximum		7	13	10	2	25	3	34	6	107	12
		Not GuiClass	Simple GetSet	Complex GetSet	NonEmpty Constructor	Using OTTER	Event Handler Methods	Other Methods	Base Constructor Call	Static Nodes In Non Static Class	Generation
N	Valid	289	289	289	289	289	289	289	289	289	**289**
	Missing	0	0	0	0	0	0	0	0	0	0
Mean		5,46	11,58	1,60	4,62	0,71	8,13	14,84	4,12	1,67	

Median		5,00	9,00	1,00	4,00	1,00	4,00	12,00	4,00	1,00	
Std. Deviation		3,850	11,814	2,171	3,445	0,456	11,333	13,462	3,776	2,774	
Variance		14,825	139,564	4,714	11,868	0,208	128,441	181,236	14,257	7,695	
Skewness		3,219	3,035	2,690	1,956	-0,908	2,630	3,682	1,954	3,151	
Std. Error of Skewness		0,143	0,143	0,143	0,143	0,143	0,143	0,143	0,143	0,143	
Kurtosis		16,781	13,859	10,512	6,044	-1,183	8,459	19,337	6,849	14,222	
Std. Error of Kurtosis		0,286	0,286	0,286	0,286	0,286	0,286	0,286	0,286	0,286	
Range		30	83	14	22	1	69	103	23	22	
Minimum		0	0	0	0	0	0	0	0	0	
Maximum		30	83	14	22	1	69	103	23	22	

Prilog 5 - SEM model 1



Model	NPAR	CMIN	DF	P	CMIN/DF
Default model	29	451,563	76	,000	5,942
Saturated model	105	,000	0		
Independence model	14	628,933	91	,000	6,911
Zero model	0	2016,000	105	,000	19,200

RMSEA

Model	RMSEA	LO 90	HI 90	PCLOSE
Default model	,131	,119	,143	,000
Independence model	,143	,133	,154	,000

AIC

Model	AIC	BCC	BIC	CAIC
Default model	509,563	512,750	615,890	644,890
Saturated model	210,000	221,538	594,975	699,975
Independence model	656,933	658,471	708,263	722,263
Zero model	2016,000	2016,000	2016,000	2016,000

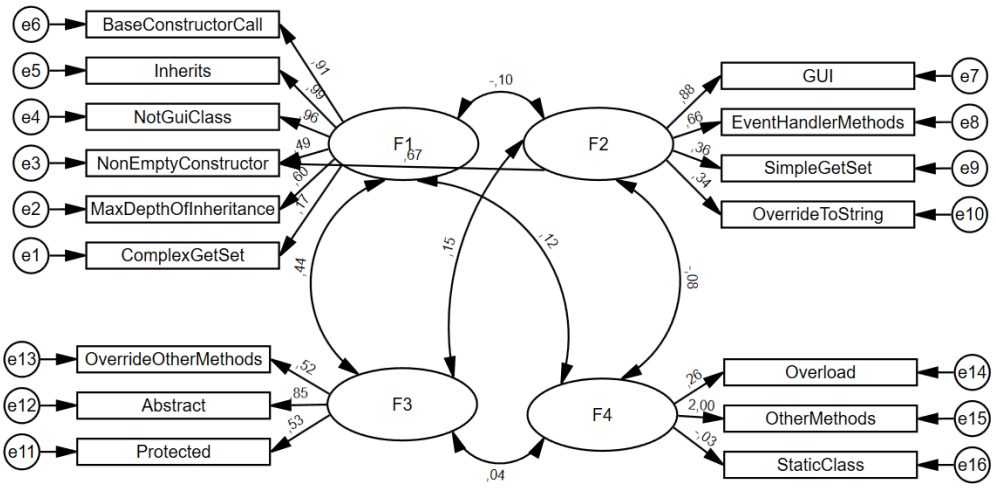
Prilog 6 - Statistika nakon izbacivanja ekstrema

