Analiza tehničkog duga i obrazaca programskih pogrešaka u studentskim projektima: utjecaj platforme i razdoblja nakon dostupnosti GenUI

Zlatko Stapić, Dario Ljubas, Lovro Posarić
Sveučilište u Zagrebu
Fakultet organizacije i informatike
Varaždin, Hrvatska
{zstapic, dljubas, lposaric}@foi.hr

***Sažetak*—U području edukacije programskog inženjerstva, kratki rokovi i fokus na funkcionalnost često dovode do nakupljanja tehničkog duga u studentskim projektima. Ovaj rad analizira 244 studentska projekta izrađena na završnoj godini prijediplomskog studija kroz razdoblje od pet godina (2020–2024), koristeći alate za statičku analizu koda. Cilj istraživanja bio je identificirati najčešće obrasce pogrešaka te utvrditi utjecaj razvojne platforme (.NET i *Android*) i razdoblja nakon šire dostupnosti generativnih UI alata na kvalitetu kôda. Rezultati pokazuju statistički značajnu ovisnost vrste pogrešaka o platformi, pri čemu *Android* projekti pate od viška nekorištenog koda, dok .NET projekti pokazuju strukturne nedostatke u primjeni objektno-orijentiranih načela. Posebno je značajan nalaz promjene profila pogrešaka nakon 2023. godine, gdje se uočava smanjenje redundancije koda, ali i porast problema s integracijom i 'mrtvim' kodom. Rad zaključuje da moderni kurikulumi trebaju preusmjeriti fokus s pisanja kôda na vještine revizije i održavanja rješenja, uključujući i ona izrađena uz pomoć generativne umjetne inteligencije.**

***Ključne riječi*—tehnički dug, programske pogreške, studentski projekti, tematska analiza**

I. Uvod

U obrazovanju iz programskog inženjerstva (eng. *software engineering education*) studenti se gotovo uvijek nalaze u napetosti između dvaju ciljeva: s jedne strane očekuje se usvajanje dobrih praksi i temeljnih načela kvalitetnog dizajna, a s druge strane rad u semestralnim projektima često je obilježen kratkim rokovima, paralelnim obvezama i fokusom na isporuku funkcionalnosti [1], [2]. U takvom okruženju prirodno dolazi do kompromisa koji rezultiraju nakupljanjem tehničkog duga (eng. *technical debt*) [3] i različitih vrsta programskih grešaka, osobito u završnim verzijama studentskih projekata koje se nakon predaje rijetko dalje održavaju ili refaktoriraju [4].

Tehnički dug predstavlja metaforu koja opisuje troškove budućih dorada i prerada uzrokovane izborom bržih ili jednostavnijih rješenja umjesto kvalitetnijih pristupa koji bi inicijalno zahtijevali veći napor. U profesionalnim okruženjima posljedice takvih odluka često postaju vidljive tijekom održavanja i nadogradnji [3], dok u studentskim projektima studenti te posljedice rjeđe izravno osjete jer je životni ciklus rješenja kratak i ograničen akademskim semestrom [4]. Ipak, tehnički dug se u studentskom kodu očituje kroz prepoznatljive obrasce: "mrtvi" ili nekorišteni kod (eng. *dead code*), kršenja načela objektno-orijentiranog dizajna, nedosljedne konvencije imenovanja, nepravilno rukovanje iznimkama, kao i pojave koje alati za statičku analizu koda (eng. *static code analysis*) klasificiraju kao potencijalne probleme (eng. *code smells*), greške (eng. *bugs*) i ranjivosti (eng. *vulnerabilities*) [5–7].

Sustavna analiza takvih problema u studentskim projektima može poslužiti kao instrument unaprjeđenja ili vrednovanja nastave: umjesto oslanjanja isključivo na subjektivne dojmove ili pojedinačne primjere, moguće je na temelju većeg uzorka projekata identificirati ponavljajuće slabosti i povezati ih s ishodima učenja, tehnologijama i načinom rada studenata [8], [9]. U ovom radu analiziramo 244 studentska projekta izrađena u sklopu triju kolegija tijekom pet akademskih godina (2020–2024), pri čemu su projekti razvijani pretežito na platformama .NET i *Android*. Set podataka (eng. *data set*) izrađen je temeljem rezultata statičke analize kôda korištenjem alata *SonarQube (Community Edition)*. Kvantitativna i kvalitativna analiza podataka provedena je na rezultatima tematske analize (eng. *thematic analysis*) kojom se detektirani nalazi grupiraju u semantički smislenije kategorije pogrešaka [10].

U skladu s navedenim ciljevima, rad odgovara na sljedeća istraživačka pitanja:

IP1: Koje su najčešće programske pogreške koje se pojavljuju u studentskim projektima na završnoj godini prijediplomskog studija?

IP2: Postoji li statistički značajan utjecaj razvojne platforme/tehnologije (.NET u odnosu na *Android*) na učestalost pojedinih vrsta programskih pogrešaka?

