Univerzitet „Džemal Bijedić“ Mostar

Fakultet informacijskih tehnologija

Statička verifikacija softvera

- Formalne metode –

Profesor: van.prof.dr. Nina Bijedić Student: Adna Zlomužica

Sadržaj

[1. Uvod 4](#_Toc32840547)

[2. Šta su formalne metode? 5](#_Toc32840548)

[3. Statička verifikacija softvera 6](#_Toc32840549)

[3.1 Ciklomatična složenost 7](#_Toc32840550)

[3.2 Broj linija koda 10](#_Toc32840551)

[3.3 Halstead volumen 11](#_Toc32840552)

[3.4 Indeks održivosti 12](#_Toc32840553)

[5. NDepend i Code Analysis Tool 13](#_Toc32840554)

[7. Zaključak 19](#_Toc32840555)

[8. Reference 20](#_Toc32840556)

Slike

[Slika 1. Oznaka grafikona kontrolnog toka 8](#_Toc32840574)

[Slika 2. Demo izvorni kod napisan u C# 9](#_Toc32840575)

[Slika 3. Grafikon kontrolnog toka 9](#_Toc32840576)

[Slika 4. NDepend glavni izbornik u VS 15](#_Toc32840577)

[Slika 5. NDepend forma za odabir projekta 15](#_Toc32840578)

[Slika 6. Forma za odabir načina prikaza rezultata 16](#_Toc32840579)

[Slika 7. NDepend Dashboard 16](#_Toc32840580)

[Slika 8. Odabir opcije Code Metrics u NDepend izborniku 17](#_Toc32840581)

[Slika 9. Pregled ciklomatične složenosti u NDepend alatu 17](#_Toc32840582)

[Slika 10. Način pokretanja Code analysis alata 18](#_Toc32840583)

[Slika 11. Pregled kodne metrike unutar Code Analysis alata 18](#_Toc32840584)

[Slika 12. Pregled ciklomatične složenosti unutar Code Analysis alata 18](#_Toc32840585)

[Slika 13. Pregled broja linija unutar NDepend alata 18](#_Toc32840586)

[Slika 14. Pregled broja linija unutar Code Analysis alata 19](#_Toc32840587)

[Slika 15. Pregled indeksa održivosti unutar Code Analysis alata 19](#_Toc32840588)

# 1. Uvod

Od prvog pojavljivanja informacijskih sistema postavlja se pitanje zašto sistemi nisu bez grešaka. Jednostavan odgovor je da su čak i najjednostavniji sistemi pod uticajem parametara koje ne možemo dokučiti i koje ne možemo kontrolirati. Najčešći način provjere grešaka u informacijskim sistemima je upotreba testiranja i simulacije. Ipak ove tehnike ne mogu jamčiti nepostojanje nedostataka u sistemu. Metod koji mnogo više obećava po pitanju uočavanja i otklanjanja grešaka u sistemu jeste korištenje formalnih metoda tijekom procesa razvoja informacijskih sistema, na primjer, pisanjem formalne specifikacije sistema na kojem se mogu dokazati različita svojstva i matematički dokazati da se implementacija sistema pridržava specifikacije. Cilj ovog seminarskog rada jeste prikazati statičku verifikaciju softvera upotrebom formalnih metoda, te prikazati šta su formalne metode i kako mogu smanjiti broj grešaka u sistemu. Statička verifikacija softvera je vrsta softverskog testiranja koje se izvodi kako bi se provjerili nedostaci u softveru bez da se izvrši kod softverske aplikacije. Dok se kod dinamičke verifikacije softvera kod izvršava kako bi se otkrili nedostaci i greške u softveru. Statičko ispitivanje vrši se u ranoj fazi razvoja kako bi se izbjegle pogreške jer je lakše pronaći izvore grešaka i lahko ih je ispraviti. Pogreške koje se ne mogu pronaći pomoću dinamičkog testiranja lahko se mogu pronaći pomoću statičkog testiranja. Postoje uglavnom dvije vrste tehnika koje se koriste u statičkom ispitivanju,a to su: pregled i statička analiza. U ovom seminarskom radu bit će prikazana statička verifikacija korištenjem statičke analize. Statička analiza uključuje procjenu kvalitete koda koju pišu programeri. Za analizu koda i usporedbu istog sa standardom koriste se različiti alati. Također pomaže u identificiranju sljedećih nedostataka: mrtvi kod, beskonačne petlje, varijable sa nedefiniranom vrijednošću i pogrešna sintaksa. Statička analiza u ovom seminarskom radu će biti prikazana preko tri parametra, a to su: ciklomatična složenost (cyclomatic complexity), indeks održivosti (maintaibility index) i broj linija koda. Ciklomatična složenost je mjerenje složenosti izvornog koda koje je povezano sa brojnim pogreškama kodiranja. Izračunava se razvijanjem grafikona kontrolnog toka koda koji mjeri broj linearno nezavisnih staza kroz programski modul.[1] Indeks održivosti koda jedan je od važnijih mjernih podataka koji pokazuju koliko je lahko održavati kod. Kada se počne pisati kod za određenu metodu, indeks održivosti ima vrijednost 100, te sa nekvalitetnim promjenama na kodu i drugim faktorima metrike, indeks se počinje smanjivati. Smanjivanje indeksa ovisi o broju spojenih klasa u jednu cijelinu, kao i o dubini naslijeđivanja među klasama. Linije koda označavaju broj izvršivih redaka koda u metodi. Ovaj je broj približan broj, a temelji se na intermedijarnom jeziku koda (IL). Uključuje samo izvršne retke koda, tako da su isključeni komentari, zagrade i bijeli razmak. Kao što se može pretpostaviti, što je više koda u aplikaciji, to je više koda za održavanje. Stoga, za linije koda, niska vrijednost je dobra, a visoka vrijednost loša. Za potrebe ovog seminarskog rada koristit će se alati NDepend i alat Code Analysis Tool koji je dio razvojnog okruženja Visual Studio. NDepend je alat za analizu statičkog koda u kojem je primarni cilj omogućiti analiziranje aplikacije i sticanje jasne slike o tehničkom dugu, prekršenim pravililma, ciklomatičnoj složenosti unakrsnim ovisnostima objekata, te stepenu povezanosti među njima. S tim, NDepend je alat dizajniran za dobivanje više razine statičke analize koji nam pomaže dizajnirati bolji sistem, a ne pisati ispravniji kod. Code Analysis Tool je alat koji nam daje informacije o indeksu održivosti sistema, ciklomatičnoj složenosti i broju linija koda,a na taj način razvojni programeri mogu u bilo kojem trenutku provjeriti na kojoj je razini kvaliteta koda softvera koji se razvija. S tim je moguće da se prijevremeno reagira na neke propuste u kodu,tako što se u ranim fazama razvoja rješava nepotrebna kompleksnost u određenim dijelovima koda. Tokom rada na ovom seminarskom radu za potrebe testiranja koristit će se jedan .NET demo projekat.