		GUI	Abstract	Protected	StaticClass	Inherits	MaxDepth Of Inheritance	Virtual	Override ToString	Override Other Methods	Overload
N	Valid	286	286	286	286	286	286	286	286	286	286
	Missing	0	0	0	0	0	0	0	0	0	0
Mean		1,77	0,99	1,07	0,15	4,02	1,45	0,29	0,23	1,27	0,85
Median		1,00	1,00	0,00	0,00	4,00	2,00	0,00	0,00	0,00	0,00
Std. Deviation		1,211	1,139	1,884	0,407	2,807	0,788	0,752	0,636	1,886	1,757
Variance		1,467	1,298	3,549	0,166	7,877	0,621	0,566	0,405	3,557	3,087
Skewness		1,613	2,078	2,186	2,680	1,269	-0,391	5,083	4,370	2,329	3,525
Std. Error of Skewness		0,144	0,144	0,144	0,144	0,144	0,144	0,144	0,144	0,144	0,144
Kurtosis		2,533	7,553	4,498	6,839	4,113	-0,513	40,903	28,185	7,144	15,328
Std. Error of Kurtosis		0,287	0,287	0,287	0,287	0,287	0,287	0,287	0,287	0,287	0,287
Range		7	8	10	2	20	3	8	6	12	12
Minimum		0	0	0	0	0	0	0	0	0	0
Maximum		7	8	10	2	20	3	8	6	12,000	12
		NotGui Class	SimpleGetSet	ComplexGetSet	NonEmpty Constructor	Using OTTER	Event Handler Methods	Other Methods	Base Constructor Call	StaticNodes In NonStatic Class	Generation
N	Valid	286	286	286	286	286	286	286	286	286	286
	Missing	0	0	0	0	0	0	0	0	0	0
Mean		5,20	10,83	1,48	4,47	0,71	8,10	13,92	3,92	1,65	
Median		5,00	9,00	1,00	4,00	1,00	4,00	12,00	4,00	1,00	
Std. Deviation		2,928	9,315	1,839	3,130	0,453	11,389	10,024	3,258	2,785	

Variance		8,574	86,774	3,380	9,794	0,205	129,705	100,481	10,615	7,757	
Skewness		1,422	1,722	2,144	1,755	-0,948	2,627	2,084	1,194	3,160	
Std. Error of Skewness		0,144	0,144	0,144	0,144	0,144	0,144	0,144	0,144	0,144	
Kurtosis		4,238	4,502	8,213	5,977	-1,109	8,385	6,944	3,128	14,193	
Std. Error of Kurtosis		0,287	0,287	0,287	0,287	0,287	0,287	0,287	0,287	0,287	
Range		21	59	14	22	1	69	73	20	22	
Minimum		0	0	0	0	0	0	0	0	0	
Maximum		21	59	14	22	1	69	73,000	20	22	

Rotated Component Matrix[a]					
	Component				
	1	2	3	4	5
BaseConstructorCall	0,913				
Inherits	0,909				
NotGuiClass	0,834				
NonEmptyConstructor	0,697	0,484			
MaxDepthOfInheritance	0,627				
ComplexGetSet	0,407				
GUI		0,815			
EventHandlerMethods		0,800			
SimpleGetSet		0,592			
OverrideToString		0,587			
OverrideOtherMethods			0,850		
Abstract			0,671		
Protected			0,634		
Overload				0,820	
OtherMethods				0,787	
StaticClass				0,539	
StaticNodesInNonStaticClass					0,775
Virtual					0,494
Extraction Method: Principal Component Analysis. Rotation Method: Varimax with Kaiser Normalization.[a]					
a. Rotation converged in 6 iterations.					

Prilog 7 - SEM model 2 - GLS



Model	NPAR	CMIN	DF	P	CMIN/DF
Default model	39	347,187	97	,000	3,579
Saturated model	136	,000	0		
Independence model	16	636,014	120	,000	5,300
Zero model	0	2280,000	136	,000	16,765