IP3: Postoji li značajna razlika u profilu programskih pogrešaka u razdoblju prije šire dostupnosti generativnih UI alata i nakon nje?

Ostatak rada organiziran je kako slijedi: u drugom poglavlju opisani su skup podataka i metodologija prikupljanja i obrade nalaza (statička analiza i tematsko kodiranje), treće poglavlje prikazuje rezultate po istraživačkim pitanjima, četvrto poglavlje raspravlja ograničenja istraživanja, a u zaključku se sintetiziraju implikacije za unaprjeđenje kurikuluma i nastavnih

aktivnosti usmjerenih na kvalitetu, održavanje i reviziju koda.

A. *Postojeća istraživanja*

Iako se tehnički dug najčešće promatra u industrijskom kontekstu, istraživanja sve češće pokazuju da se analitika tehničkog duga (eng. *technical debt analytics*) može koristiti i kao evaluacijski instrument u nastavi: mjerenjem učestalih uzoraka duga i pogrešaka na većem broju studentskih projekata moguće je identificirati stabilne slabosti, povezati ih s nastavnim sadržajem i planirati ciljane intervencije (npr. dodatne vježbe refaktoriranja, sigurnosti, dizajna). Horváth i sur. [9] prikazuju primjer kako se alati za analizu tehničkog duga mogu iskoristiti za poboljšanje poučavanja i povratnih informacija studentima. Dodatno, noviji radovi u domeni obrazovanja pokazuju da uvođenje statičke analize (npr. *SonarCloud/SonarQube*) kao dijela nastavnog procesa može mjerljivo poboljšati određene aspekte kvalitete (posebno sigurnost i pouzdanost), te da se promjene mogu pratiti usporedbom kohorti kroz akademske godine [7], [11]. Upravo analiza finalnih verzija projekata nakon završenih kolegija (eng. *post-course*) može biti iskorištena kao podloga za kurikularne prilagodbe.

S druge strane, Berger i sur. [12] ističu da ukupna gustoća pogrešaka u programskom kôdu nije određena samim jezikom, već domenom projekta, primjerice ako se radi o web projektu ili mobilnom projektu. Također, empirijska istraživanja u visokom obrazovanju ukazuju da ChatGPT i slični alati mogu mijenjati ponašanja studenata u procesu programiranja (npr. više iteracija uz "*debugging*" i čitanje povratnih informacija), uz kompleksne učinke na izvedbu i percepciju korisnosti [13], te na samoučinkovitost i motivaciju [14]. Cotroneo i sur. [15] ukazuju na to kako programski kod koji su napisali ljudi karakterizira veća razina strukturne složenosti, više sintaktičkih pogrešaka, te više problema s održavanjem zbog lošeg dizajna. Međutim, UI-generiran kod je jednostavniji i repetitivniji. Sintaktičkih pogrešaka je značajno manje, ali se broj sigurnosnih ranjivosti i haluciniranih varijable značajno povećan.

U svrhu otkrivanja pogrešaka i nedostataka u programskom kodu razvijeni su i razni alati statičke i dinamičke prirode kao što su SARIF (*Static Analysis Results Interchange Format*) razvijen je u svrhu lakše interoperabilnosti alata za analizu koda, kao način za automatizirano izvještavanje o kvaliteti koda, sigurnosnim propustima i usklađenosti sa standardima [16], *AnaConDebt* - fokusira se na analizu usklađenosti s arhitektonskim obrascima i identificiranje neispravnosti na razini arhitekture koji dovode do duga [17], *SATD Detector* - koristi obradu prirodnog jezika za skeniranje komentara koda u potrazi za frazama poput "*TODO*", "*Fix me*" ili "*Hack*", klasificirajući ih kao *namjerni tehnički dug* [18], *VisiminerTD* - pretražuje Git repozitorije kako bi vizualizirao razvoj tehničkog duga tijekom vremena, pomažući studentima da vide kada je dug uveden [19], *SonarQube* - glavni primjer implementacije SQALE metodologije za evaluaciju kvalitete programskog koda [20] *Clean-CaDET* - skup alata pogonjenih umjetnom inteligencijom za detekciju tehničkog duga, Tutor modul za podučavanje čistog koda [21], i dr.

Iznad navedeni rezultati istraživanja podupiru pretpostavku da šira dostupnost generativne UI (eng. *generative AI*) može promijeniti ne samo brzinu izrade rješenja, nego i vrstu pogrešaka koje ostaju prisutne u završnim verzijama projekata (npr. integracija, nekorišteni artefakti, imenovanje, sigurnosne ranjivosti), što mi u ovom istraživanju kvantificiramo preko statičke analize i potom konceptualno organiziramo tematskim grupiranjem. Zaključno, smatramo da je bitno razumjeti kako se mijenja priroda studentskih grešaka kako bi se ishodi učenja, metode poučavanja i oblici nastave prilagodili i poboljšali.