# 2. Šta su formalne metode?

Formalne metode su tehnike dizajniranja sistema koji koriste strogo određene matematičke modele za izgradnju softverskog i hardverskog sistema. Za razliku od drugih dizajnerskih sistema, formalne metode koriste matematički dokaz kao dodatak ispitivanju sistema kako bi se osiguralno ispravno ponašanje.[2] Snaga formalnih metoda jeste da omogućavaju potpunu provjeru cjelokupnog prostora sistema i da će se svojstva koja se mogu dokazati držati u sistemu za sve moguće ulaze. Kada se formalne metode ne mogu koristiti kroz cjelokupni razvojni proces sistema (zbog složenosti sistema, nedostataka alata ili drugih razloga), oni se i dalje mogu uspješno koristiti na određenim dijelovima sistema. Raznolikost dostupnih metoda rezultat je različitih metoda modeliranja i dokazanih pristupa potrebnih za različite domene primjene. Također, različite faze razvoja sistema mogu zahtijevati različite alate i tehnike. Kada se treba primijeniti novi sistem, prvi korak je pisanje specifikacije zahtjeva. Specifikacija bi trebala ispravno opisati željeno ponašanje sistema i trebala bi biti cjelovita i nedvosmislena, što je teško postići. Specifikaciju potom programer transformira u kod. Programer treba da ispravno razumije specifikaciju i da riješi nejasnoće koje se jave. Također, programerov način kodiranja i rješavanja tehničkih izazova može uvesti pogreške u kod. Zatim slijedi sama veličina sistema; danas su sistemi toliko veliki da je teško pratiti da li svi dijelovi sistema ispravno slijede specifikaciju. Nadalje, često postoji tim programera koji radi zajedno, što je također izvor grešaka u sistemu, jer svi imaju svoj način tumačenja specifikacije i informacija dijeljenih tijekom razvojnog procesa. Tijekom i nakon kodiranja sistema, funkcionalnost sistema obično se testira kako bi se osiguralno da rezultirajući program zadovoljava zahtjeve i da nema grešaka. Ispitivanje velikih i složenih sistema može oduzeti puno vremena zbog veličine sistema i količine koda, što praktično nije izvedivo. Ipak, kada je za sistem bitna sigurnost, mora biti zajamčena ispravna funkcionalnost, što zahtjeva ili iscrpno testiranje ili način dokazivanja da kod ispravno implementira specifikaciju. Koncept formalnih metoda uvodi alate za matematičko opisivanje sistema ili dijelova sistema u specifikaciji i za dokazivanje da rezultirajući program ispunjava zahtjeve opisane u specifikaciji. Formalna specifikacija je precizna i nema rizika od pogrečnih tumačenja. Također, ako postoji dokaz da se implementacija pridržava specifikacije, onda se može biti sigurno da su programeri implementirali ono što je opisano u specifikaciji. U praksi se ne može u potpunosti jamčiti da je dobivena implementacija bez grešaka, jer korištena formalna metoda može imati nedostatke ili može postojati neka greška u dokazu. Unatoč tome, povećana upotreba formalnih metoda i alata će rezultirati boljim i pouzadnijim metodama i alatima. Korištenjem formalnih metoda u razvoju sistema, pogreške se mogu pronaći ranije, a neke klase grešaka se mogu u potpunosti otkloniti. Ograničenje formalnih metoda je to što se one mogu koristiti samo za dokazivanje ispravnosti sistema u odnosu na specifikaciju. Stoga samo zato što je program matematički implementiran kako bi se pridržavao specifikacije, nije jamstvo da je specifikacija sama po sebi ispravna i da nema grešaka. Na kraju, stepen primjene formalnih metoda može varirati. Ponekad je potpuna primjena formalnih metofa preskupa, osim u slučaju kritično važnih sistema gdje pogreške mogu rezultirati skupim redizajnom. Često se formalne metode koriste samo za opisivanje željene fukcije i usmjeravanje razvoja. To se smatra stepenom 0. Kada se formalne metode koriste za provjeru fukcija, ona se smatra stepenom 1. Stepen 2, najviši stepen formalnih metoda je kada se cijeli sistem provjerava kroz sve svoje funkcije. Korištenje formalnih metoda dugoročno može biti isplativo i može se smanjiti broj grešaka. Međutim, formalne metode se još uvijek ne koriste dovoljno, i prema tome, postoji ograničen broj alata koji se mogu koristiti.

