Uloga mjerenja kompleksnosti u osiguranju kvalitete softvera

Merlić, Nikola

Sveučilište Jurja Dobrile u Puli
Tehnički fakultet u Puli

Nikola Merlić

Uloga mjerenja kompleksnosti u osiguranju kvalitete softvera

Završni rad

Pula, rujan, 2022. godine.

Sveučilište Jurja Dobrile u Puli
Tehnički fakultet u Puli

Nikola Merlić

Uloga mjerenja kompleksnosti u osiguranju kvalitete softvera

Završni rad

JMBAG:0303090413, redoviti student
Studijski smjer: Računarstvo
Predmet: Kvaliteta u informacijsko – komunikacijskoj tehnologiji
Znanstveno područje: Područje tehničkih znanosti
Znanstveno polje: Računarstvo
Znanstvena grana: Programsko inženjerstvo

Mentor: Izv. prof. dr. sc. Giorgio Sinković

Pula, rujan, 2022. godine

Tehnički fakultet u Puli

Ime i prezime studenta/ice Nikola Merlić

JMBAG 0303090413

Status: ☒ redoviti ☐ izvanredni

PRIJAVA TEME ZAVRŠNOG RADA

Giorgio Sinković

Ime i prezime mentora

Računarstvo

Studij

Kvaliteta u informacijsko-komunikacijskoj tehnologiji

Kolegij

Potvrđujem da sam prihvatio/la temu završnog/diplomskog rada pod naslovom:

Uloga mjerenja kompleksnosti u osiguranju kvalitete softvera

(na hrvatskom jeziku)

Complexity measurements as tools for software quality assurance

(na engleskom jeziku)

Datum: 13.4.2022.

IZJAVA O AKADEMSKOJ ČESTITOSTI

Ja, dolje potpisani **Nikola Merlić** , kandidat za prvostupnika računarstva ovime izjavljujem da je ovaj Završni rad rezultat isključivo mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na objavljenu literaturu kao što to pokazuju korištene bilješke i bibliografija. Izjavljujem da niti jedan dio Završnog rada nije napisan na nedozvoljen način, odnosno da je prepisan iz kojega necitiranog rada, te da ikoji dio rada krši bilo čija autorska prava. Izjavljujem, također, da nijedan dio rada nije iskorišten za koji drugi rad pri bilo kojoj drugoj visokoškolskoj, znanstvenoj ili radnoj ustanovi.

Student

U Puli, 6.09., 2022. godine

IZJAVA
o korištenju autorskog djela

Ja, **Nikola Merlić** dajem odobrenje Sveučilištu Jurja Dobrile u Puli, kao nositelju prava iskorištavanja, da moj Završni rad pod nazivom „Uloga mjerenja kompleksnosti u osiguranju kvalitete softvera“ koristi na način da gore navedeno autorsko djelo, kao cjeloviti tekst trajno objavi u javnoj internetskoj bazi Sveučilišne knjižnice Sveučilišta Jurja Dobrile u Puli te kopira u javnu internetsku bazu završnih radova Nacionalne i sveučilišne knjižnice (stavljanje na raspolaganje javnosti), sve u skladu s Zakonom o autorskom pravu i drugim srodnim pravima i dobrom akademskom praksom, a radi promicanja otvorenoga, slobodnoga pristupa znanstvenim informacijama.
Za korištenje autorskog djela na gore navedeni način ne potražujem naknadu.

U Puli, 6.09.2022.

Potpis

Sadržaj

1.Uvod

Mjerenje kompleksnosti softvera zapravo je jedan od glavnih čimbenika koji osigurava razinu određene kvalitete te diktira daljnji razvitak samog proizvoda u financijskom te vremenskom smislu. Developeri ili programeri, puno vremena izdvajaju na pisanje samog programskog koda, no isto tako još više vremena troše na održavanje koda. Tijekom održavanja često su u situaciji da ne znaju kako radi i zbog čega su implementirali na određeni način, te postavljaju pitanja kako i zašto. Jedan od najčešćih krivaca tome je kompleksnost softvera, ovisno o korištenom programskom jeziku kod može postati kompleksan do granice da ni sam programer ne može predvidjeti njegovo djelovanje prema okolini. Naravno, kompleksnost softvera nije prva stvar koja programeru treba privući svu pažnju, veliku ulogu u samome radu projekta igraju određeni programski jezici koje programer odabire ovisno o vrsti i namjeni proizvoda te okruženje u kojem će program raditi ili je to nametnuo kupac. Kako bi osigurali što veću kvalitetu poželjno je ranije početi razmišljati o široj slici projekta pa i o samoj kompleksnosti. Što je veće poznavanje o kompleksnosti softvera, moguće je povećati i kvalitetu samog programskog koda tj. bit će podložan rjeđim greškama tijekom faza testiranja te kasnije tijekom korištenja softvera. Kompleksnost softvera može se definirati kao pojam koji uključuje mnoge dijelove softvera koji imaju nekakvu međusobnu povezanost te interakciju između sebe. Bitno je razlikovati dva međusobno različita pojma u ovome kontekstu a to su: kompliciranost te kompleksnost. Kompliciranost označava nešto što je teško shvatljivo, no s vremenom i trudom može se shvatiti, kompleksnost opisuje interakcije između određenog broja entiteta, kako raste broj entiteta eksponencijalno može rasti i broj interakcija između njih te može narasti do granice gdje je nemoguće predvidjeti sve međusobne interakcije. U nekim ekstremnim slučajevima velika kompleksnost softvera može ograničiti mogućnosti modificiranja projekta što može dovesti do kraja životnog vijeka softverskog proizvoda. Kako bi spriječili takve slučajeve, developeri koriste brojne metode pomoću kojih mjere kompleksnost te osiguravaju kvalitetu proizvoda.

2.Softverska kompleksnost

Kompleksnost softvera pojam je koji uključuje mnogo svojstva nekog dijela softvera, koji utječu na unutarnje interakcije. Jednoznačna i nedvosmislena definicija kompleksnosti ne postoji nego je za njeno razumijevanje korisno poznavati načine mjerenja. Definicije navode više tipova kompleksnosti: esencijalna i slučajna. Esencijalna je ona kompleksnost koja je nužna da bi se zadovoljile zadane funkcionalnosti softvera, a slučajna je ona koja je rezultat propusta u dizajnu ili nedovoljnog upravljanja kompleksnosti. Drukčije formulirana definicija opisuje kompleksnost kao interakcije koje se odvijaju između broja entiteta u programskome proizvodu. Kako se broj entiteta povećava, eksponencijalno se povećava broj interakcija te može doći do određene veličine gdje ne bi bilo moguće imati pregled nad programom u cijelosti. Veća kompleksnost povećava rizik interferencija unutar softverskog proizvoda te povećava mogućnost razvijanja defekata unutar programa, čak u nekim slučajevima kompleksnost može doći do te razine gdje modificiranje softvera postaje nemoguće. Korištenjem metoda koje su kasnije navedene ukazuju na neke mogućnosti nadzora kompleksnosti.

3.Metrike kompleksnosti

Kompleksnost kao i produktivnost mogu se mjeriti na puno načina koji ovise o puno parametara. Jedna od najjednostavnijih načina mjerenja kompleksnosti može biti brojanje linija koda. Kada se govori o samoj produktivnosti programera, kod ove metode može se zaključiti kako što je projekt kompleksniji to će programer zapravo proizvesti manje linija koda, proporcionalno tome što je projekt jednostavniji to će programer proizvesti više linija koda. Kompleksnost ima veliku ulogu u cijeni održavanja projekta te isto tako što je projekt kompleksniji pravilo je kako će i troškovi biti veći, zbog toga bitno je računati o tome na vrijeme te koristiti neke od metoda kako bi dobili što precizniji rezultat. Neke od metoda koje se koriste su: “Size Measures“, “Halstead's Measure“, “Halstead's Complexity Measure“, “Live Variables“, “Knot Count“, “Topological Complexity“, “lines of code“, “Cyclomatic complexity“…

4.Tipovi strukturnih kompleksnosti

Veza između kompleksnosti softvera, veličine te produktivnosti nije tako jednostavno objašnjiva, ako se pretpostavi da su svi drugi aspekti programa ili modula jednaki, velikom modulu treba više vremena za kodiranje, test, dizajn i specificiranje od manjeg modula, no u ovome aspektu vrsta strukturne složenosti igra veliku ulogu u pronalaženju te učestalosti mogućih grešaka. Strukturna kompleksnost može se promatrati sa više gledišta kao što su: struktura kontrolnog tijeka, struktura tijeka podataka i podatkovna struktura.