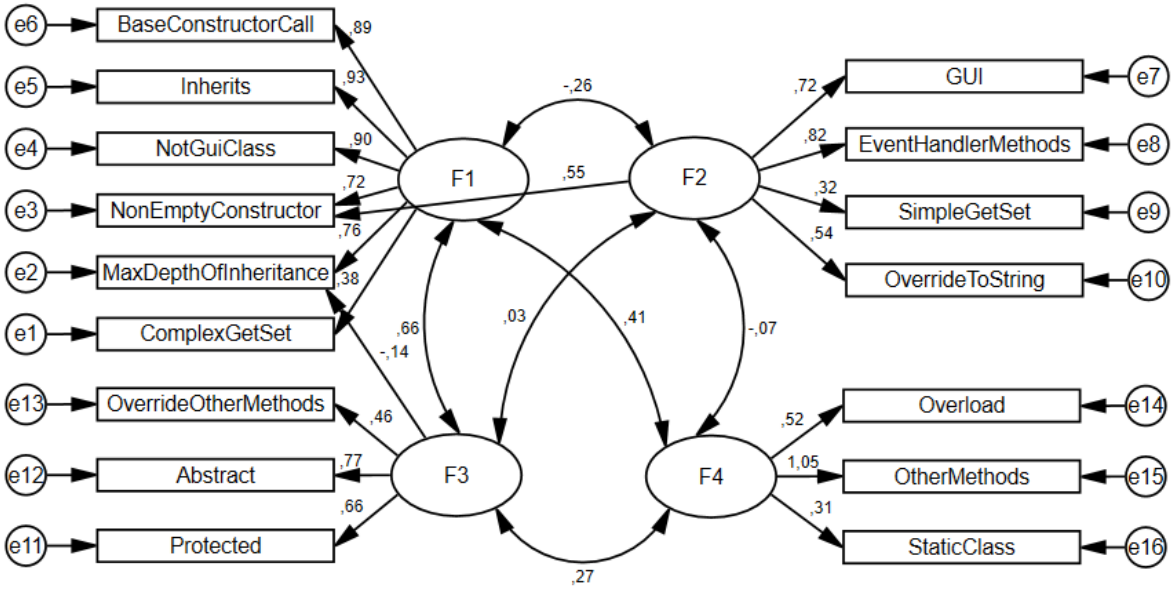
RMSEA

Model	RMSEA	LO 90	HI 90	PCLOSE
Default model	,095	,084	,106	,000
Independence model	,123	,114	,132	,000

AIC

Model	AIC	BCC	BIC	CAIC
Default model	425,187	430,135	567,771	606,771
Saturated model	272,000	289,254	769,215	905,215
Independence model	668,014	670,044	726,510	742,510
Zero model	2280,000	2280,000	2280,000	2280,000

Prilog 8 - SEM Model 2 SFLS



Model Fit Summary

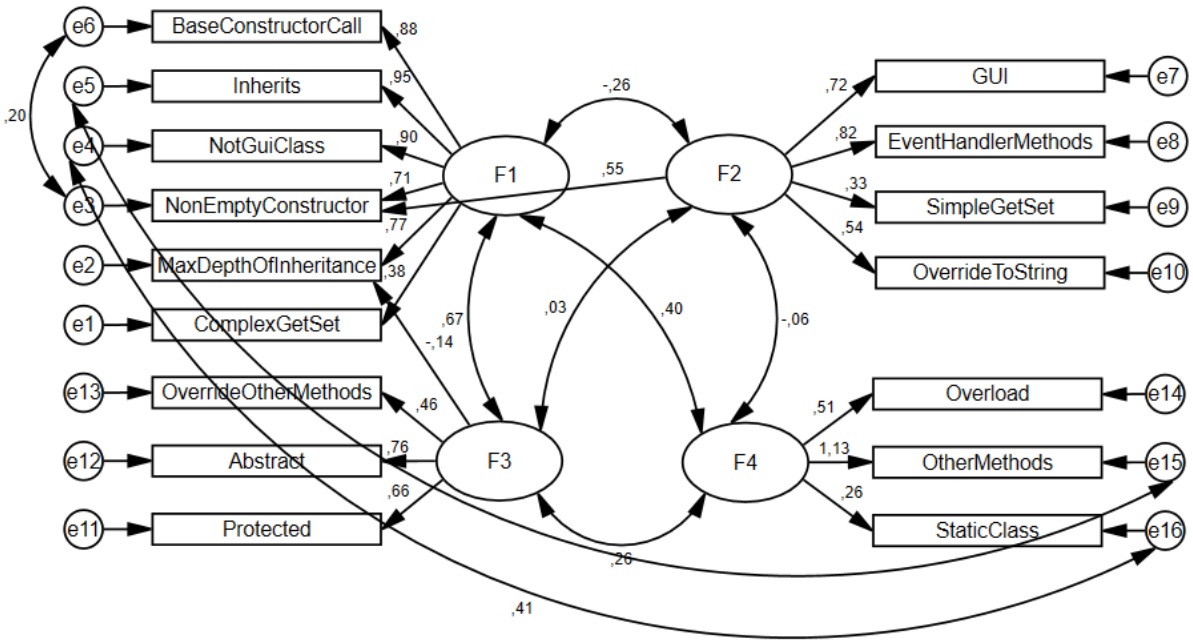
CMIN

Model	NPAR	CMIN
Model2-SLS-K1	40	306,612
Saturated model	136	,000
Independence model	16	3118,039
Zero model	0	5398,039