# 3. Statička verifikacija softvera

Poznata je činjenica da je vrijednost otkrivanja grešaka i nedostataka tijekom razvojne faze aplikacije iznimno visoka. Isto tako mogućnost otkrivanja nedostataka u ranim fazama razvoja softvera je moguće ukoliko se može desiti otkrivanje grešaka bez izvršenja glavnog koda aplikacije. Postizanje ovog cilja testiranja omogućeno je statičkim ispitivanjem, što predstavlja jednu od važnijih tehnika testiranja koja se provodi u fazi provjere softvera. Statičko testiranje, što predstavlja vrstu metodologije testiranja softvera je provjera softverskog proizvoda, izvršena u statičkom okruženju, tj. testiranje izvedeno bez izvršavanja koda. Uz pomoć ručnih i automatiziranih pregleda dokumenata, statičko ispitivanje omogućava rano otkrivanje oštećenja tijekom početne faze razvojnog ciklusa proizvoda. Ova vrsta testiranja provjerava kod, dokumente zahtjeva i projektne dokumente, te daje komentare na radnom dokumentu. Kada je softver neoperativan i neaktivan, provodimo sigurnosno testiranje kako bismo analizirali softver u izvanmrežnom okruženju. Statičkim testiranjem pokušavamo saznati pogreške, nedostatke koda i potencijalno zlonamjerni kod u softverskoj aplikaciji. Ona počinje ranije u životnom ciklusu razvoja, pa se zbog toga naziva verifikacijsko testiranje. Statičko testiranje može se obaviti na radnim dokumentima poput specifikacije zahtjeva, projektnim dokumentima, izvornom kodu, planovima ispitivanja testnih slučajeva kao i na sadržaju web stranica. Tehnika statičkog ispitivanja uključuje sljedeća dva koncepta: pregled prateće projektne dokumentacije i statičku analizu. Pregled prateće projektne dokumentacije je tehnika koja uključuje metodu za hvatanje i uklanjanje grešaka, redudancije ili nejasnoća u popratnim dokumentima, kao što su specifikacije softverskih zahtjeva, tehnički projektni dokumenti, dokumenti o poslovnim zahtjevima, dokumenti funkcionalnih zahtjeva, itd. Statička analiza uključuje ocjenu kvalitete koda koji su napisali programeri. Postoje različiti alati koji mogu analizirati kod i usporediti ga sa standardnim praksama, identificirati složenost sistema, neiskorištene varijable, mrtvi kod, beskonačne petlje, itd. To pomaže brzo prepoznati strukturne nedostatke koji kasnije mogu rezultirati stvarnim oštećenjima softvera. U ovom seminarskom radu kao što je već navedeno bit će prikazana tehnika statičke analize. Možemo pregledati nedostatke u kodu prije nego što započnemo stvarno testiranje uz pomoć alata koji tijekom statičke analize pokrivaju određene vrste nedostataka. Neke od vrsta nedostataka koje ćemo mi prikazati u ovom radu jesu ciklomatična složenost, indeks održivosti i broj linija koda, o čemu će biti rečeno više u narednim dijelovima ovog seminarskog rada. Alati koji će se koristiti za prikaz rezultata statičke analize su NDepend i Code Analysis Tool. Način na koji se vrši testiranje jeste da se softver prvenstveno testira sa testnim podacima koji statistički modeliraju radno okruženje. Potom se neuspjesi uspoređuju i analiziraju. Iz izračunatih podataka izračunava se procjena stope neuspjeha programa. Statistička metoda za ispitivanje mogućih putanja izračunava se izgradnjom algebarske funkcije. Cilj statičke verifikacije koja je prikazana u ovom seminarkom radu jeste prikazati koliko je naš demo projekat zapravo složen, te koliko je održiv po pitanju izmjena i snalaženja u softveru. Prije provođenja statičkog ispitivanja ključno je da tim sakupi sve potrebne i relevantne informacije i znanje o tehnici, što će programerima pomoći u donošenju bolje odluke, kao i omogućiti im da budu pripremljeni sa potrebnim alatima. Također, ono smanjuje troškove prerade jer identificira nedostatke u ranim fazama ciklusa razvoja softvera. Povratne infromacije dobivene ovim testiranjem pomažu poboljšanju funkcioniranja postupka, što timu programera dodatno pomaže da se izbjegnu slični nedostatci i problemi. Statičko testiranje pruža povećanu svijest o različitim pitanjima kvalitete softvera, poboljšava komunikaciju o kritičnim i važnim informacijama među članovima tima, te značajno smanjuje napore za preradu, što dodatno promiče produktivnost razvoja. Neke od mana statičkog testiranja su da proces statičkog ispitivanja može dugo trajati ukoliko se on provodi ručno bez korištenja alata za testiranje, te sprečavaju otkrivanje ranjivosti uvedenih u runtime okruženje. Prije nego što zaključimo raspravu o statičkom testiranju, izuzetno je važno spomenuti popularne alate koje softverski testeri koriste u cijelom svijetu za postupak statičkog testiranja. Danas, kada je tržište prepuno raznih open source i dobro plaćenih alata, jako je bitno nabaviti sigurne i pouzdane alate koji svojim korisnicima nude izvanredne prednosti. Neki od popularnih alata koji se koriste za statičko testiranje su: NDepend, Code Analaysis Tool, Veracode, Gamma, Coverty, te mnogi drugi. Unatoč činjenici da statičko testiranje zahtjeva dugo vremena provedeno u raspravama i sastancima, dobro je uložiti vrijeme kako bi se spriječilo da se oštećenja pojave u posljednim fazama razvoja proizvoda. Stoga se statičko testiranje s pravom smatra glavnim korakom prema razvoju aplikacije sa nula grešaka.

## 3.1 Ciklomatična složenost

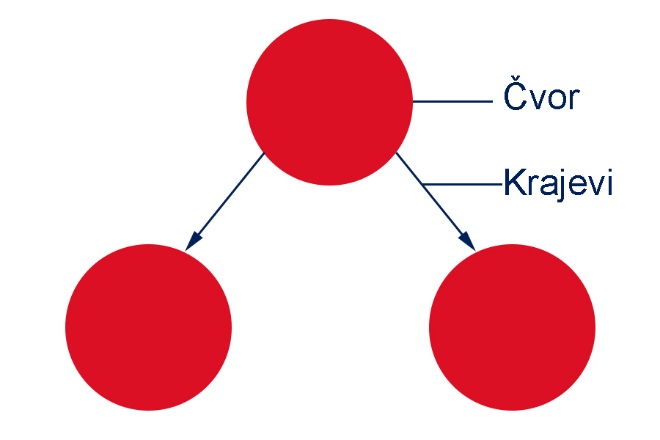
U današnjem svijetu učinkovitost tima za testiranje se procjenjuje kroz otkrivene nedostatke naspram testnih slučajeva ili scenarija. Slično tome, trebao bi postojati pristup pomoću kojeg se može izračunati složenost koda koji je napisao razvojni tim, a koji bi mogao biti povezan sa brojem pogrešaka kodiranja. Kao rezultat, dolazi do pojave ciklomatične složenosti koja predstavlja mjerenje složenosti izvornog koda. Izračunava se putem grafikona kontrolnog toka koji je razvijen na temelju izvornog koda koji mjeri broj linearno nezavnih staza kroz programski modul. Nezavisna staza definirana je kao staza koja ima barem jedan kraj koji prethodno nije pređen ni na jednom drugom putu. Ciklomatična složenost može se izračunati s obzirom na funckije, module ili klase unutar programa. U ovom dijelu rada ćemo prikazati koncept i formulu za izračun ciklomatične složenosti. Cilj je da ciklomatična složenost bude što niža za najbolji izvorni kod. Ako je dobivena vrijednost niska, rizik za promjenu koda je manji i kod će biti lakše razumijeti i održavati. Na primjer, ako izvorni kod ne sadrži nijedan uvjet unuatr kontrolnog toka, tada će njegova ciklomatična složenost biti 1, a izvorni kod sadrži samo jednu stazu. Slično tome, ako izvorni kod sadrži jedan uvjet, tada će ciklomatična složenost biti 2 jer će biti dvije staze, jedna istinita, a druga neistinita. Matematički, za strukturirani program, usmjereni graf kontrolnog toka je kraj koji spaja dva osnovna bloka programa jer kontrola može prolaziti od prvog do drugog. [3] Dolje je prikazana formula kroz koju se procjenjuje ciklomatična složenost.