II. Metodologija

Skup podataka korišten u istraživanju stvoren je analizom 244 studentska projekta izrađena u sklopu tri kolegija iz područja programskog inženjerstva i razvoja aplikacija izvedenih tijekom pet akademskih godina (2020–2024). Većina projekata razvijena je u .NET okruženju, dok manji dio čine *Android* aplikacije razvijene u Kotlinu, kako je prikazano u Tablici I.

Tablica I
Analiza projekata po kolegiju, godini i platformi

Platforma	Kolegij	2020	2021	2022	2023	2024	Ukup.
.NET	Programsko inženjerstvo	54	39	40	12	2	**147**
	Razvoj programskih proizvoda	–	–	17	14	17	**48**
Android	Razvoj aplikacija za mobilne i pametne uređaje	–	–	17	18	18	**53**

Istraživanje je provedeno korištenjem mješovitog istraživačkog pristupa koji kombinira kvantitativnu analizu metrika kvalitete koda dobivenih statičkom analizom sa kvalitativnom tematskom analizom (eng. *thematic analysis*) [10] ponavljajućih obrazaca grešaka u studentskim softverskim projektima. Najprije su se prikupljali i kvantificirali podaci pomoću statičke analize programskih kodova svih projekata, a zatim se vršila *tematska analiza* kako bi se utvrdila povezanost između učestalosti grešaka s metrikama tehničkog duga.

U uzorak su uvršteni isključivo projekti koji su zadovoljili tri uvjeta: dostupnost kompletnog izvornog koda, mogućnost uspješnog kompajliranja (eng. *build*) te mogućnost provedbe statičke analize. Time je osigurana usporedivost projekata i pouzdanost dobivenih metrika. Prikupljanje podataka provedeno je primjenom alata SonarQube (Community Edition), koji je korišten za statičku analizu izvornog koda svih uključenih projekata. Analiza je izvršena uz korištenje zadanih kvalitativnih profila za C# i Kotlin, pri čemu su iz skeniranja isključene biblioteke trećih strana. Svaki projekt je prije analize uspješno kompajliran kako bi se osigurala tehnička ispravnost analize, nakon čega je SonarQube Scanner pokrenut s istom konfiguracijom za sve projekte. Koristeći PowerShell skriptu za komunikaciju sa *SonarQube Web API*-jem, rezultati analize izvezeni su u strukturiranu CSV datoteku te objedinjeni u skup podataka koji je sadržavao ukupno 18.828 detektiranih problema (eng. *issues*) u studentskim projektima.

Svaki zapis sadrži podatke o tipu problema (*code smell*, *bug* ili *vulnerability*), razinu ozbiljnosti, opis problema, platformu,

akademsku godinu i identifikator projekta. Podaci su dodatno očišćeni i standardizirani kako bi se uklonili projektno-specifični elementi, uz zadržavanje konteksta potrebnog za interpretaciju grešaka. Kvantitativna obrada i statističke analize provedene su u *Python* okruženju, dok su tehnički dug i povezane metrike izračunate prema SQALE [20] metodologiji implementiranoj u *SonarQubeu*.

Kvalitativni dio istraživanja temelji se na *tematskoj analizi* prema okviru Braun i Clarke [10], prilagođenom za računalnu obradu softverskih grešaka. Nakon inicijalne analize definiranog skupa podataka, razvijen je klasifikacijski algoritam koji provodi kodiranje opisa problema kombinacijom deduktivnog pristupa temeljenog na *SonarQube* taksonomiji, i induktivnog pristupa koji je omogućio identifikaciju obrazaca pogrešaka specifičnih za studentske projekte. Kodovi su putem ključnih riječi grupirani u tematske cjeline koje opisuju dominantne obrasce pogrešaka, poput problema održivosti koda, upravljanja resursima, sigurnosnih propusta i logičkih pogrešaka.

Točnost klasifikacijskog algoritma osigurana je iterativnim postupkom razvoja i provjere. Nakon inicijalne izrade algoritma provedena je ručna validacija slučajnog uzorka od 100 redaka ($N = 100$), pri čemu su rezultati automatske klasifikacije uspoređeni s ljudskom procjenom. Na temelju uočenih neslaganja algoritam je postupno dorađivan (npr. prilagodbom pravila i ključnih riječi te preciziranjem mapiranja kodova u teme), nakon čega je validacija ponavljana na novim slučajnim uzorcima. Taj ciklus dorade i ponovne provjere ponavljan je sve dok nije postignuta vrlo visoka usklađenost automatske i ručne klasifikacije, veća od 99%.

Nakon validacije identificiranih tema provedena je analiza učestalosti tema i njihova povezanost s metrikama tehničkog duga. Za ispitivanje odnosa korištene su neparametrijske statističke metode (Hi-kvadrat test) i usporedbe distribucija pogrešaka, s ciljem utvrđivanja tematskih obrazaca koji su statistički značajno povezani s određenom platformom ili razdobljem

Cjelokupni metodološki postupak omogućuje integraciju kvantitativnih pokazatelja i kvalitativne interpretacije, čime se osigurava sveobuhvatan uvid u obrasce studentskih grešaka i njihov utjecaj na kvalitetu i dugoročno održavanje softvera.