RMR, GFI

Model	RMR	GFI	AGFI	PGFI
Model2-SLS-K1	1,958	,943	,920	,666
Saturated model	,000	1,000		
Independence model	4,499	,422	,345	,373
Zero model	16,573	,000	,000	,000

Prilog 9 - SEM model 2 SFLS (povezivanje)



Model Fit Summary

CMIN

Model	NPAR	CMIN
Model2-SLS-K2	43	296,567
Saturated model	136	,000
Independence model	16	3118,039
Zero model	0	5398,039

RMR, GFI

Model	RMR	GFI	AGFI	PGFI
Model2-SLS-K2	1,951	,945	,920	,646
Saturated model	,000	1,000		
Independence model	4,499	,422	,345	,373
Zero model	16,573	,000	,000	,000

ŽIVOTOPIS I POPIS JAVNO OBJAVLJENIH RADOVA

Divna Krpan rođena je 21. rujna 1977. godine u Imotskom. Osnovnu školu i Prirodoslovno-matematičku gimnaziju Dr. Mate Ujevića završila je u Imotskom. Diplomirala je na Fakultetu prirodoslovno matematičkih znanosti i odgojnih područja Sveučilišta u Splitu 2004. godine i stekla zvanje profesora matematike i informatike. Iste godine zaposlila se kao učitelj matematike i informatike u Osnovnoj školi "Plokite" u Splitu. U razdoblju od 2004. do 2006. je radila i kao vanjski suradnik u naslovnom zvanju asistenta na Odjelu za informatiku PMF-a u Splitu te kao vanjski suradnik – asistent na Kemijsko-tehnološkom fakultetu Sveučilišta u Splitu.

Od početka školske godine u 2006. godini radila je kao učitelj informatike u Osnovnoj školi "Meje" u Splitu gdje je radila do prosinca 2006. kada se zapošljava na PMF-u u Splitu na radno mjesto asistenta na Odjelu za informatiku. U ak. godini 2008./2009. radila je i kao vanjski suradnik - asistent na Filozofskom fakultetu Sveučilišta u Splitu.

Godine 2012. Izabrana je u nastavno zvanje i radno mjesto predavača za znanstveno područje tehničkih znanosti, polje računarstvo, a godine 2018. izabrana je u nastavno zvanje i radno mjesto višeg predavača.

ZNANSTVENI RADOVI

Izvorni znanstveni i pregledni radovi u CC časopisima

Krpan, Divna; Mladenović , Saša; Zaharija, Goran The Framework for Project Based Learning of Object-Oriented Programming. // International journal of engineering education, 35 (2019), 5; 1366-1377 (međunarodna recenzija, članak, znanstveni)

Mladenović, Saša; Krpan, Divna; Mladenović, Monika Using Games to Help Novices Embrace Programming: From Elementary to Higher Education. // International journal of engineering education, 32 (2016), 1B; 521-531 (međunarodna recenzija, članak, znanstveni)

Znanstveni radovi u drugim časopisima

Mladenović, Monika; Krpan, Divna; Mladenović, Saša Learning programming from Scratch. // The Turkish Online Journal of Educational Technology, 2 (2017), 419-427 (međunarodna recenzija, članak, znanstveni)

Stankov, Slavomir; Glavinic, Vlado; Krpan, Divna Group Modeling in Social Learning Environments. // International Journal of Distance Education Technologies (IJDET), 10 (2012), 2; 39-56 doi:10.4018/jdet.2012040103 (međunarodna recenzija, članak, znanstveni)

Stankov, Slavomir; Grubišić, Ani; Žitko, Branko; Krpan, Divna Vrednovanje učinkovitosti procesa učenja i poučavanja u sustavima za e-učenje. // Školski vjesnik, 54 (2005), 1/2; 21-31. (https://www.bib.irb.hr/209610) (podatak o recenziji nije dostupan, članak, znanstveni)

Znanstveni radovi u zbornicima skupova

Krpan, Divna; Mladenović, Saša; Ujević, Biserka TANGIBLE PROGRAMMING WITH AUGMENTED REALITY. // INTED2018 Proceedings / Gómez Chova, L. ; López Martínez, A. ; Candel Torres, I. (ur.). Valencia, SPAIN: IATED Academy, 2018. str. 4993-5000 (predavanje, cjeloviti rad (in extenso), znanstveni)