**M = E – N + 2 \* P**

**E** = predstavlja broj krajeva na grafu kontrolnog toka

**N** = predstavlja broj čvorova na grafu kontrolnog toka

**P** = predstavlja broj čvorova koji imaju izlazne tačke na grafu kontrolnog toka



Slika 1. Oznaka grafikona kontrolnog toka

Sljedeće korake treba slijediti za izračunavanje ciklomatične složenosti i oblikovanje testnih slučajeva.

Korak 1 – Izrada grafikona sa čvorovima i krajevima iz koda

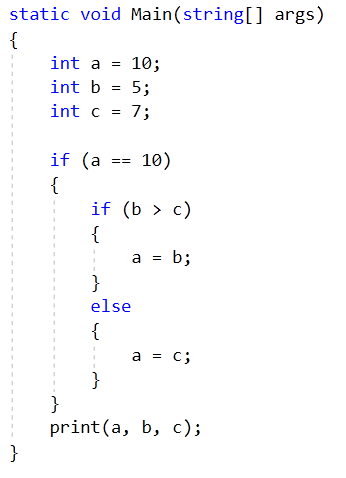
Korak 2 – Identifikacija nezavisnih staza

Korak 3 – Izračunavanje ciklomatične složenosti

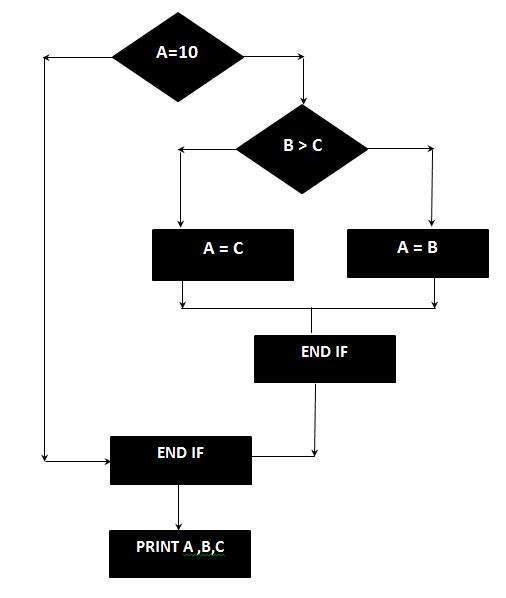
Korak 4 – Dizajn testnih slučajeva

Jednom kada se formira osnovni skup, treba napisati testne slučajeve kako bi se izvršile sve staze.

Ciklomatičnu složenost ćemo razjasniti pomoću donjeg primjera. Ovdje je prikazan izvorni kod za prikaz vrijednosti različitih varijabli na temelju usporedbe. U razmatranje su uzete 3 varijable sa različitim vrijednostima.



Slika 2. Demo izvorni kod napisan u C#



Slika 3. Grafikon kontrolnog toka

Na slici iznad je prikaz grafika kontrolnog toka na temelju demo izvornog koda koji smo gore napisali. Izračunat ćemo broj krajeva, čvorova i izlaznih tačaka da bismo izračunali ciklomatičnu složenost pomoću formule.

Primjetivši gornji dijagram kontrolnog toka, možemo procijeniti sljedeće vrijednosti ciklomatičnih parametara složenosti.

**E** = predstavlja broj krajeva u grafiku kontrolnog toka = 8 krajeva

**N** = predstavlja broj čvorova u grafiku kontrolnog toka = 7 čvorova

**P** = predstavlja broj čvorova koji imaju izlazne tačke u grafiku kontrolnog toka = 1 izlazna tačka

Upotrebom formule za računanje ciklomatične složenosti => **E – N + 2 \* P** dobijemo sljedeće:

**M = 8 – 7 + 2 \* 1**

**M=3**

Ciklomatična složenost može se izračunati ručno ako je program mali. Ako je program složen, potrebno je koristiti automatizirane alate jer uključuje više dijagrama toka. Na temelju broja složenosti, tim može zaključiti o radnjama koje je potrebno poduzeti za mjeru.

Sljedeća tablica daje pregled broja složenosti i odgovarajućeg značenja vrijednosti M.

|  |  |
| --- | --- |
| **Broj složenosti** | **Značenje** |
| 1-10 | Strukturiran i dobro napisan kod  Visoka testabilnost  Trošak i trud su manji |
| 10-20 | Složeni kod  Srednja testabilnost  Trošak i trud su srednji |
| 20-40 | Vrlo složen kod  Niska testabilnost  Trošak i trud su visoki |
| >40 | Nije uopšte testabilan  Vrlo visoki trošak i trud |

*Tabela 1. Tabela pregleda broja složenosti*

Stoga je ciklomatična složenost za gornji graf prihvatljiva vrijednost od 3. Na sličan način, koristeći istu formulu možemo izračunati ciklomatičnu složenost za bilo koji izvorni kod i na temelju rezultata možemo utvrditi rizike i troškove održavanja.

Da bismo razumijeli važnost cikomatične složenosti iz QA perspektive, rezultat koji dobivamo iz formule određuje koliko je ispitivanja potrebno za testiranje tog dijela izvornog koda. Na primjer, ako je broj veći, taj dio zahtijeva dubinsko ispitivanje u odnosu na kod koji ima nižu ciklomatičnu slloženost. Opseg ispitivanja i metode bile bi definirane prema složenosti funkcionalnosti. Grafik kontrolnog toka za ispitivani kod može se dobiti od razvojnog programera za izračin broja krajeva, čvorova i izlaznih tačaka. Također, može se odrediti broj testnih slučajeva za maksimalno pokrivanje testiranja na temelju staza u grafiku kontrolnog toka i grana. Drugim riječima, ciklomatična složenost vrlo je korisna za QA u procjeni opsega ispitivanja, utvrđivanju koliko je potreban vješt tester i potreban opseg testiranja za najbolji mogući način pokrivanja koda. Također, to je jedan od najboljih načina na koji programeri mogu natjerati QA tim da shvati složenost izvornog koda. Izračun ciklomatične složenosti jedan je od standardnih pristupa za izračun složenosti izvornog koda i određivanje rizika koji izvorni kod ima za buduće modifikacije i održavanje. Što je niža ciklomatična složenost, to je bolja kvaliteta koda u smislu složenosti. Ako program ima veliki broj složenosti, vjerovatnost pogreške je velika s povećanim vremenom za održavanje i rješavanjem problema.