4.1.1 Struktura kontrolnog tijeka

Mjerenja kontrolnih tijekova programa počela su se provoditi tijekom ranih 1970. godina. Bili su to zapravo prvi načini mjerenja atributa za kompleksnost softvera. Modelirani su uz pomoću usmjerenih grafova, gdje svaki čvor odgovara određenoj izjavi programa te strelice između izjava označavaju kontrolni tijek između izjava. U doslovnome prijevodu izjava označava sintaktičku jedinicu koja izražava neku akciju koja se treba izvršiti unutar programa. Svaki graf sadržava određeni broj čvorova koji su spojeni. Kretanje između čvorova je moguće dokle god se prate putanje strelica između čvorova. Korištenjem slijeda, selekcije te iteracije može se konstruirati program, u današnje vrijeme postoji puno automatiziranih programa koji mogu izvući iz programskog koda strukturu pomoću koje je kasnije moguće izvući jednadžbu mjerenja kompleksnosti. Metoda koja koristi kontrolne tijekove zove se McCabe'ova ciklomatska kompleksnost.

4.1.2 Struktura tijeka podataka

Mjerenje tijeka podataka naglašuje veze između više određenih dijelova programa ili modula. Zapravo cilj promatranja ove strukture je kolekcija dijelova programa, neovisno

u kojoj fazi, može biti u početnoj fazi dizajna pa sve do faze implementacije. U ovome kontekstu, modul se interpretira kao objekt koji se promatra kao jedinstveni konstrukt neovisno o veličini njegove apstrakcije, on stoji zasebno te neovisno o drugim modulima.

4.1.3 Strukture podataka

Struktura podataka je mjesto koje služi za držanje te organiziranje podataka. Razlikuju se više tipova struktura podataka kao što su linearne strukture podataka u koje se ubrajaju liste, vezane liste, stog, …, pod nelinearne strukture podataka spadaju: stabla i grafovi, te podaci unutar struktura mogu biti dinamički što znači da im je veličina memorije zadana unaprijed tj. nepromjenjiva ili dinamičke u kojima nemaju fiksnu memoriju. U pravilu lakše je pristupiti elementima statičnog tipa podataka za razliku od dinamičkih no dinamički tipovi podataka smanjuju kompleksnost samog programa. (GeeksforGeeks, 2022.)

U prijašnjim točkama objašnjeno je kako zapravo informacijski tijek ili tijek podataka se može mjeriti, kako bi postojao uvid o interakciji podataka unutar modula. Fokus ovog dijela su zapravo podaci kao atomske komponente određenog programa. Poanta ovoga je da se za svaki dani sustav identificiraju konstrukti od kojih je rađen program te da ih se prikladno zbroji ovisno o njihovoj „težini“ tako da prikažu kompleksnost sustava u kojem se nalaze. Što se više konstrukta koristi u programu to će kompleksnost programa i rasti, ako koristimo puno različitih konstrukta kompleksnost će još više rasti. Zbog toga mjerenje tijeka podataka i kontrolnog tijeka podataka nije dosta jer se kompleksnost može skrivati unutar strukture podataka koju ne bi bilo moguće pronaći korištenjem prije nabrojanih načina (Nystedt, S., Sandros, C., 1999.).

4.2 Algoritamska kompleksnost

Algoritamske kompleksnosti reflektiraju kompleksnost algoritma koji je namijenjen za rješavanje zadanog problema. Naravno, što je problem apstraktniji ili je složeniji postupak za njegovo rješenje, mogu se koristiti složeniji algoritmi koje je teže razumjeti i

implementirati. U širem smislu kompleksnost algoritma bavi se sa kompleksnošću opsega što znači sa veličinom softvera i sa volumenom i složenosti algoritma.

Pomoću algoritamske kompleksnosti mjeri se koliko dugo bi trebalo određenom algoritmu da se izvrši ako mu je dan "input" veličine n, njegov rezultat bi uvijek trebao biti konačan te praktičan čak i za velike količine danih podataka. Kompleksnost se većinski mjeri u smislu vremena, nekada to jednostavno nije dovoljno te treba razmišljati i u smislu memorije koje zauzima. Algoritamska kompleksnost grana je teorijske računarske znanosti imenom računarska teorija kompleksnosti. Ako korisnik traži specifičnu stvar unutar liste koja nije sortirana, vjerojatno će uspoređivati svaku stvar unutar liste sa svakom, što znači da je vrijeme traženja proporcionalno veličini liste tj. kompleksnost je linearna (Clobo, 2017.). Sa druge strane, ako korisnik traži riječ u rječniku, traženje će biti puno brže zbog toga jer su riječi već sortirane logičnim redoslijedom što daje prednost pri traženju, takva vrsta kompleksnosti naziva se logaritamska (Clobo, 2017.). Ako se traži od nekoga da izabere prvu riječ u rječniku, operacija će biti konstantno vremenske kompleksnosti, neovisno o broju te dužini riječi unutar rječnika. Slično, ako korisnik mora platiti račun u banci pravilo je da red radi po LI.LO metodi tj. "last in last out", no neovisno o dužini samog reda kompleksnost je i dalje konstantna. Ako korisniku dana ne sortirana lista te u njoj je potrebno pronaći sve duplikate, onda kompleksnost postaje kvadratne veličine. Traženje duplikata za jednu stvar unutar liste sadržava linearnu kompleksnost, no ako su u pitanju sve stvari unutar liste kompleksnost postaje kvadratna (Clobo 2017.). Velika O notacija reprezentira kompleksnost algoritma. Notacija algoritma prikazuje gornju granicu složenosti tj. prikazuje izvedbu algoritma u najgorem slučaju. Notacija se koristi pri uspoređivanju više različitih algoritma te pomoću grafa lako je iščitati kako se algoritmi skaliraju kada se veličina ulaza ili "inputa" povećava. „Konstantno izvođenje reprezentirano je sa O(1); linearni rast je O(n); logaritamski rast je prikazan kao O(log n); logaritamsko-linearni rast je O(n log n); kvadratni rast je O(n^2); eksponencijalni rast prikazuje se kao O(2^n) te rast faktorijela O(n!)"(Devopedia, 2022.). Rast navedenih algoritama moguće je usporediti od najboljeg prema najgorem:

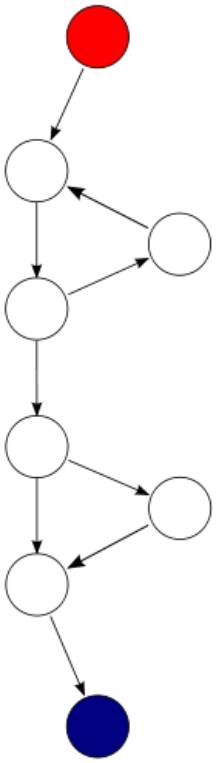
$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(10^n)$$

Razlikuju se tri moguća scenarija kod izračunavanja kompleksnosti algoritma: najbolji slučaj, najgori slučaj i prosječni slučaj. Pomoću algoritma traži se u nesortiranoj listi neku određenu stvar, ona može biti bilo gdje unutar te liste, najbolji slučaj je ako se taj podatak, stvar ili objekt nalazi na prvome mjestu, onda je kompleksnost O(1), najgori slučaj reprezentira ako se on nalazi na samome kraju nesortirane liste te kompleksnost u tom slučaju je O(n). Prosječni slučaj kompleksnosti iznosi O(n), no kada se priča u smislu kompleksnosti većinom se uzima onaj najgori mogući slučaj koji algoritam može postići (Vilches, 2015). Umjesto vremenskog praćenja koliko pojedinome algoritmu treba da se izvrši, moguće je evaluirati broj kompleksnijih instrukcija koje su povezane sa veličinom "inputa". Petlja koja prolazi kroz "input" ili ulaz je linearna, ako postoji petlja unutar petlje kompleksnost algoritma raste na kvadratnu neovisno ako algoritam procesira alternativne stvari ili preskače fiksni broj stvari, kompleksnost isto tako ignorira konstante te faktore skaliranja. Petlja koja se nalazi unutar petlje koja je omeđena još jednom petljom je isto tako kompleksno kvadratna (Zindros, 2012.). Rekurzivna funkcija koju je moguće pozvati n broj puta je linearna, naravno ako ostale operacije koje se nalaze unutar funkcije ne ovise o veličini ulaza, ali zato implementacija Fibonaccijevog broja je eksponencijalna (Nakov, 2013.). Algoritam pretraživanja dijeli ulaz na dva dijela te odbacuje jedan od njih pri svakoj iteraciji, njegova kompleksnost je logaritamska, "mergesort" dijeli ulaz na pola pri svakoj iteraciji te radi operaciju spajanja u linearnom vremenu, njegova kompleksnost spada pod logaritamsko-linearnu kompleksnost (Zindros 2012.). Kompleksnosti algoritama često se znaju zanemarivati, pretpostavka je kako par milisekundi razlike nije dovoljan razlog za promjenu algoritma iako bi možda drukčija implementacija algoritma zapravo bila puno efikasnija, manje kompleksnija te ultimativno u dugoročnome smislu uštedila organizaciji veliku svotu novaca i vremena.