III. REZULTATI

Kako bi se prepoznali učestali obrasci pogrešaka u studentskim radovima, nad kvantitativnim rezultatima statičke analize provedena je kvalitativna tematska analiza, a rezultati i identificirane teme su prikazane u Tablici II. Na temelju opisanog metodološkog okvira, provedena je sveobuhvatna analiza prikupljenih podataka na uzorku od 244 studentska projekta, s ukupno 18.828 detektiranih problema. U nastavku ovog poglavlja, nalazi su strukturirani i prezentirani redoslijedom koji izravno odgovara na prethodno definirana istraživačka pitanja. Prikaz započinje analizom frekvencije i vrste programskih pogrešaka, nakon čega slijedi dublji uvid u specifičnosti platformi i tehnološkog okruženja.

Tablica II
OPIS KATEGORIJA TEHNIČKOG DUGA

Kategorija	**Opis**
Složenost i održivost koda	Složeni tijekovi kontrole, prekomjeran broj parametara metoda te kršenje SOLID načela, što narušava čitljivost i čini kod teškim za izmjenu.
Mrtvi kod i nekorišteni artefakti	Logika koja se nikada ne izvršava, suvišne deklaracije i ostali artefakti koji prikrivaju predviđeni put izvršavanja koda.
Sigurnost i izloženost podataka	Prisutnost vjerodajnica u čistom tekstu, nesigurnih kriptografskih praksi i hardkodiranih metapodataka sustava koji kompromitiraju povjerljivost i integritet aplikacijskog okruženja.
Enkapsulacija i vidljivost	Obuhvaća 'arhitekturno curenje' neispravni modifikatori pristupa i izložena unutarnja stanja koja krše načelo najmanjih privilegija, čineći unutarnju logiku sustava krhkom i teškom za izolirati tijekom testiranja ili refaktoriranja.
Imenovanje i organizacija	Kršenja konvencija imenovanja specifičnih za određeni jezik, arhitekturnih struktura projekta te pravila formatiranja datoteka koja, iako ne utječu na izvršavanje koda, povećavaju kognitivno opterećenje i narušavaju dosljednost koda.
Greške u logici pomičnog zareza	Nesigurno korištenje brojčanih tipova s pomičnim zarezom, što može dovesti do nedeterminističkog ponašanja i nakupljenih pogrešaka pri zaokruživanju.
Manipulacija stringova	Neprimjerno rukovanje znakovnim nizovima i poziva metoda s visokim *overhead-om*, što dovodi do nepotrebne alokacije memorije i smanjene brzine izvršavanja.
Redundancije u kontrolom toku	Nepotrebno komplicirana uvjetna grananja, ručne implementacije značajki koje moderni jezici već podržavaju te nelinearni skokovi u kontroli toka koji povećavaju prostor za nastanak logičkih pogrešaka.
Pogrešna upotreba statičkih el.	Metode koje su nepravilno dodijeljene instancama objekata umjesto da budu statičke, što rezultira nepotrebnim memorijskim troškovima i nejasnim vlasništvom nad podacima.
Neispravna uporaba kolekcija	Neispravna obrada kolekcija i suvišni načini enumeracije koji povećavaju pritisak na memoriju i vrijeme izvršavanja u usporedbi s izvornim svojstvima jezika ili optimiziranim agregatnim funkcijama.
Hardkodirane vrijednosti	Neobjašnjeni numerički literali i ugrađeni znakovni nizovi koji prikrivaju poslovnu logiku i ometaju prenosivost aplikacije, čineći buduća ažuriranja podložnima pogreškama.
Rukovanje datumom i vremenom	Dvosmislen prikaz datuma i vremena, te nedostatak lokalizacije, što može dovesti do oštećenja podataka u višeregionalnim okruženjima i logičkih pogrešaka tijekom prijelaza na ljetno računanje vremena.
Web standardi i pristupačnost	Kršenja semantičkog HTML-a, najboljih praksi CSS-a, *Android* XML-a i standarda pristupačnosti koji narušavaju korisničko iskustvo osobama s invaliditetom i ometaju dosljednost prikaza u različitim preglednicima.
Sigurnost nultih referenci	Tiho rukovanje iznimkama i nepotpune logike za oporavak od pogrešaka koji kompromitiraju stabilnost aplikacije i otežavaju analizu uzroka kvara.
Upravljanje resursima	Nepravilno oslobađanje resursa, što dovodi do curenja memorije i nestabilnosti sustava.
Asinkrono izvršavanje	Blokirajući pozivi, nesigurni pristupa dijeljenom stanju i sl., što može dovesti do *deadlock*-a i neodzivnih korisničkih sučelja.