## 3.2 Broj linija koda

Najjednostavniji način mjerenja veličine programa je brojanje linija. Ovo je najstarija i najčešće korištena metrika veličine. Broj linija koda ili LOC izgledaju kao jednostavan koncept. Međutim, nije. Postoji nekoliko načina brojanja linija. Ovisno o tome šta se računa dobiva se nizak ili visok broj redaka. Broj fizičkih linija koda (LINES) jednostavan je, ali nije savršen način mjerenja veličine koda. Budući da se logička linija može proširiti na više linija, broj fizičkih linija pretjeruje sa veličinom koda. Česti problem u brojanju linija je i to što su prazne linije (ili razmaci), kao i komentari, uključeni u brojanje. Logičke linija kodne metrike (LLOC) imaju i prednosti i nedostatke. To je jednostavna mjera, lahko razumljiva i široko korištena. Može se koristiti za mjerenje produktivnosti, iako je potrebno biti na oprezu, jer stil programiranja može utjecati na vrijednost. Logička linija je logička linija koda ako ima bilo koji drugi prostor osim komentara ili praznog prostora. Dakle, sve izvršne linije, kao i deklarativne linije, broje se u logičke linije koda. Jedan iskaz ili više njih praćeni komentarom na raju reda je linija koda. Komentar punog reda nije linija koda, kao ni prazan red. Logičke linije koda su dobra mjera veličine programa. Štoviše, dobra je procjena složenosti jedne datoteke, klase ili postupka. Kako na komentare, praznine ne utječu logičke linije koda, to je praktičan način za mjerenje količine stvarnog programa programiranja. Program sa većim brojem logičkih linija koda gotovo sigurno čini više od programa sa nižim brojem logičkih linija.

Brojevi linija su notorni po tome što mogu varirati između programskih jezika i stilova kodiranja. Eksperimenti su u više navrata potvrdili da je napor povezan sa brojem linija koda, onodnosno da programi sa većim vrijednostima broja linija koda trebaju više vremena za razvoj. Prema tome, broj linija koda može biti učinkovit u procjeni napora. Međutim, funkcionalnost je manje povezana sa brojem linija koda, iz razloga što, kvalificirani programeri mogu razviti isu funkcionalnost sa daleko manje koda, tako da jedan program sa manjim brojem linija koda može pokazati više funkcionalnosti od drugog sličnog programa.

## 3.3 Halstead volumen

Metrike Halsteadove složenosti razvijene su kao sredstvo za određivanje kvantitativne mjere složenosti izravno od operatora i operanda u modulu za mjerenje složenosti programskog modula izravno iz izvornog koda. Budući da se primjenjuju na kod, najčešće se koriste kao metrika za održavanje. Postoje dokazi da su Halsteadove mjere korisne i tijekom razvoja za ocjenu kvalitete koda u naprednim aplikacijama. Budući da bi održavanje trebalo biti značajno tijekom razvoja, Halsteadove mjere se trebaju razmotriti za upotrebu tijekom izrade koda radi praćenja trenova složenosti. Ove mjere su jedna od najstarijih mjera složenosti programa. Halsteadove metrike temelje se na interpretaciji izvornog koda kao slijeda tokena i klasificiranju svakog tokena kao operator ili operand.

* **n1** = broj različitih operatora
* **n2** = broj različitih operanda
* **N1** = ukupan broj pojava operatora
* **N2** = ukupan broj pojava operanda

Ostale Halsteadove mjere izvode se iz ove četiri mjere određenim fiksnim formulama. Daljina programa (**N**) je zbroj ukupnog broja operatora i ukupnog broja operanda u programu:

**N=N1+N2**

Veličina vokabulara (**n**) je zbroj broja jedinstvenih operatora i broj jedinstvenih operanda u programu:

**n=n1+n2**

Obim programa (**V**) je informativni sadržaj programa, mjeren je matematičkim bitovima. Izračunava se kao daljina programa (N**)** pomnožena sa logaritmom sa osnovom 2 veličine vokabulara (**n**):

Halsteadov volumen (V) opisuje veličinu implementacije algoritma. Računanje V temelji se na broju izvedenih operacija i operandama kojima se rukovodi u algoritmu. Stoga je Halsteadov volumen manje osjetljiv na izgled koda od broja linija koda. Volumen neke funkcije treba biti najmanje 20, a najviše 1000. Volumen jednostruke funkcije bez parametara koja ije prazna; iznosi oko 20. Volumen veći od 1000 govori da funkcija vjerovatno čini previše stvari. Volumen datoteke trebao bi biti najmanje 100, a najviše 8000. Ta se ograničenja temelje na količinama izmjerenim za datoteke čiji je broj linija i složenost blizu preporučenih granica.

## 3.4 Indeks održivosti

Predloženo je nekoliko modela i metodologija za održavanje kako bi se dizajnerima pomoglo u izračunavanju plana održivosti softvera radi razvoja boljih i poboljšanih softverskih sistema. Održavanje se definira kao postupak promjene softverskog sistema ili komponente nakon isporuke za ispravljanje grešaka, poboljšanje radne karakteristike ili drugih atributa ili se prilagoditi promijenjenom okruženju. [4] Najšire korištena softverska metrika koja kvantificira održivost je poznata kao indeks održivosti (MI). Indeks održivosti (MI) je vrijednost s jednim brojem za procjenu relativne održivosti koda. Indeks održivosti izračunava se određenim formulama pomoću parametara broj linija koda, ciklomatične složenosti i Halstead vrijednosti. Halsteadova metrika temelji se na interpretaciji izvornog koda kao slijeda tokena i klasificiranju svakog tokena kao operator ili operand. [5] Te se metrike izračunavaju statički iz izvornog koda. Namjera održivosti mjerenja i praćenja imaju za cilj smanjiti ili preokrenuti tendenciju sistema prema „entropiji koda“ ili degradiranom integritetu i naznačiti kada postaje jeftinije i/ili manje rizično prepisati kod umjesto promijeniti ga. [6] Vrijednost indeksa održivosti iznad 85 ukazuje da je softver visoko održiv, vrijednost između 85 i 65 sugerira umjereno održavanje, a vrijednost ispod 65 ukazuje da je sistem teško održavati. Indeks održivosti softvera se koristi u nekoliko automatiziranih softverskih metričkih alata, uključujući razvojno okruženje Microsoft Visual Studio, koje koristi vrijednosti između 0 i 100. Da bismo u potpunosti iskoristili indeks održivosti, okruženje za održavanje mora omogućiti prepisivanje modula kad postane mjerljivo neizdrživo. Svrha mjerenja indeksa održivosti je prepoznavanje rizika; kada se utvrdi neprihvatljivo rizičan kod, treba ga prepisati.