4.3. Ciklomatska kompleksnost

Ciklomatska kompleksnost jedna je od metrika softvera pomoću koje se indicira kompleksnost programa. Koristi se kao kvantitativno mjerenje broja linearno neovisnih puteva kroz programski kod. Razvijena je 1976. godine od znanstvenika Thomas J. McCabe-a. Složenost metode računa se pomoću upravljačkog tijeka programa: čvorovi

grafa odgovaraju nedjeljivim skupinama naredbi programa, a usmjereni rub povezuje dva čvora ako se druga naredba može pokrenuti odmah nakon prve naredbe. Ciklomatska primjenjuje se na pojedinačne funkcije, module, metode ili klase unutar programa.



Slika 4.3. – tijek grafa jednostavnog programa (Anon, 2022., https://upload.wikimedia.org/wikipedia/commons/2/2b/Control_flow_graph_of_function_with_loop_and_an_if_statement_without_loop_back.svg)

Na slici 4.3. prikazan je jednostavan tijek programa koji se počinje izvoditi na crvenome krugu (čvoru), nakon početka ulazi u petlju (petlja se reprezentira u ovome slučaju kao grupa od tri čvora). Nakon prolaska kroz graf zaključuje se kako on ima 9 rubova (rubove reprezentiraju strelice koje pokazuju na čvorove), 8 čvorova i 1 spojenu komponentu. Komponenta je spojeni graf koji nije fizički povezan sa ostalim grafovima već je nezavisan, komponentu grafova svrstava se pod područje proučavanja teorije grafova. Kada su parametri određeni može se izračunati ciklomatska kompleksnost, u ovome slučaju kompleksnost je: 9 – 8 + 2*1 = 3.

Ciklomatsku kompleksnost računa se po slijedećoj formuli:

$$M = V(G) = e - n + 2p$$

Gdje je: V(G) – ciklomatski broj

e – broj rubova

n – broj čvorova

p – broj nepovezanih dijelova na grafu

Kompleksnost više grafova smatra se kao grupa koja je jednaka sumi individualno zbrojenih grafova tj. zbroj njihovih individualnih kompleksnosti. Navedena metoda sadržava poveću manu, ne može diferencirati razlike između petlja i “statement-a“. Kako bi testiranje i održavanje bilo uspješno, McBabe preporuča kako kompleksnost ne bi smjela prekoračiti broj 10. Baziranost metrika svodi se na grane i odluke koje se sastoje od logičkih uzoraka implementiranih u dizajnu i programiranju tijekom izrade projekta . Prelaskom preko kompleksnosti broja 10 projekt postaje previše ovisan i težak za održavanje do granica neisplativosti.

4.4. Halsteadov-e mjere kompleksnosti

“Halstead complexity measure“ ili Halsteadov-a mjera kompleksnosti softverska je metrika izumljena 1977. godine od strane Maurice Howard Halstead-a. Opservacija softverske metrike reflektira te uzima u obzir implementaciju algoritama u različitim programskim jezicima, no izvođenje se odvija na uvijek istom kontroliranom okruženju. Cilj ove metrike svođenje je kompleksnosti na manje mjerljive pojmove te moguće relacije između pojmova. Operatori i operandi koriste se za definiranje mjerljivih pojmova, pomoću navedenog dobivaju se mjere kao što su dužina, volumen, težina, vokabular, greške u programu, vrijeme testiranja i trud programiranja tj. ljudski faktor. “You can't manage what you can't measure.“ stara je poslovica Peter-a Druckera koja je i dan danas validna. Halstead, M. H. definira sljedeće spomenute jednadžbe i varijable koje omogućuju računanje kompleksnosti:

n1 = broj različitih operatora

n2 = broj različitih operanda

N1 = konačan broj operatora

N2 = konačan broj operanda

Veličinu vokabulara dobijemo na način:

$$n = n1 + n2$$

Duljina programa smatra se kao zbroj pojavljivanja operatora i operanda u programu:

$$N = N1 + N2$$

Izračunata procijenjena duljina programa:

$$\mathrm{N^{\wedge}} = \mathrm{n1log_2n1} + \mathrm{n2log_2n2}$$

Volumen programa proporcionalan je samoj veličini programa, reprezentira veličinu u bitovima, no vrlo je ovisan o implementaciji algoritma koristi. Formula volumena je sljedeća:

$$v = N \cdot log\, n$$

Težina programa označava koliko je program razumljiv:

$$D = \frac{n_1}{2} * \frac{N_2}{n_2}$$

Trud mjeri težinu koja je potrebna kako bi pretvorila korišteni algoritam u identičnu implementaciju u drugačijem programskome jeziku. Formula glasi na sljedeći način:

$$E = D * V$$

Gdje 'E' označava prije spomenuti trud dobiven po umnošku težine i volumena:

$$T = \frac{E}{18}$$

Broj “bugova“ tj. greška u programu računa se na sljedeći način:

$$B = \frac{v}{3000}$$

4.5 Kompleksnost varijabla

Varijable veliki su dio programa bez kojih izvođenje te pisanje programa nije moguće. Definicija varijable može se interpretirati na više načina, no varijabla je memorijska lokacija unutar računala čiji se sadržaj može mijenjati tijekom izvođenja programa. Memorijska lokacija te veličina prostora kojeg ista zauzima ovisi o vrsti varijable. Kako bi programer konstruirao ili razumio određenu smisao dijela koda, potrebno je da prati broj i poziciju varijabla unutar programskog koda, ovisno o poziciji varijabla, mogu se svrstati globalno te lokalno. Lokalne varijable djeluju u određenome segmentu koda te mogu se pozvati i ispisati samo u njemu, a globalne su prisutne u cijelome programu te mogu se koristiti tijekom cijele izvedbe. Žive varijable ili “live variables“ su varijable koje sadržavaju određenu vrijednost te ta vrijednost se može iskoristiti negdje u budućnosti izvođenja programskog proizvoda. Što programski kod sadržava više živih varijabla smatra se kako je teži za razumjeti, iako ne treba biti uvijek tako. Može se koristiti kao jedan od načina mjerenja kompleksnosti programa i može se lako automatizirati pri izvođenju testova. Prosječan broj varijabla dobiva se kao suma svih živih varijabla koja se dijeli sa brojem izvršnih izjava (“number of executable statements“). Žive varijable definirane su sa stajališta korištenja podataka gdje logika dijela programa nije eksplicitno uključena, logika dijela programa koristi se pri definiranju prvih i zadnjih izjava ili referenca za varijablu, ovaj način mjerenja kompleksnosti uvelike se razlikuje od prijašnje napomenutog ciklomatskog oblika mjerenja u kojem se logika programa stavlja na prvo mjesto dok su podaci sekundarna stvar. Primjer koncepta koji primarno stavlja korištenje podataka na prvo mjesto je “span“ ili raspon, definira se kao broj izjava između dviju uzastopno korištenih varijabla. Ako je varijabla referencirana na 'n' različitih mjesta unutar dijela programa, onda za ovu varijablu pretpostavlja se kako ima (N-1) raspon. Prosjek raspona definira se kao prosječni broj izvršenih izjava između dvije uzastopno referencirane varijable. Veći raspon modula programa implicira više referentnih varijabla

unutar programa što usporedno znači i teže razumijevanje programa, drugim riječima raspon se smatra mjerom kompleksnosti.

4.6. Broj čvorova

Broj čvorova kvantitivna je metoda kompleksnosti koja se bazira na lokacijama prijenosa kontrole programa. Metoda je uvelike dizajnirana za korištenje u starom programskome jeziku 'Fortran'. Fortran je imperativni programski jezik koji se koristi za numerička i znanstvena računanja. Izumljen je 1950-ih godina od strane IBM-a specifično za znanstvene te inženjerske primjene kao što su: numeričko predviđanje vremena, geofizika, računska dinamika fluida, računska kemija te fizika. Kako bi se razumjela kompleksnost programa, programer obično crta strelice od točke prijenosa kontrole pa sve do njene destinacije. Poanta metode je stvaranje mentalne slike programa. Prema ovoj metrici, što su strelice više isprepletene, program se smatra više složenijim, te dolazi do usporedbe sa čvorom koji što je više isprepleten to ga je teže za odvezati, u slučaju programa odnosi se na preneseno značenje koje označava zapravo veću kompleksnost programa.