A. IP1: *Koje su najčešće programske pogreške koje se pojavljuju u studentskim projektima na završnoj godini prijediplomskog studija?*

Distribucija po tematskim kategorijama prikazana u Tablici III i na Slici 1, pokazuje izraženu koncentraciju pogrešaka u relativno malom broju kategorija, pri čemu nekoliko dominantnih tema čini većinu svih zabilježenih problema. Najzastupljenija kategorija pogrešaka odnosi se na *mrtvi kod i nekorištene artefakte* (4562; 24,23%). Slijede pogreške povezane s *pogrešnom upotrebom statičkih i objektnih članova* (3009; 15,98%) te *redundancije i dupliciranje koda* (1814; 9,63%). U prvih pet kategorija, uz navedene, nalaze se i *enkapsulacija i vidljivost* (1483; 7,88%) te *sigurnost nultih referenci i upravljanje greškama* (1475; 7,83%).

Tablica III
PREGLED UČESTALOSTI GREŠAKA PO KATEGORIJAMA

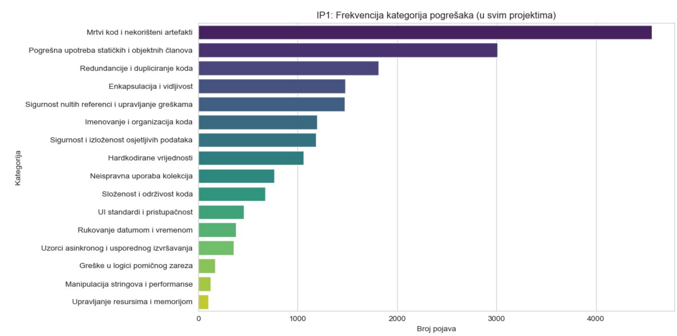
Kategorija	Broj
Mrtvi kod i nekorišteni artefakti	4562
Pogrešna upotreba statičkih i objektnih članova	3009
Redundancije i dupliciranje koda	1814
Enkapsulacija i vidljivost	1483
Sigurnost nultih referenci i upravljanje greškama	1475
Imenovanje i organizacija koda	1197
Sigurnost i izloženost osjetljivih podataka	1184
Hardkodirane vrijednosti	1063
Neispravna uporaba kolekcija	769
Složenost i održivost koda	676
UI standardi i pristupačnost	460
Rukovanje datumom i vremenom	378
Uzorci asinkronog i usporednog izvršavanja	358
Greške u logici pomičnog zareza	170
Manipulacija stringova i performanse	128
Upravljanje resursima i memorijom	102

Kumulativno, prvih pet kategorija obuhvaća 65,56% svih detektiranih pogrešaka, što upućuje na to da većina uočenih problema proizlazi iz ograničenog skupa ponavljajućih obrazaca. Dodatno, deset najčešćih kategorija obuhvaća 91,52% ukupnog broja pogrešaka, čime se potvrđuje da se preostali udio pogrešaka raspoređuje na velik broj rjeđe zastupljenih kategorija. Ovakav raspored opravdava fokus daljnje analize na najfrekventnije kategorije, jer one nose najveći informacijski doprinos pri opisu tipičnih pogrešaka u studentskim projektima.

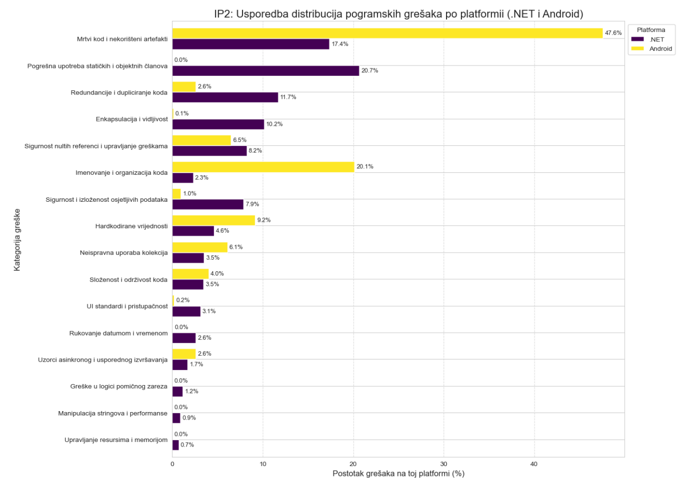
B. IP2: *Postoji li statistički značajan utjecaj razvojne platforme/tehnologije (.NET u odnosu na Android) na učestalost pojedinih vrsta programskih pogrešaka?*

Proveden je Hi-kvadrat test neovisnosti kako bi se utvrdilo ovisi li vrsta programskih pogrešaka o korištenoj razvojnoj platformi (.NET naspram *Android*-a). S obzirom na to da je test rezultirao p-vrijednošću manjom od $0,001$, postoje snažni dokazi za odbacivanje nulte hipoteze, što potvrđuje da je distribucija vrsta pogrešaka statistički značajno ovisna o platformi.