Indeks održavanje izračunava se za svaku funkciju, klasu, strukturu, za svaku datoteku i za sve datoteke zajedno. Kod računanja indeksa održivosti postoje tri mjere:

* **MIwoc** – indeks održivosti bez komentara
* **MIcw** – težina komentara indeksa održivosti
* **MI** – indeks održivosti = MIwoc + MIcw

Opća formula za računanje indeksa održivosti je sljedeća:

**MIwoc = 171 - 5.2 \* ln(aveHV) – 0.23 \* aveM – 16.2 \* ln(aveLOC)**

**MIcw = 50 \* sin(sqrt(2.4)\*perCM)**

**MI = MIwoc + MIcw**

* **aveHV** = prosječni Halstead volumen po modulu
* **aveM** = prosječna ciklomatična složenost po modulu
* **aveLOC** = prosječan broj linija po modulu
* **perCM** = prosječni postotak linija komentara po modulu

# 5. NDepend i Code Analysis Tool

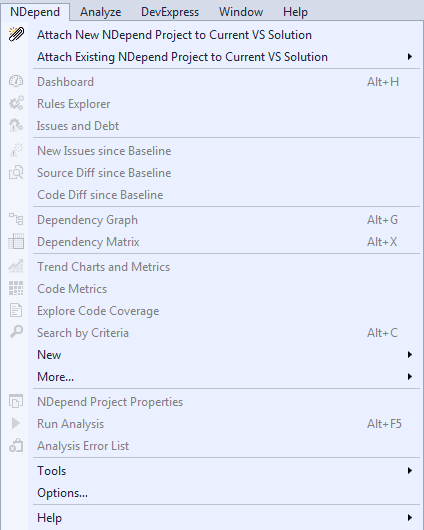
U svijetu softverskog inžinjeringa prilično je normalno raditi s velikim i složenim projektima. Kada projekat postaje sve veći, svaka nova značajka dodana u sistem čini bazu kodova složenijom i težom za održavanje, osim ako se pažljivo ne izrađuje arhitektura aplikacija. U stvarnosti su prije mnogo godina izrađene brojne softverske aplikacije sa lošom arhitekturom, a odrđavanje njih je jako teško, što najčešće vodi do potpunog prepisivanja softvera. I danas postoje mnogo projekata koji su slabo dizajnirani i to zbog toga što ne pregledavaju kod, a niko ne gleda cjelokupnu arhitekturu sistema. Za izradu dobrog sistema potrebni su dobri arhitekti sofvera sa iskustvom iz stvarnog svijeta koji znaju kako izgraditi bolji sistem na temelju poslovnih ciljeva koji su fleksibilni i koji se mogu lahko dodati ili ukloniti. Međutim, preusmjeravanje postojećeg sistema na softver sa boljim perfomansama i većom fleksibilnosti je drugačija stvar pogotovo ako nismo upoznati sa načinom na koji cijeli sistem funkcionira. Drugim riječima, vrlo je teško odrediti specifične probleme i predložiti potencijalna rješenja bez preciznog alata. Korištenje pravog alata za analizu sistema pomaže nam da lahko identificiramo koja se područja trebaju popraviti i sugerira gdje promijeniti dizajn tako da poboljša postojeću arhitekturu prije nego što postane veća i složenija za održavanje.

NDepend je alat za analizu statičkog koda u kojem je primarni cilj omogućiti analiziranje aplikacije i stjecanje jasne predodžbe o složenosti, tehničkom dugu, prekršenim pravilima i njihovoj ozbiljnosti, unakrsnim ovisnostima između objekata, stepenu povezanosti među njima. Stoga, NDepend je alat dizajniran za dobivanje statičke analize koji nam pomaže dizajnirati bolji sistem, a ne pisati ispravniji kod. NDepend je komercijalni alat i uglavnom radi kao dodatak za Visual Studio, tako da je dostupan za sva nedavna izdanja Visual Studio-a. Može se koristiti i nezavisno od Visual Studio-a sa vlastitim korisničkim sučeljem zvanim Visual NDepend. Srž NDepend alata je statička analiza koja može otkriti neka svojstva koda, i što je još važnije, poštuju li se određena pravila. Temelj statičke analize su kodni mjerni podaci, oni se koriste za definiranje pravila i stvaranje vizualizacija projekta. Oni se kreću od osnovnih statističkih podataka o cijelom projektu i njegovim sastavnim dijelovima, do složenih strukurnih mjera. Primjer prvog je broj linija koda koji se mjeri za sve razine: aplikaciju, assembler, namespace, klasu i metodu. Primjer druge vrste mjera je ciklomatična složenost koja je objašnjena ranije. Važna napomena je da su neke od tih mjera dostupne samo za C# programski jezik. NDepend je alat dizajniran za višu razinu statičke analize koji nam pomaže da prepoznamo potencijalne probleme unutar aplikacije. Daje nam visoku razinu slike o kvaliteti koda koji nam pomaže donositi odluke o tome kako bolje dizajnirati softver. Da sumiramo, NDepend alat je uglavnom za arhitekte softvera ili za ljude unutar određene organizacije koji donose odluke o dizajniranju softvera. To je alat dizajniran za višu razinu statičke analize koji nam pomaže dizajnirati bolji sistem, a ne pisati ispravniji kod. Može spriječiti da se sistem komplicira za održavanje kako raste, prepoznavanjem pogrešaka unutar koda u ranoj fazi i provođenjem pravila kvalitete koda. Također, može prepoznati da li naš kod poštuje standardna pravila kvalitete, te može izračunati standardne metrike sa različitim vizualizacijama kako bismo što bolje razumijeli cijeli svoj projekat.