4.7. Topološka kompleksnost

Topološka kompleksnost mjera je kompleksnosti koja je osjetljiva na ugniježđene strukture. Slično kao i ciklomatska kompleksnost temelji se na grafu tijeka modula ili programa, a složenost smatra se maksimalni intersektni broj. Maksimalni intersektni broj računa se tako da se graf tijeka pretvori u spojeni graf (crtanjem strelica od terminalnog ruba pa sve do ruba inicijalizacije), zatim spojeni graf dijeli se na konačan broj regija, broj regija dobiva se tako da se oduzmu broj rubova od broja čvorova te se sumira sa brojem 2.

4.8. broj linija koda (LOC)

Brojanje linija koda u programu vrsta je metrike pomoću koje se opisuje kompleksnost programskog proizvoda. Navedeni način kompleksnosti služi za brojanje linija koje služe za izvedbu programa tj. brojanje instrukcijskih linija. Pojmovi linije koda te instrukcijske izjave zamjenjivi su pojmovi u ovom slučaju istog značenja, a njihov izvor proizlazi iz asemblerskih programa gdje su se oba pojma koristila za isto značenje. Metoda reprezentira veličinu programa te kompleksnost, pretpostavlja se kako veći broj linija označava veću mogućnost za defektnost. Istraživanja su pokazala kako gustoća defekata sadržava veliku korelaciju sa brojem linija koda, prva istraživanja koja su izvedena na ovu temu pokazala su negativnu korelaciju tj. sve što je program sadržavao više linija koda to je gustoća defekata bila manja te ultimativno dovela do skeptičnosti jasnoće ove metrike. 1984. provedeno je istraživanje gdje je proučavan programski kod napisan u FORTRAN-u gdje je broj linija iznosio manje od 200, iz navedenog istraživanja zaključeno je kako je gustoća defekata veća u programima sa manjim brojem linija koda (Basili, V. R., Perricone, B. T.). Proučavani su softveri koji su napisan u PL/S asemblerskom jeziku i Pascalu te pronađena je inverzna povezanost do 500 linija koda (Shen, V. Y.). Veći programski moduli generalno su više kompleksni, niži broj defektnosti iznenađujuće je kontra intuitivan. Izvedeni zaključak iz istraživanja su greške u grafičkom sučelju koje su pretežito konstantne neovisno o samoj veličini programskog proizvoda, no manji softveri (manji broj linija koda) sadržavaju veću gustoću grešaka zbog manjih denominatora. Kasnija istraživanja pokazivala su na krivo linijsku vezu između broja linija koda i stope defekta (Shen, V. Y.). Gustoća defekta smanjuje se sa veličinom modula, ponovno se zakrivljuje prema gore kada moduli narastu u veličini. Proučen je programski kod pisan u jeziku Ada specifično za velike projekte te potvrđena je konkavnost veze između gustoće defekta i veličine modula (Withrow, C., 1990.). Od proučenih 362 modula, interval napisanih linija koda kreće se između 63 i 1000. Zaključeno je kako najmanja gustoća defekta se nalazi u kategoriji od oko 250 linija koda. Kada veličina i apstraktnost programa postane pre opširna, kompleksnost se povećava preko ljudske mogućnosti praćenja i razumijevanja. Krivo linijski model daje informacije kako veličina i gustoća grešaka utječu na kvalitetu, ujedno i implicira kako optimalna veličina programa može

utjecati te imati najmanju stopu defekta. Naravno, optimalna veličina ovisi o okolišu, projektu, jeziku, proizvodu te vrsti, ne postoji jedinstven točan odgovor, no nekakav empirički optimum se može derivirati pomoću metoda prijašnjih istraživanja sličnih ili istih proizvoda te se može postaviti određeni interval koji može služiti kao smjernica za daljnji razvoj softvera.

4.9 Strukturne metrike

Prijašnje nabrojane metrike mjere kompleksnost programa te pretpostavljaju kako je softver entitet za sebe, bez ikakvih međusobnih interakcija. Strukturne metrike uzimaju u obzir različite interakcije između modula u proizvodu te pokušavaju ih kvantificirati. Neki od primjera su: mjerenje particiranja sustava (Belady, L.A., Evangelisti, C.J. 1980.), metrika informacijskog tijeka, mjerenje stabilnosti (Yau, S.S., Collofello, J.S. 1985) te invokacijska kompleksnost (McClure). Jedna od najčešće dizajniranih struktura metrika je "fan-in and fan-out", koja je bazirana na idejama G. L. Myers, L. L. Constantine i E. N. Yourdon-a. Metrika se sastoji od dvije varijable: "fan-in" i "fan-out".
"Fan-in" broji module u programu koji pozivaju druge module, "fan-out" broji module koji su pozvani od određenog modula tj. suprotna interakcija prijašnje navedene varijable. Moduli sa velikim "fan-in-om" relativno su jednostavni i mali, te locirani u nižim slojevima dizajna struktura. Moduli koji su pretežito veliki i kompleksni imaju mali "fan-in" što može indicirati na grešku ili loš dizajn, takvi moduli predisponirani su za redizajn. Sa gledišta kompleksnosti i defekta, moduli sa velikim "fan-in-om" očekivani su da će imati negativnu korelaciju sa stopom defekta, a modeli sa velikim "fan-out-om" očekivani su da će imati pozitivnu korelaciju. Krajnji rezultat istraživanja pokazao se kao neuvjerljiv zbog velikih standardnih devijacija od tih dviju varijabla koje su pronađene unutar istraživanog seta podataka. Jedan od načina na koji je definirana struktura kompleksnosti je:

$$C_P = (fan_{in} * fan_{out})^2$$

U pokušaju uključivanja modula kompleksnosti i strukturne kompleksnosti, definiran je hibridni način njihove metrike informacijskog tijeka:

$$HC_p = C_{ip} * (fan_{in} * fan_{out})^2$$

Gdje C_{ip} označava unutarnju kompleksnost procedure p, koja može biti mjerena sa bilo kojom prije navedenom kompleksnosti. Kompleksnost modela sustava mjeri se kao:

$$C_t = S_t + D_t$$

Gdje C_t označava sustavnu kompleksnost, S_t označava strukturalnu kompleksnost i D_t označava kompleksnost podataka. Relativna kompleksnost sustava može se izračunati pomoću navedene formule:

$$C = C_t \; l \; n$$

Gdje n je broj modula koji se nalaze unutar sustava, strukturnu kompleksnost dalje je moguće definirati kao:

$$S = \frac{\sum f^2 \, (\mathrm{i})}{n}$$

Gdje je S strukturna kompleksnost, f(i) je "fan-out" modul od i, n je broj modula unutar nekog sustava te kompleksnost podataka dalje se izračunava pomoću formule:

$$D_i = \frac{V(i)}{f(i) + 1}$$

Gdje D_i sadržava kompleksnost podataka modula i, V(i) sadržava ulazne/izlazne varijable modula i f(i) je "fan-out" varijabla modula i.

$$D = \frac{\sum D(i)}{n}$$

D označava kompleksnost podataka, D(i) je kompleksnost podataka modula, n je broj modula unutar određenog programa. Prijašnje navedena kompleksnost programa suma je podatkovne kompleksnosti i strukturne kompleksnosti. “Fan-in“ nije veliki pokazatelj kompleksnosti kao što je “fan-out“ koji se pokazao kao bitniji čimbenik. Kompleksnost modula ovisi o broju ulazno izlaznih varijabla, što više ulazno izlaznih jedinica program sadrži, smatra se kako sadržava veću funkcionalnost modula što ujedno znači i veću kompleksnost. Više “fan-out“ varijabla unutar modula označava funkcionalnost nižih razina što označava manju kompleksnost. Provedeno je istraživanje koje dovodi zaključke vezane uz strukturne metrike, pri testiranju modula korišteni su novi moduli, razlog tome je što puno sustava koristi module koji su korišteni više puta za različite svrhe, te već su bili dizajnirani, korišteni i stabiliziranih od trećih strana, što znači da se njihova kvaliteta i pouzdanost dovodi u pitanje. Istraživanje je provedeno na 8 softverskih projekta i zaključeno je kako se sustavna mjera kompleksnosti nalazi blizu subjektivne procjene starijeg i iskusnijeg voditelja pridodano sa stopom pogreške. Korelacija između sustavne kompleksnosti te stope grešaka u razvoju bila je 0.83, gdje je kompleksnost sadržavala 69% varijacija u području greška (Card, D. N., Glass, R. L. 1990.). Regresijsku formulu koja je definirana iz zaključka:

$$Error\ rate = -5.2 + 0.4 * Complexity$$

Formula izražava kako pri svakom rastu kompleksnosti, mogućnost greške (“error rate“) raste za 0.4 greške pri svakih 1000 napisanih linija koda u programskome jeziku.