Vizualna analiza podataka na Slici 2 otkriva nerazmjerno visoku učestalost pogrešaka vezanih uz *mrtvi kod* i *UI standarde* na *Android* platformi. To je vjerojatno posljedica prirode razvoja *Android* aplikacija (upravljanje događajima), gdje studenti često ostavljaju nekorištene *event handler*-e ili pogrešno konfiguriraju XML datoteke. Nasuprot tome, kod .NET platforme prevladavaju pogreške vezane uz *enkapsulaciju* i *pogrešnu upotrebu statičkih članova*. Ovo odražava naglasak platforme na strožim objektno-orijentiranim obrascima, pri čemu studenti češće imaju poteškoća s pravilnom upotrebom modifikatora pristupa i statičkih ključnih riječi.



Slika 1. Frekvencija kategorija pogrešaka.



Slika 2. Distribucija programskih grešaka po platformi.

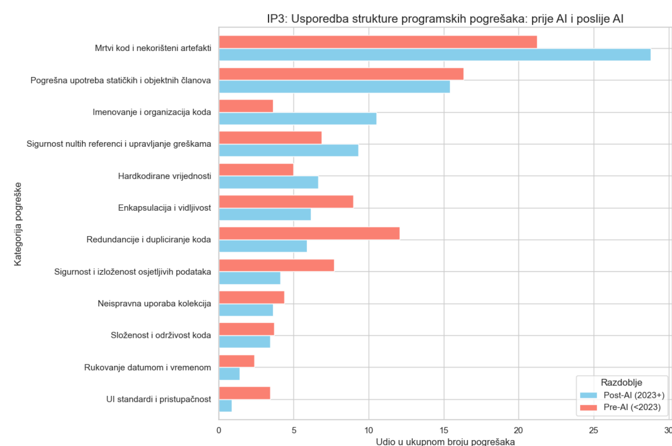
Izračunata veličina učinka Cramerov V ($V = 0,53$) ukazuje na snažnu povezanost varijabli i sugerira da izazovi čistog koda nisu univerzalni, već su uvelike uvjetovani kontekstom. Platforma se ovdje ne pokazuje samo kao sporedni faktor, već kao dominantan uzrok koji diktira vrstu pogrešaka. Primjerice, *Android* okvir svojom strukturom čini studente sklonijima specifičnim problemima, poput mrtvog koda u UI *handler*-ima ili složenosti asinkronih metoda, koji su znatno manje zastupljeni u standardnom .NET razvoju.

C. IP3: *Postoji li značajna razlika u profilu programskih pogrešaka u razdoblju prije šire dostupnosti generativnih UI alata i nakon nje?*

U okviru IP3 analizirano je postoji li razlika u strukturi detektiranih problema između razdoblja "prije UI" i "poslije

UI”, pri čemu se “poslije UI” odnosi na 2023. i 2024. (nakon šire dostupnosti generativnih alata), a “prije UI” se odnosi na 2020, 2021. i 2022. Polazna motivacija za ovo pitanje je pretpostavka da se s promjenom načina rada (npr. uz pomoć generativnih alata) može promijeniti i profil pogrešaka koje ostaju u završnim verzijama studentskih projekata.

Analiza podataka u okviru IP3 prikazana na Slici 3 ukazuje na statistički značajnu promjenu u strukturi programskih pogrešaka prije i nakon 2023. godine, koja se uzima kao prijelomna točka za široku primjenu generativnih UI alata u obrazovanju. U razdoblju nakon 2023. godine zabilježen je primjetan porast udjela pogrešaka unutar kategorija *mrtvi kod i nekorišteni artefakti* te *imenovanje i organizacija koda*. Ovakav trend sugerira da studenti sve češće integriraju strojno generirana rješenja koja, iako sintaktički ispravna, često sadrže suvišne elemente (poput nekorištenih biblioteka) ili ne slijede specifične konvencije imenovanja definiranih projektom. Nasuprot tome, razdoblje prije 2023. godine karakterizira veća učestalost kategorija *redundancije i dupliciranja koda* te *sigurnosti i izloženosti osjetljivih podataka*. Smanjenje ovih pogrešaka u novijem razdoblju može se pripisati sposobnosti UI asistenata da generiraju sažetiji kod i izbjegnu ručno dupliciranje logike, čime se težište problema pomiče s pisanja *boilerplate* koda na integraciju i čišćenje generiranih artefakata.