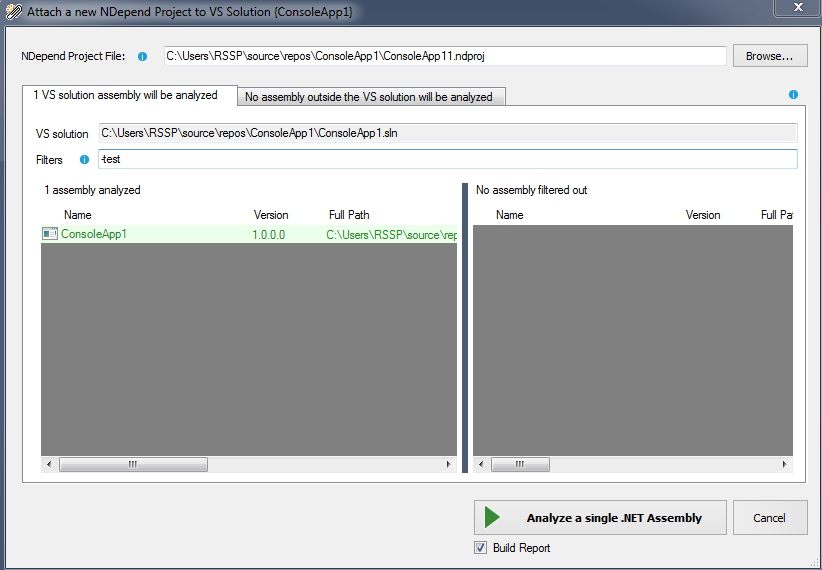
Visual Studio Code Analysis alat pomaže programerima da otkriju perfomanse, sigurnost, dizajn, upravljanje iznimkama, globalizaciju i druge probleme za poboljšanje kvalitete koda. Analiza koda može se izvesti na cjelovitom rješenju ili na određenom projektu u aplikaciji. Code Analysis alat je moguće pokrenuti automatski sa svakom izvedbom projektnog koda, a može se pokrenuti i ručno. Da bi se izvršila analiza koda pomoću Code Analysis alata potrebno je da se projekat otvori u Visual Studio 2012 ili veće, potom je potrebno sa glavnog menija otvoriti dio „Analyze“ i odabrati opciju Run Code Analysis on Solution. Da bismo konfigurirali Code Analysis alat za određeni projekat ili za čitavu aplikaciju, desnim klikom miša kliknemo na željeni projekat i odaberemo „Properties“ i zatim odaberemo „Code Analysis“. U ovom prozoru programer može postaviti postavke konfiguracije i platforme. A također, može odabrati skup pravila ili višestrukih skupova pravila za analizu koda sa zadanog popisa skupa pravila. Da bismo automatski pokrenuli analizu koda tijekom izrade izvornog koda, potrebno je označiti opciju „Enable Code Analysis on Build“ u prozoru za analizu koda. Code Analysis može da izvrši analizu koda pisanog u C, C++, C# programskom jeziku. Parametri koje dobijemo prilikom analize koda u ovom alatu su: indeks održivosti, ciklomatična složenost, broj linija koda, dubina naslijeđivanja i broj spojenih klasa u jednu cijelinu.

6. Praktični primjeri

Prije nego što počnemo raditi sa alatom NDepend, isti je potrebno preuzeti sa oficijalne web stranice. Da bismo instalirali NDepend, potrebno je raspakirati datoteke u mapu na računaru, te potom pokrenuti NDepend Visual Studio Installer. Nakon uspješne instalacije, možemo početi raditi sa alatom NDepend. Kada želimo da uradimo statičku analizu u alatu NDepend za određeni projekat na kojem radimo, prvobitno je potrebno otvoriti taj projekat u Visual Studio. Potom, potrebno je odabrati opciju u NDepend padajućem meniju *Attach New NDepend Project to Current VS Solution* kako bi pristupili formi za odabir projekata za statičku analizu.

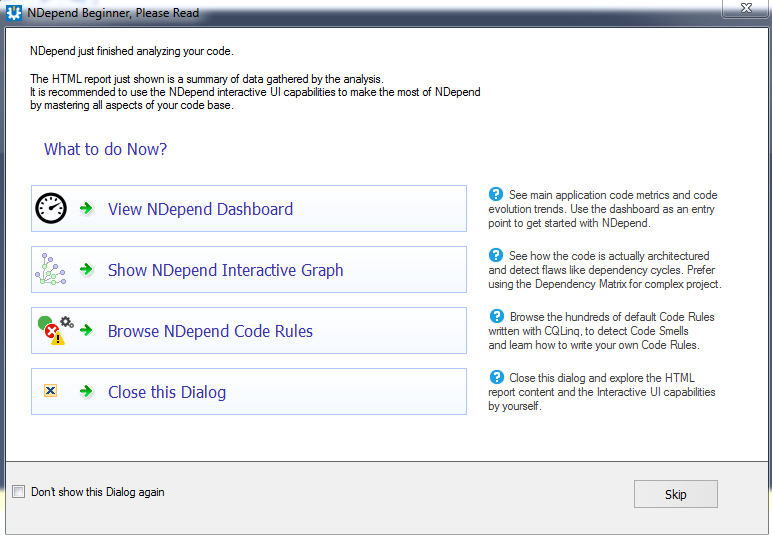


Slika . NDepend glavni izbornik u VS



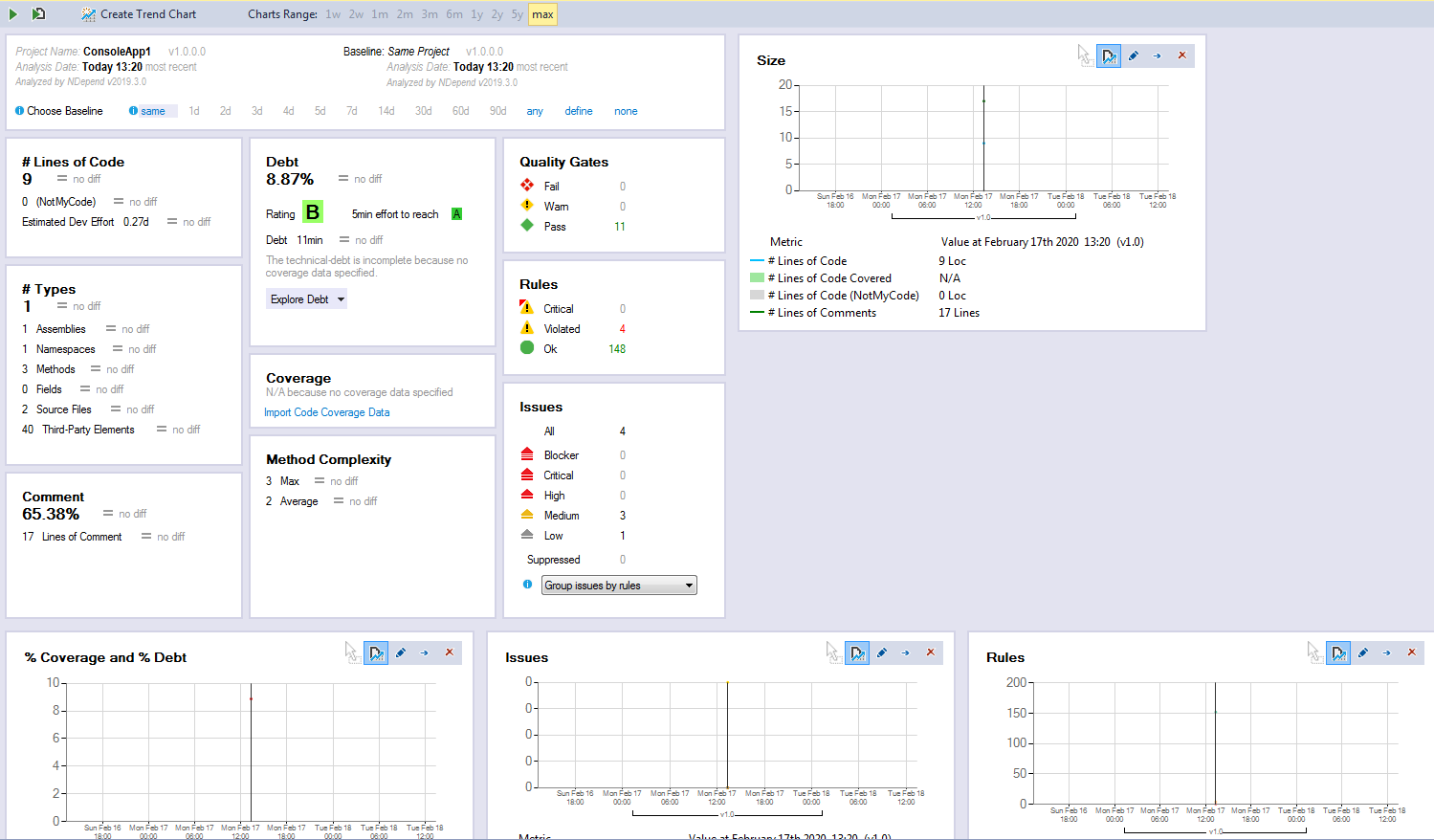
Slika . NDepend forma za odabir projekta