5.Kvaliteta i razvoj softvera

Razvoj softvera proces je koji obuhvaća ideju, specificiranje, dizajniranje, programiranje, dokumentiranje, testiranje i popravljanje grešaka uključenih u održavanje ili kreiranje nove aplikacije ili druge softverske komponente. U užem smislu razvoj softvera obuhvaća pisanje programskog koda te njegovo održavanje, no u širem, u sam proces uključuju se prije navedeni procesi koji su planirani i strukturirani od samog začeća projekta. U razvoju, ovisno o namjeni, često se uključuje konstantno istraživanje tržišta, proizvoda, izrada prototipa, reinženjering i modifikacije. U kontekstu kvalitete softvera, kvalitetu je moguće svrstati pod dva pojma. Prvi pojam je funkcionalna kvaliteta softvera, pod time se smatra koliko je napravljeni softver kvalitetan u odnosu na konkurenciju te da li zadovoljava sve određene zahtjeve koji su postavljeni na početku projekta. Drugi pojam je strukturna kvaliteta softvera, pod time se smatra koliko je dobro izvedena strana ne funkcionalnih zahtjeva kao što su mogućnost dodavanja novih stvari u programski kod po zahtjevu kupca, koliko dobro su pokriveni granični slučajevi pri testiranju i održivost programa. Softverske greške pri dizajniranju, funkcionalnoj logici, greške oko definiranja podataka te greške u programiranju neke su od najčešćih stvari koje narušavaju kvalitetu softvera. Defekti ili greške kategoriziraju se po važnosti i veličini nastale greške. Uhodane i iskusne organizacije koriste razne alate pomoću kojih računaju defektnu probojnost matrice u kojoj traže broj defekata koji prolaze neprimjetno iz faza razvoja pa sve do kontrolnih grafova kako bi ubuduće unaprijedili proces razvoja te smanjili mogućnost nastajanja te pronalaženja iznenadnih i nenadanih grešaka. Standard ISO/IEC 25010:2011 opisuje hijerarhiju od 8 kvalitetnih karakteristika od kojih se svaka sastoji od 8 karakteristika: funkcionalna pogodnost, pouzdanost, operativnost, efikasnost, sigurnost, kompatibilnost, održivost te prenosivost. Funkcionalna pogodnost prva je karakteristika zapisana od ISO/IEC standarda, a odnosi se na funkcionalnu potpunost, ispravnost i primjerenost, što označava da softverski proizvod treba biti gotov u cjelovitosti, treba biti ispravan tj. raditi u svakome određenome vremenu kada je on potreban i imati mogućnost izvršavanja svih zadataka za koje je on i predviđen. Efikasnost performansa druga je karakteristika koja se odnosi na efikasnost softverskog proizvoda, drugim riječima koliko treba softverskom proizvodu da izvrši jedan zadatak za koji je zadužen te pri tome koliko

troši resursa za određenu radnju i kolika mu je zauzetost memorije pri izvršavanju. Kompatibilnost proizvoda je karakteristika kvalitete koja predstavlja da li je proizvod moguće koristiti na više računala, na drukčijem operacijskom sustavu, kompatibilnost sa drukčijim verzijama sustava, kompatibilnost sa različitim vrstama memorije i procesora. Upotrebljivost softverskog proizvoda odnosi se na krajnjeg korisnika koji će provoditi najviše vremena sa njegovim korisničkim sučeljem, zbog toga je potrebno da sučelje bude jednostavno, intuitivno i estetično. Softverski proizvod treba biti i pouzdan, što znači da treba biti dostupan, tolerantan na moguće greške, zreo te u slučaju rušenja sustava da ima mogućnost oporavka. Sigurnost je jedna od najvećih karakteristika koja osigurava visoki stupanj kvalitete, proizvod ako koristi senzitivne podatke kao što su korisnička imena, lozinke, bankovne podatke te ostale informacije, mora ih pohraniti na način na koji ih neće biti moguće ukrasti i zloupotrijebiti. Kako bi razina sigurnosti ostala visoka konstantno je potrebno pratiti pristupe senzitivnim podacima te njihovim upravljanjem, postavljanje dodatnih sigurnosti kao što je dupla autentifikacija, korištenje vatrenog zida, različitih anti virusa te ostalih mjera sigurnosti. Softverski proizvod mora sadržavati svoj integritet, autentičnost i povjerljivost kako bi bio vjerodostojan. Održavanje sustava je karakteristika koja opisuje kvalitetu proizvoda, softverski proizvod mora biti modularan, sklon modifikacijama, korišten u više scenarija, dobro analiziran te testiran od samih jednostavnih slučajeva pa sve do graničnih testova koji mogu narušiti karakteristiku održivosti. Zadnja karakteristika opisuje portabilnost proizvoda, što znači da proizvod mora imati mogućnost adaptacije na nova mjesta te na nove slučajeve, mora biti zamjenjiv u slučaju kvara i mora imati mogućnost ugradnje na više mjesta, bilo to različite lokacije unutar centra velikih podataka, drugih organizacija, skladišta ili pak ugrađen kao ugradbeni sustav u strojeve, automobile ili ostale ugradbene računalne sustave. ISO/IEC 5055:2021 je ISO standard pomoću kojega se mjeri unutarnja struktura softverskog proizvoda na četiri poslovno orijentirana faktora: sigurnost, efikasnost, otpornost i održavanost, navedeni pojmovi objašnjeni su pri analiziranju ISO/IEC 25010:2011 standarda. Prije dolaska ISO 5055 standarda nije bilo nikakvih internacionalnih standarda koji su mjerili kvalitetu i integritet sustava tako da su evaluirali njegove unutarnje konstrukcije. Zbog toga nastao je navedeni standard koji mjeri te prati proizvod tijekom njegovog razvoja kako bi lakše identificirao te eliminirao slabosti sustava prije nego one

prouzroče veće probleme. Beneficirati od ovog standarda mogu npr. računalni informacijski te uredski sustavi koji mogu koristiti navedeni standard kako bi: evaluirali rizik pojedinih aplikacija koje koriste te alocirali resurse tamo gdje je najviše potrebno, procijenili buduće troškove korištenja aplikacija tako da izračunaju tehnički dug koji se može nakupiti i procijeniti da li poduzimaju prave mjere tijekom proizvodnje pri nastajanju kvalitetnog te manje rizičnog koda. Dobavljači i timovi za nabavu isto tako mogu koristiti ISO 5055 standard kako bi: procijenili te napomenuli kontraktorima koje slabosti se ne smiju nalaziti u kodu, evaluirati kvalitetu dobivenog koda te zatražiti moguću korekciju i postaviti određene kvalitete koje su očekivane od kontraktora. Programski inženjeri mogu beneficirati od navedenog standarda tako da: nauče o kritičnim točkama koje trebaju riješiti prije nego što razviju kod i evaluiraju ga. ISO 5055 evaluira 4 od 8 karakteristika koje su specificirane u prije navedenom te objašnjenom ISO/IEC 25010 standardu. Svaka od navedenih mjera u ISO 5055 standardu u skladu je s prije definiranim kvalitetama unutar ISO/IEC 25010 standarda. Timovi koji su zaduženi za razvoj te proizvodnju proizvoda koriste standarde kvalitete kako bi prije nego što softver izađe na tržište evaluirali strukturnu kvalitetu. Korištenje standarda u ranim fazama razvoja programskog proizvoda omogućava veći postotak uspješnosti projekta, što znači da se kasnije to može implicirati i na ostale programske proizvode sa većim samopouzdanjem te mogućnosti da će proizvod sadržavati manji tehnički dug i kompleksnost. Navedene mjere primarno su dizajnirane da budu automatizirane, tj. da ljudski “input“ bude minimalan u takvim situacijama. Mjere pomoću statičke analize pruže temelje za cijelu industriju uključujući i praćenje poboljšanja te smjer u kojem ide proizvod, postavljanje određenih ciljeva kvalitete te testiranje proizvoda u fazama razvoja.