Krpan, Divna; Mladenović, Saša; Zaharija, Goran Mediated Transfer from Visual to High-level Programming Language. // The 40th Jubilee International ICT Convention – MIPRO 2017 / Petar Biljanović (ur.). Opatija, 2017. str. 906-912 (predavanje, međunarodna recenzija, cjeloviti rad (in extenso), znanstveni)

Mladenović, Monika; Krpan, Divna; Mladenović, Saša INTRODUCING PROGRAMMING TO ELEMENTARY STUDENTS NOVICES BY USING GAME DEVELOPMENT IN PYTHON AND SCRATCH. // EDULEARN16 Proceedings Barcelona, Španjolska, 2016. str. 1622-1629 (ostalo, međunarodna recenzija, cjeloviti rad (in extenso), znanstveni)

Krpan, Divna; Mladenović, Saša; Rosić, Marko Undergraduate Programming Courses, Students' Perception and Success. // International Conference on New Horizons in Education, INTE 2014 Pariz, Francuska: Procedia - Social and Behavioral Sciences, Elsevier, 2014. str. 3868-3872 (predavanje, međunarodna recenzija, cjeloviti rad (in extenso), znanstveni)

Krpan, Divna; Rosić, Marko; Mladenović, Saša Teaching Basic Programming Skills to Undergraduate Students. // Proceedings of CIET 2014 / Plazibat, Bože ; Kosanović, Silvana (ur.). Split: University of Split, 2014. (predavanje, međunarodna recenzija, cjeloviti rad (in extenso), znanstveni)

Krpan, Divna; Stankov, Slavomir Educational Data Mining for Grouping Students in E-learning System. // Proceedings of the ITI 2012 34th Int. Conf. on Information Technology Interfaces / Luzar-Stiffler, Vesna ; Jarec, Iva ; Bekic, Zoran (ur.). Zagreb: University Computing Centre,

University of Zagreb, 2012. str. 207-212 (ostalo, međunarodna recenzija, cjeloviti rad (in extenso), znanstveni)

Krpan, Divna; Stankov, Slavomir Standards and Specifications for E-Learning Systems. // Proceedings ELMAR - 2009 / Grgić, Mislav ; Božek, Jelena ; Grgić, Sonja (ur.). Zagreb: Croatian Society Electronics in Marine - ELMAR, 2009. str. 189-192 (predavanje, međunarodna recenzija, cjeloviti rad (in extenso), znanstveni)

Krpan, Divna; Bilobrk, Ivan Introductory Programming Languages in Higher Education. // Computers in Education (Računala u obrazovanju) / Čičin-Šain, Marina ; Uroda, Ivana ; Turčić-Prstačić, Ivana ; Sluganović, Ivana (ur.). Rijeka: MIPRO, 2011. str. 375-380 (predavanje, međunarodna recenzija, cjeloviti rad (in extenso), ostalo)

Poglavlja u knjigama

Krpan, Divna; Tomaš, Suzana; Vladušić, Roko Using Effect Size for Group Modeling in E-Learning Systems. // Intelligent Tutoring Systems in E-Learning Environments: Design, Implementation and Evaluation / Stankov, Slavomir ; Glavinić, Vlado ; Rosić, Marko (ur.). Hershey, PA, USA: IGI Global, 2011. str. 237-257

Stručni radovi

Krpan, Divna; Brčić, Damir Posredovani prijenos u poučavanju programiranja s vizualnim programskim jezicima. // Politehnika - Časopis za tehnički odgoj i obrazovanje, 2 (2018), 1; 72-79 (domaća recenzija, članak, stručni)

Zaharija, Goran; Mladenović, Saša; Krpan, Divna NetLogo – novo okruženje za podučavanje informatike. // 16. CARNetova korisnička konferencija - CUC 2014 - Zbornik radova / Ana Orlović (ur.). Zagreb: Hrvatska akademska i istraživačka mreža - CARNet, 2014. (predavanje, domaća recenzija, cjeloviti rad (in extenso), stručni)

Krpan, Divna; Mladenović, Saša; Zaharija, Goran Vizualni programski jezici u visokom obrazovanju. // 16. CARNetova korisnička konferencija - CUC 2014 - Zbornik radova / Ana Orlović (ur.). Zagreb: Hrvatska akademska i istraživačka mreža - CARNet, 2014. (predavanje, domaća recenzija, cjeloviti rad (in extenso), stručni)