Slika 3. Usporedba strukture programskih pogrešaka prije UI i poslije UI

IV. Ograničenja istraživanja

Istraživanje podliježe određenim ograničenjima koja je potrebno uzeti u obzir pri interpretaciji rezultata. Prvo, uzorak je ograničen na studentske projekte jedne visokoškolske institucije, što može utjecati na mogućnost generalizacije zaključaka na širu populaciju studenata kolegija u domeni programskog inženjerstva. Drugo, iako je veličina uzorka od 244 projekta statistički relevantna, promjene u izvedbenim planovima kolegija tijekom promatranih pet godina mogle su unijeti varijacije u zahtjevima projekata, što je moglo djelomično utjecati na vrste detektiranih pogrešaka neovisno o tehnološkim faktorima. Treće, podjela na razdoblja *prije* i *poslije* popularizacije generativnih UI alata temelji se na vremenskom presjeku, a ne na izravno potvrđenoj upotrebi alata od strane pojedinih studenata. Štoviše, istraživanje pretpostavlja i rezultatima potvrđuje da su studenti na projektima izrađenim tijekom posljednje dvije ak. godine koristile alate generativne umjetne inteligencije. Konačno, metodologija se oslanja na alate za statičku analizu koda koji, iako precizni u detekciji tehničkog duga i potencijalnih pogrešaka, ne mogu detektirati sve semantičke ili logičke pogreške u poslovnoj logici aplikacija koje ne krše sintaksna pravila jezika.

V. Zaključak

Analiza tehničkog duga i programskih pogrešaka na studentskim projektima ukazuje na to da su problemi održivosti, prvenstveno *mrtvi kod* i *pogrešna primjena statičkih članova*, znatno učestaliji od kritičnih funkcionalnih *bug*-ova. Istraživanje je potvrdilo da tehnološka platforma značajno diktira vrstu pogrešaka s kojima se studenti bore; dok .NET okruženje češće rezultira strukturnim problemima vezanim uz objektno-orijentirane principe, razvoj na *Android*-u podložniji je nakupljanju nekorištenih artefakata i nepravilnostima vezanim uz UI.

Najvažniji uvid proizlazi iz usporedbe razdoblja prije i poslije 2023. godine, koje u ovom radu koristimo kao aproksimaciju šire dostupnosti generativnih UI alata u obrazovanju. Uočena promjena od pogrešaka redundancije i dupliciranja prema pogreškama integracije, imenovanja i nekorištenog koda sugerira da se s promjenom načina rada mijenja i profil tehničkog duga u završnim verzijama studentskih projekata. Ti nalazi upućuju na potrebu za unaprjeđenjem kurikuluma programskog inženjerstva, s pomakom fokusa s pisanja sintakse prema vještinama kritičke revizije, refaktoriranja i arhitektonskog razumijevanja te održavanja rješenja. U tom smislu, *post-course* analiza finalnih verzija projekata može služiti kao praktičan instrument za planiranje ciljnih nastavnih intervencija i provjera znanja usmjerenih na kvalitetu i održivost koda.

Napomena

Ovo istraživanje provedeno je u okviru šireg istraživanja na projektu SQUAD: Softversko inženjerstvo u doba umjetne inteligencije, kvantnog računarstva i drugih disruptivnih tehnologija (FOI-IIP-2025-13), koji financira Europska unija – NextGenerationEU.

Literatura

[1] M. Kuutila, M. Mäntylä, U. Farooq i M. Claes, „Time pressure in software engineering: A systematic review,” *Information and Software Technology*, sv. 121, str. 106 257, 2020., ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2020.106257

[2] D. Basten, M. Müller, M. Ott, O. Pankratz i C. Rosenkranz, „Impact of time pressure on software quality: A laboratory experiment on a game-theoretical model,” *PLOS ONE*, sv. 16, br. 1, str. 1–21, siječanj 2021. DOI: 10.1371/journal.pone.0245599

[3] P. Kruchten, R. L. Nord i I. Ozkaya, „Technical Debt: From Metaphor to Theory and Practice,” *IEEE Software*, sv. 29, br. 6, str. 18–21, 2012. DOI: 10.1109/MS.2012.167

[4] C. Szabo, „Student projects are not throwaways: teaching practical software maintenance in a software engineering course," *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, serija SIGCSE '14, Atlanta, Georgia, USA: ACM, 2014., str. 55–60, ISBN: 9781450326056. DOI: 10.1145/2538862.2538965

[5] P. C. Avgeriou i dr., „An Overview and Comparison of Technical Debt Measurement Tools," *IEEE Software*, sv. 38, br. 3, str. 61–71, 2021. DOI: 10.1109/MS.2020.3024958

[6] M. T. Baldassarre, V. Lenarduzzi, S. Romano i N. Saarimäki, „On the diffuseness of technical debt items and accuracy of remediation time when using SonarQube," *Information and Software Technology*, sv. 128, str. 106 377, 2020., ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2020.106377

[7] S. Nocera, S. Romano, R. Francese i G. Scanniello, „Software engineering education: Results from a training intervention based on SonarCloud when developing web apps," *Journal of Systems and Software*, sv. 222, str. 112 308, 2025., ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2024.112308

[8] M. De Luca, S. Di Meglio, A. R. Fasolino, L. L. L. Starace i P. Tramontana, „Automatic Assessment of Architectural Antipatterns and Code Smells in Student Software Projects," *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, serija EASE '24, Salerno, Italy: ACM, 2024., str. 565–569. DOI: 10.1145/3661167.3661290