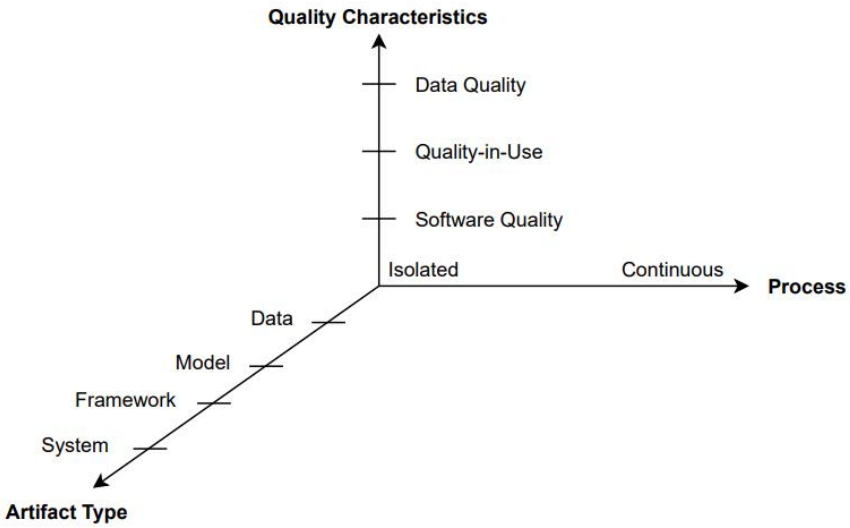
6.Utjecaj kompleksnosti na kvalitetu softvera

Mjerenje kompleksnosti može uvelike smanjiti troškove projekta, poželjno je ranije razmišljanje o samoj kompleksnosti, takav prilazak problemu zapravo daje projektu veću šansu za realizaciju. Mjerenjem kompleksnosti programer zapravo stječe svijest o veličini problema koji je prisutan tj. koliko utječe kompleksnost programa na daljnje faze projekta te koliko neočekivanih troškova je moguće akumulirati tijekom rada, zbog toga mjerenje kompleksnosti na dnevnoj bazi može pomoći pri uočavanju određenih trendova naglih porasta kompleksnosti. Jedan od načina koji može naglo prouzročiti porast kompleksnosti je novi zaposlenik u timu, moguće je da zaposlenik ima drukčije navike te prakse koje se tiču sintakse i načina programiranja koje možda nisu najprikladnije pri izgradnji ili održavanju određenog projekta te isto tako može prouzročiti nagli skok kompleksnosti koji bez čestog provjeravanja ne bi bio uočljiv do kasnije faze, što bi prouzročilo veliki trošak. Skupljanje podataka vezanih uz kompleksnost može pomoći ostalim timovima vizualizirati situaciju te po potrebi dodijeliti više vremena programerima kako bi pojednostavili ili refaktorirali kod. Mjerenjem kompleksnosti u ranim fazama pomaže timovima da otkriju određene probleme u fazi razvoja projekta te omogućuje prepoznavanje područja gdje kompleksnost je konstantno visoka ili u porastu, pronalazak problema kompleksnosti u ranim fazama krucijalan je korak zbog toga jer što je više programskog koda napisano, testirano te poslano u produkciju to otežava refaktoriranje te ga ujedno i uvelike poskupljuje u kasnijim fazama. Kompleksnost uvelike utječe na kvalitetu napisanoga koda, poznato je kako u većini slučajeva kompleksan kod smatra se ne kvalitetnim kodom, mjerenje te konstantno provjeravanje programskog koda prvi je signal opasnosti koji tumači kako kvaliteta koda može predstavljati problem u bližoj i daljnjoj budućnosti. Kako bi programer podigao kvalitetu koda na novu razinu konstantno provjeravanje kompleksnosti nije dosta, treba mu ponuditi određeno vrijeme kako bi naučio pisati kvalitetan kod, osim toga potreban je trening i pomoć koju može zatražiti od starijih ili iskusnijih programera. Rano praćenje kompleksnosti gotovo sigurno će donijeti smanjenje budućih troškova, dokazana je inverzna veza između kompleksnosti i jednostavnosti održavanja navedenog koda. Jednostavno rečeno smanjenje kompleksnosti uvelike će smanjiti rad potreban za rješavanje mogućih problema ili modificiranja dijela programa.

Druga pogodnost koju donosi takav pristup je povećavanje produktivnost programera. Produktivnost programera u većini slučajeva mjeri se tako da se provjerava njegova brzina programiranja; dodavanje novih pogodnosti programa, popravljanje grešaka, provođenja testova te ostalih stvari. U današnjem vremenu većina programera radi zapravo na unaprjeđivanju već postojećeg koda, te što je kod manje kompleksan, to će ga programer lakše i brže riješiti, što će mu pomoći pri rješavanju budućih zadataka te podignuti mu razinu samopouzdanja. Jednostavniji kod ujedno je i lakši za čitati, razumjeti i testirati, što mu omogućuje lakše implementiranje novih promjena i smanjuje mu mogućnost nailaženja na nove probleme pri mijenjanju. Prema mnogim istraživanjima znanstvenici tvrde kako neriješena kompleksnost postaje veliki tehnički dug u budućnosti, programeri su nekada forsirani predati kompleksan programski kod kako bi zadovoljili određene datume te da bi zadovoljili normu koja im je postavljena, jednostavnije je predati špageti (zapetljan) kod za razliku od čistog i elegantnog koda. Činjenica je da nisu svi programeri svjesni važnosti pisanja kvalitetnog koda, zbog toga jer u pravilu se ne prakticira organizacijski proces koji nadgleda pisanje kompleksnosti koda te potiče čisto kodiranje. Kada se piše kompleksan kod, on predstavlja prije spomenuti tehnički dug koji će se negdje u budućnosti trebati počistiti tj. promijeniti kako bi se osigurala čitljivost i korisnost programskog koda te lakše održavanje. Naravno, nemoguće je izbjeći 100% kompleksnost svakog programskog koda, svaki projekt sadržava određeni stupanj kompleksnosti, te u praksi nemoguće je iskorijeniti svaki pre kompleksni slučaj. Cilj je identificirati koji kompleksni kod ima najveći utjecaj na krajnjeg korisnika te fokusirati se na rješavanje takvog problema. Pomoću raznih softverskih programa moguće je identificirati područja proizvoda koji reprezentiraju rizik. Način na koji oni mogu raditi je da proučavaju dijelove koda koji su promijenjeni, nedovoljno testirani i često pristupani od strane krajnjeg korisnika. Kada se identificira takav kod, može se fokusirati na refaktoriranje te pojednostavljivanje kompleksnog koda gdje je najbitnije.

7.Osiguranje kvalitete u sustavima umjetne inteligencije

„Broj te važnost UI – baziranih sustava u svim domenama raste. Uz sveopću upotrebu i ovisnost o sustavima temeljenim na umjetnoj inteligenciji, kvaliteta tih sustava postaje ključna i esencijalna za njihovu praktičnu ulogu u društvu (Felder, M., Ramler, R., 2021).“ Osiguravanje kvalitete navedenih sustava nova je grana istraživanja koja nije dovoljno istražena te zahtjeva kolaboraciju i diskusiju između grana kao što su računarska znanost, računalno inženjerstvo te programsko inženjerstvo. Veliki napredak u dubokom učenju i umjetnoj inteligenciji postavlja nova pitanja i izazove za inženjere te znanstvenike. sustavi koji se baziraju na umjetnoj inteligenciji skloni su konstantnoj promjeni tj. evoluiranju, njihovo ponašanje se konstantno mijenja, ovise o velikim količinama podataka na kojim treniraju svoj model i imaju mogućnost adaptiranja na promjene bez fizičke promjene sa strane programera. Ovakve karakteristike modela nisu viđene mnogo u prošlosti te potreban je novi pristup problemu kako bi se pravilno osigurala kvaliteta. Koncepti kvalitete u sustavima nisu dobro definirani zbog same prirode sustava (Borg, M., 2021). Terminologija se uvelike razlikuje u umjetnoj inteligenciji i programskome inženjerstvu što znači da neki od principa koji se nalaze u PI sustavima ne vrijede za AI sustave (Lenarduzzi, L.V., 2021). Potrebno je dobro definiranje te razmatranje pojmova između dvije navedene grane, npr. regresija unutar umjetne inteligencije označava regresijski model, a kod programskog inženjerstva označava regresijsko testiranje. Testiranje je još jedan od pojmova koji se uvelike razlikuje između grana, u programskome inženjerstvu označava pokretanje programa kako bi se prikazale te otkrile greške, dok kod umjetno inteligentnih sustava evaluira preciznost treniranog modela. komponente umjetne inteligencije fundamentalno se razlikuju od tradicionalnih komponenti pretežito zbog njihovog ponašanja koje je ne deterministično te orijentirano statistici i mijenja se tijekom određenog prolaska vremena koje ovisi o novim podacima na kojima model uči. Podaci na kojima takav model uči su pre procesirani i pripremljeni za daljnju obradu.



Slika 7.1. - Dimenzije UI sustava i karakteristike kvalitete (Felderer, M., Ramler, R. 2021.)

Sa strane artefakta tj. njegove dimenzije mogu se iščitati komponente UI a to su: sustav, "framework" tj. okruženje, model i podaci. Dimenzija procesa može procijeniti način na koji je sustav razvijen tj. da li je konstantno iterativno uzimao odgovore od prije objašnjenih komponenata ili je u procesu bio izoliran, iz navedenih dimenzija iščitava se kvaliteta karakteristika sustava (Slika 5.1.). Sljedeće karakteristike smatraju se kao svojstva kvalitete unutar AI modela i komponenta: točnost, robusnost, relevantnost modela, sigurnost, privatnost podataka, efikasnost, interpretaciju i poštenje (Zhang, J.M.). „Točnost ili korektnost referira se na vjerojatnost da UI komponenta napravi ispravnu stvar (Zhang, J.M.)". Relevantnost modela koristi se kao mjerilo koliko dobro komponenta odgovara podacima, robusnost sustava referira koliko je komponenta otporna na perturbaciju tj. da li narušava postavljeni red, sigurnost referira otpornost sustava prema potencijalnoj opasnosti kroz ilegalno pristupanje UI komponenata preko treće osobe, privatnost podataka jednostavno referira mogućnost komponenata da sačuva one osjetljive dijelove podataka koji nisu namijenjeni za osobe koje nemaju dozvoljenu autorizaciju pri pristupanju, efikasnost mjeri brzinu predviđanja navedenih komponenata, poštenje osigurava da komponente donose prave odluke radi pravog razloga kako bi izbjegle probleme ljudskih prava, diskriminacije te ostale moguće etične probleme sa kojima se može susreći, i na kraju interpretacija koja označava stupanj razumijevanja donesenih odluka od strane UI (J.M. Zhang). Kvaliteta podataka jedan je od vrlo bitnih

čimbenika o kojima ovisi kvaliteta umjetno inteligentnih komponenta. Prema ISO/IEC 25012 kvaliteta podataka može se klasificirati primarno na 2 načina: inherentna kvaliteta podataka koja se odnosi na moguće restrikcije, vezu, vrijednosti te meta podatke, druga vrsta podataka su podaci kvalitete koji su ovisni o sustavu tj. stupanj podataka u kojem je kvaliteta dostojna te očuvana (ISO 2008). “Framework“ tj. softversko okruženje bazira se na prijašnje objašnjenom ISO/IEC 25010 standardu. U današnje vrijeme postoji manjak standardiziranih pristupa kako bi osigurali kvalitetu navedenog sustava, razumijevanje problema je i dalje vrlo daleko. Takvom problemu treba pristupiti sa drukčijeg i inovativnog gledišta, veliku ulogu igraju ulazni podaci gdje mali jedva vidljiv šum u podacima može dramatično utjecati na krajnje rezultate klasifikacije. „Neki od faktora koje se treba uzeti u obzir pri osiguravanju kvalitete su: razumijevanje te interpretiranje umjetno inteligentnih modela, manjak definiranih potreba i specifikacija, potreba za validacijom podataka, točnost mjerenja, definiranje očekivanih ishoda te korištenje takvih ishoda pri testiranju modela, ne funkcionalna svojstva sustava, dinamička te često mijenjana okruženja i samo adaptivne tj. samoučeće karakteristike“ (M. Felderer, R. Ramler). Razumijevanje i interpretiranje prvi je ključni faktor u kojem se podatkovni znanstvenici muče zbog same apstrakcije umjetne inteligencije te dubokog učenja. Glavni problem su ne intuitivnost te težina razumijevanja, još se nazivaju i „crne kutije“ (Goebel, R. 2018). Zbog same apstraktnosti, manjak specifikacija te definiranih zahtjeva, programeru je teško razumjeti kako točno takav sustav funkcionira te prilikom testiranja vrlo je zahtjevno postavljanje pravih graničnih testova radi lakših pronalaženja defekata. Manjak specifikacija i definiranih zahtjeva sekundarni je faktor, podatkovni pristup ne oslanja se na predefinirane zahtjeve ili specifikacije, modeli se automatski generiraju iz već postojećih podataka. Podaci koji se koriste pri učenju modela sadrže određeni ulaz i izlaz. Učenje se odvija pomoću korištenja različitih algoritama koji se biraju ovisno o vrsti problema, no svi pomoću predefiniran ih pravila spajaju određene ulazne podatke sa označenim izlaznim podacima. Pravila se ne definiraju u samome početku. Konvencionalni razvoj softvera zapravo radi u inverznom ili obrnutom smislu za razliku od učenja koji se bazira na velikim podacima te njihovim pristupima (Fischer, L., Ehrlinger, L., Geist, V., Ramler, R., Sobieczky, F., Zellinger, W., Moser, B., 2020.). Pravila u konvencionalnom razvoju softvera definiraju se prije nego što je sustav implementiran. U pitanje se dovodi kako

zapravo testirati umjetno inteligentni sustav, pošto tehnike testiranja u konvencionalnim sustavima rade na način da za dane ulazne podatke, se pretpostavi mogući rezultat za taj dani “input“ ili ulaz, to dovodi do problema kako zapravo generirati testne podatke u umjetno inteligentnim sustavima. Razlikuje se primarno više vrsta tehnika testiranja koje su bazirane na specifikacijama (crna kutija), strukturama (bijela kutija) ili bazirane na iskustvu, slični testovi bazirani na umjetno inteligentnim sustavima tek se trebaju izumiti. Prve tehnike koje su predložene rade na način iskorištavaju strukturne informacije dubokih neuronskih mreža kako bi izmjerile raspršenost neurona (Xie, X., Ma, L., Juefei-Xu, F., Xue, M., Chen, H., Liu, Y., Zhao, J., Li, B., Yin, J., See, S, 2011). Testirani podaci generiraju se na način da probaju maksimizirati raspršenost. Različiti pristupi pri pronalaženju dobrih testova se provode od nasumično generiranih podataka pa sve do metamorfnih pristupa (Zhang, M., Zhang, Y., Zhang, L., Liu, C., Khurshid, S, 2019). Cilj testiranja je pronalaženje određenih grešaka programa, kako bi se greška pronašla programer mora za neki određeni “input“ postaviti očekivani “output“ tj. za nekakav dani ulaz treba biti pretpostavljen koji izlaz je točan, a koji je netočan. Problem je što se programer ili tester nalazi u kontroliranom okruženju sa kontroliranim podacima te se u tome slučaju može činiti kako softver radi 100% te prolazi sve testove, čak i one granične, problem nastaje kada taj programski proizvod odlazi u „divljinu“ tj. kada dođe do krajnjih korisnika gdje on više nije u kontroliranom okruženju te je dinamički kreiran. U tom slučaju program korisnici „testiraju“ sa kombinacijama koje nisu probane ili iščekivane od strane programera, ovo se naziva “oracle problem“. Preciznost te točnost pojmovi su koji se uvelike razlikuju u umjetno inteligentnim modelima za razliku od ostalih modela. Softverski proizvod generalno treba bili točan te deterministčan, bilo kakva mala ili velika devijacija od toga pretvara takav proizvod u nefunkcionalan te ga podlaže popravku. Naravno, niti jedan softver u vanjskome svijetu nije 100% točan, ne postoji savršen sustav, ali principi točnosti uvelike se razlikuju u softverskom inženjerstvu te u umjetno inteligentnim sustavima. UI sustavi sadržavaju određenu točnost koja se nalazi u intervalu između 0% i 100%. Konvencionalni sustav sa točnosti od 99% označava da neće raditi kako spada u 1 od 100 pokrenutih puta, dok kod umjetno inteligentnih sustava 99% točnosti označava vrlo visoki stupanj kvalitete, naravno ovisno o kakvome sustavu se radi te koja je njegova upotreba u stvarnome životu. Ako je sustav napravljen za prepoznavanje životinja na