[9] M. Horvath, E. Pietrikova i J. Sasak, „Improving Software Education with Technical Debt Analysis Tool," *2024 ICETA*, Stary Smokovec, Slovakia: IEEE, listopad 2024., str. 187–192. DOI: 10.1109/ICETA63795.2024.10850816

[10] V. Braun i V. Clarke, „Using thematic analysis in psychology," *Qualitative Research in Psychology*, sv. 3, br. 2, str. 77–101, 2006. DOI: 10.1191/1478088706qp063oa

[11] S. Nocera, S. Romano, R. Francese i G. Scanniello, „Training for Security: Results from Using a Static Analysis Tool in the Development Pipeline of Web Apps," *Proceedings of the ICSE-SEET '24)*, Lisbon, Portugal: ACM, 2024., str. 253–263. DOI: 10.1145/3639474.3640073

[12] E. D. Berger, C. Hollenbeck, P. Maj, O. Vitek i J. Vitek, „On the Impact of Programming Languages on Code Quality: A Reproduction Study," *ACM Trans. Program. Lang. Syst.*, sv. 41, br. 4, str. 1–24, prosinac 2019. DOI: 10.1145/3340571

[13] D. Sun, A. Boudouaia, C. Zhu i Y. Li, „Would ChatGPT-facilitated programming mode impact college students' programming behaviors, performances, and perceptions? An empirical study," *International Journal of Educational Technology in Higher Education*, sv. 21, br. 1, 2024. DOI: 10.1186/s41239-024-00446-5

[14] R. Yilmaz i F. G. Karaoglan Yilmaz, „The effect of generative artificial intelligence (AI)-based tool use on students' computational thinking skills, programming self-efficacy and motivation," *Computers and Education: Artificial Intelligence*, sv. 4, str. 100 147, 2023., ISSN: 2666-920X. DOI: https://doi.org/10.1016/j.caeai.2023.100147

[15] D. Cotroneo, C. Improta i P. Liguori, *Human-Written vs. AI-Generated Code: A Large-Scale Study of Defects, Vulnerabilities, and Complexity*, arXiv:2508.21634, kolovoz 2025. DOI: 10.48550/arXiv.2508.21634

[16] „Static Analysis Results Interchange Format (SARIF) Version 2.1.0," OASIS, Standard, ožujak 2020.

[17] A. Martini, „Anacondebt: a tool to assess and track technical debt," *Proceedings of the 2018 International Conference on Technical Debt*, Gothenburg, Sweden: ACM, svibanj 2018., str. 55–56. DOI: 10.1145/3194164.3194185

[18] Z. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo i S. Li, „SATD detector: a text-mining-based self-admitted technical debt detection tool," *Proceedings of the 40th ICSE: Companion Proceeedings*, Gothenburg, Sweden: ACM, svibanj 2018., str. 9–12. DOI: 10.1145/3183440.3183478

[19] T. S. Mendes, F. G. S. Gomes, D. P. Gonçalves, M. G. Mendonça, R. L. Novais i R. O. Spínola, „VisminerTD: a tool for automatic identification and interactive monitoring of the evolution of technical debt items," *J Braz Comput Soc*, sv. 25, br. 1, str. 2, prosinac 2019. DOI: 10.1186/s13173-018-0083-1

[20] J.-L. Letouzey, „The SQALE method for evaluating Technical Debt," *2012 Third International Workshop on Managing Technical Debt (MTD)*, Zurich, Switzerland: IEEE, lipanj 2012., str. 31–36. DOI: 10.1109/MTD.2012.6225997

[21] S. Prokic i dr., „Clean Code and Design Educational Tool," *MIPRO 2021*, Opatija, Croatia: IEEE, rujan 2021., str. 1601–1606. DOI: 10.23919/MIPRO52101.2021.9597196

Abstract

In the field of software engineering education, tight deadlines and a focus on functionality often lead to the accumulation of technical debt in student projects. This paper analyzes 244 student projects created in the final year of undergraduate studies over a five-year period (2020–2024), using static code analysis tools. The aim of the research was to identify the most common error patterns and to determine the impact of the development platform (.NET and Android) and the period following the wider availability of generative AI tools on code quality. The results show a statistically significant dependence of error types on the platform, with Android projects suffering from an excess of unused code, while .NET projects show structural deficiencies in the application of object-oriented principles. A particularly significant finding is the change in the error profile after 2023, where a decrease in code redundancy is observed, alongside an increase in integration problems and 'dead' code. The paper concludes that modern curricula should shift focus from writing code to the skills of reviewing and maintaining solutions, including those created with the help of generative artificial intelligence.

ANALYSIS OF TECHNICAL DEBT AND SOFTWARE BUG PATTERNS IN STUDENT PROJECTS: THE IMPACT OF PLATFORM AND THE POST-GENAI AVAILABILITY PERIOD

Zlatko Stapić, Dario Ljubas, Lovro Posarić