slikama 99% točnosti je vrlo dobar rezultat, dok ako se sustav sa istom točnosti koristi pri sprječavanju krađa tijekom bankovnih transakcija, model neće biti dovoljno dobar. Testiranje ne funkcionalnih dijelova umjetno inteligentnog sustava rijetko je istraženo, pogotovo efikasnost te robusnost za testiranje tijekom produkcije proizvoda. Robusnost je vrlo teško testirati u takvim sustavima zbog vrsta podataka sa kojima se radi npr. audio podaci ili slike. Testiranje efikasnosti kod umjetno inteligentnih sustava isto je jedan od zahtjevnih područja zbog toga jer osim što se takve komponente bave sa brzinom predviđanja, isto tako se bave sa brzinom konstruiranja koja je teško mjerljiva pogotovo u kontekstu pravog života npr. u autonomnim sustavima auta gdje je potrebno donesti prave odluke u trenutku. Regresijska testiranja veliki su zadatak u bilo kakvom modernom razvoju softvera, neovisno o njegovoj prirodi, način na koji kvaliteta može rasti je razvijanje te implementiranje modernih automatskih testova, takvih testovi kod tradicionalnih softvera definitivno osiguravaju veći stupanj kvalitete nego kod umjetno inteligentnih sustava. Razlog tome je priroda UI sustava koji se dinamički tijekom rada konstantno mijenjaju te adaptiraju na nove promjene u njihovoj okolini te konstantno uče. Testiranje dinamičkih sustava vrlo je zahtjevno za implementirati te donosi mnoga pitanja kao kako se nositi sa kompleksnošću i dinamičnosti takvih sustava. Umjetno inteligentne komponente obavljaju svoj rad u okolini koja se konstantno mijenja. Nekakvi primjeri takvih aplikacija bili bi podaci na internetu te senzori na fizičkim objektima koji konstantno prikupljaju nove podatke sa različitih izvora. U tome slučaju dovodi se u pitanje sama kvaliteta podataka koji se skupljaju. Podaci se isto tako mogu prikupljati iz simuliranih okruženja. Zbog velikih kompleksnosti takvih sustava jedan od načina je “online“ testiranje gdje se procjenjuje te prati ponašanje sustava u realnim uvjetima dok je on pokrenut. Aplikacije u ne simuliranim uvjetima pružaju developerima kako se aplikacija ponaša kada je u doticaju sa krajnjim korisnicima te otkriva puno novih stvari koje su oni propustili tijekom proizvodnje softverskog proizvoda. Pronalaženje i implementiranje pravih testova ključ je uspjeha takvih sustava, ako se koriste krivi testovi koji nisu prilagođeni takvom sustavu može se činiti kako sustav radi odlično u svom okruženju, a zapravo u realnosti problem će biti puno veći što će rezultirati velikim gubitkom. Potrebno je definirati točne i validne parametre u kojima se sustav može testirati pritom paziti na etične odgovornosti (Felderer, M., Ramler, R.).

8.Zaključak

Činjenica je kako je kompleksnost vrlo bitan pojam i faktor pri osiguravanju kvalitete softvera koji se ne smije izostaviti pri konstruiranju novog softverskog proizvoda. Kompleksnost se uvijek treba držati na minimalnoj mogućoj razini, iako se može činit lakše i brže za napisati kompleksniji i zapetljaniji kod, u budućnosti kada spektar programskog koda postaje veći, puno će biti teži za održavati, popraviti i sama cijena će postati puno veća, osim toga teže će biti implementirati nove značajke programa što ultimativno dovodi do nezadovoljstva krajnjih korisnika. Programeri ili developeri te općenito kompanije trebaju posvetiti više vremena proučavanju kompleksnosti te njezinoj međuovisnosti sa kvalitetom kako bi omogućili što veću razinu točnosti cjelokupnog projekta, smanjili budući tehnički dug te ultimativno olakšali sebi posao te povećali šansu uspješnosti cjelokupnog projekta. U novije doba gdje se sve više spominje pojam umjetna inteligencija i njeni sustavi, potrebni su novi načini mjerenja te definiranja kompleksnosti te koja je međuzavisnost kompleksnosti i kvalitete kod takvih sustava.

9.Slike

10.Literatura

Zuse, H.(2019.) *Software complexity: measures and methods* (Vol. 4). Walter de Gruyter GmbH & Co KG.

Alexander, C. (2018.) What is software complexity and how can you manage it? *The Man in the Arena*. [Online]. Dostupno na: https://carlalexander.ca/what-is-software-complexity/#:~:text=Software%20complexity%20is%20a%20way,quality%20grade%20for%20your%20code
[Pristupljeno: 1.8.2022.]

Lehman, M.M. and Belady, L.A. eds., (1985.) *Program evolution: processes of software change*. Academic Press Professional, Inc..

GeeksforGeeks (2021.) Complexity Metrics. *GeeksforGeeks.* [Online]. Dostupno na: https://www.geeksforgeeks.org/complexity-metrics/#:~:text=A%20complexity%20measure%20is%20a,the%20complexity%20with%20maintenance%20effort
[Pristupljeno: 3.8.2022.]

Watson, A., McCabe, T. (1996). Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. Mjesto izdavanja: Computer System Laboratory, National Institute of Standards and Technology Gaithersburg, MD 20899-0001
Ikerionwu, C. (2010.) Cyclomatic complexity as a Software metric. International Journal of Academic Research. 2.

GeeksforGeeks (2020.) Sofware Engineering | Halstead's Software Metrics. *GeeksforGeeks.* [Online]. Dostupno na: https://www.geeksforgeeks.org/software-engineering-halsteads-software-metrics/
[Pristupljeno: 4.8.2022.]

Sealights (2022.) What's wrong with Software Complexity? *Sealights.* [Online]. Dostupno na: https://www.sealights.io/software-quality/whats-wrong-with-software-complexity/ [Pristupljeno: 4.8.2022.]

Zaytsev, YV., Morrison A. (2013.) Increasing quality and managing complexity in neuroinformatics sofware development with continuous integration . *Frontiers.* [Online]. Dostupno na: https://www.frontiersin.org/articles/10.3389/fninf.2012.00031/full [Pristupljeno: 7.8.2022.]

Nystedt, S., Sandros, C. (1999.) *Sofware Complexity and Project Performance.* Gothenburg: School of Economics and Commercial Law at the University of Gothenburg

GeeksforGeeks (2022.) Data Structures. *GeeksforGeeks.* [Online]. Dostupno na: https://www.geeksforgeeks.org/data-structures/ [Pristupljeno: 10.8.2022.]

Devopedia. (2022.) Algorithmic Complexity. *Devopedia.* [Online]. Dostupno na: https://devopedia.org/algorithmic-complexity [Pristupljeno: 10.8.2022.]

CISQ. (2022.) Sofware Quality Standards – ISO 5055. *CISQ.* [Online]. Dostupno na: https://www.it-cisq.org/standards/code-quality-standards/ [Pristupljeno: 10.8.2022.]

Kan, S. (2002.) *Metrics and Model sin Sofware Quality Engineering.* Second Edition. Adison wesley

Felderer, M., Ramler, R. (2021.) *Quality Assurance for AI-based Systems: Overview and Challenges.* [Online]. Dostupno na: https://arxiv.org/pdf/2102.05351.pdf [Pristupljeno: 11.8.]

Belady, L.A., Evangelisti, C.J., Power, L.R. (1980.) Greenprint: A graphic representation of structured programs. IBM Systems Journal vol. 19, no4.

S. S. Yau and J. S. Collofello, (1985.). "Design Stability Measures for Software Maintenance." *IEEE Transactions on Software Engineering*.

Software Intelligence for Digital Leaders (2022.). What is software complexity? *Software Intelligence for Digital Leaders.* [Online]. Dostupno na: https://www.castsoftware.com/glossary/software-complexity [Pristupljeno]: 23.8.2022.

11.Sažetak

U ovome završnome radu opisani te objašnjeni su pojmovi kompleksnosti te utjecaj kompleksnosti naspram kvalitete proizvoda. Prikazane su početne metrike kompleksnosti, strukturne kompleksnosti, kompleksnosti kontrolnog tijeka, objašnjena je važnost kvalitete podataka u navedenome području i njihova uloga u široj slici. Detaljno su opisane vrste kompleksnosti, njihovo podrijetlo, mogućnost implementacije te potkrijepljeni su dokazima formula deriviranih iz dokaza. Kasnije su definirane strukturne metrike, njihove razlike te implementacija istih. Prikazana je kvaliteta, razvoj te procesi koji počinju od ideje pa sve do implementacije softverskog proizvoda. Nakon toga određena je uloga kompleksnosti te njen odnos sa kvalitetom softverskog proizvoda, načine na koje se ona može vrlo brzo promijeniti, najčešće krivce koji to mogu prouzročiti, obrađena je bitnost refaktoriranja te pisanja dobrog i čistog koda kako bi u budućnosti mogući tehnički akumulirani dug bio manji te kompleksnost softvera razumljivija i lakša za održavanje. Priča je zaokružena sa modernom temom koja se bazirala na osiguranju kvalitete u umjetno inteligentnim sustavima te načinima pomoću kojih se kompleksnost može mjeriti naspram tradicionalnih softverskih sustava.

12.Abstract

In this bachelor thesis, the concepts of complexity and how the influence of complexity affects product quality are described and explained. The initial metrics of complexity, structural complexity, control flow complexity are presented, and the importance of data quality in the mentioned area is explained. The types of complexity are described in detail, their origin and where they can be used. Those claims are supported by the evidence of various formulas that are related, and derived from them. Structural metrics, their differences and their implementation were defined. The quality, development and processes that start from the idea all the way to the implementation of the software product are presented. After that, the role of complexity on software quality, the ways in which it can change very quickly, the most common culprits that can cause it, and the importance of refactoring and writing good and clean code, in order to reduce the possible technical accumulated debt in the future alongside with maintenance of software were discussed. The story is rounded off with a modern theme that was based on quality assurance in artificially intelligent systems and the ways in which complexity can be measured against traditional software systems.