Na slici iznad prikazana je forma gdje možemo odabrati projekat unutar solution-a koji želimo da testiramo. Nakon što odaberemo željeni projekat potrebno je pokrenuti analizu na dugme *Analyze a single .NET Assembly.*



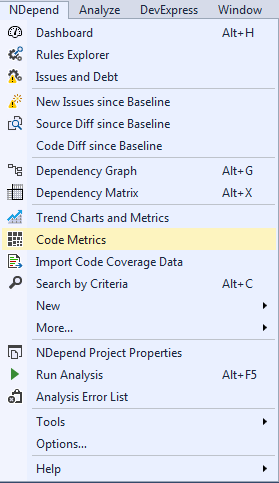
Slika . Forma za odabir načina prikaza rezultata

Nakon što se završi analiza, prikaže se forma koja nas obaviještava o završetku analize, te nam nudi da odaberemo način na koji možemo pregledati rezultate koje dobijemo.



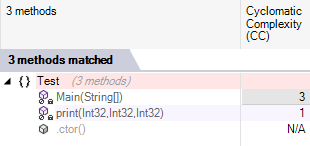
Slika . NDepend Dashboard

Ukoliko odaberemo opciju View NDepend Dashboard dobit ćemo pregled vrijednosti nekih osnovnih parametara za ocjenu kvalitete softvera. Ovdje možemo da vidimo informacije o broju linija koda, kompleksnosti metoda, broju linija komentara i mnoge druge.



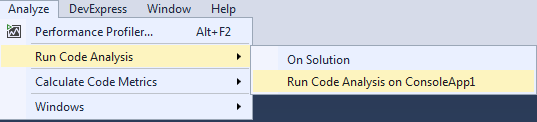
Slika . Odabir opcije Code Metrics u NDepend izborniku

Odabirom opcije Code Metrics iz NDepend izbornika moguće je pregledati neke detaljnije informacije o ciklomatičnoj složenosti, odnosno pregledati vrijednosti iste za svaku od implementiranih metoda unutar softvera. Pregled ciklomatične složenosti za sve implementirane metode je prikazan na slici ispod za demo kod koji je prikazan ranije.



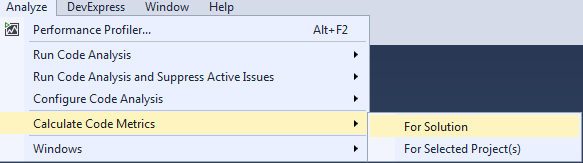
Slika . Pregled ciklomatične složenosti u NDepend alatu

Code Analysis je alat integrisan unutar Visual Studio-a. Na slici ispod je prikazan način pokretanja ovog alata.



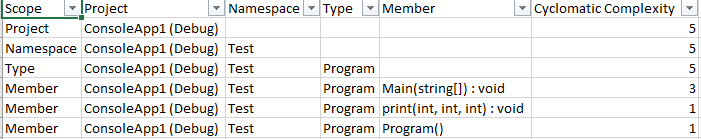
Slika . Način pokretanja Code analysis alata

Nakon što se završi analiza koda,pokretanje pregleda metrike i vrijednosti svih željenih parametara je prikazano na slici ispod.

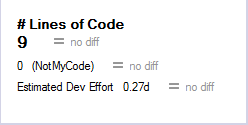


Slika . Pregled kodne metrike unutar Code Analysis alata

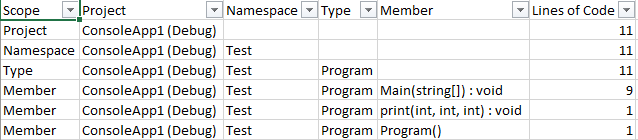
Kao što je već rečeno i alat Code Analysis Tool nudi pregled i analizu ciklomatične složenosti softvera. Stoga, ispod možemo vidjeti analizu Code Analysis Tool-a za ciklomatičnu složenosti, te se uvjeriti da je rezultat identičan kao i kod NDepend alata.



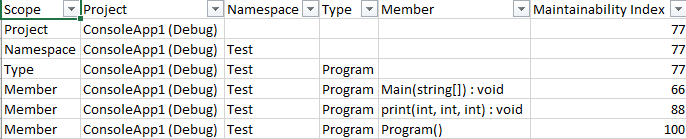
Slika . Pregled ciklomatične složenosti unutar Code Analysis alata



Slika . Pregled broja linija unutar NDepend alata



Slika . Pregled broja linija unutar Code Analysis alata



Slika . Pregled indeksa održivosti unutar Code Analysis alata

# 7. Zaključak

U bilo kojoj metodologiji razvoja softvera, i postupak verifikacije i postupak validacije se provodi kako bi se potvrdilo da konačni softver ima sve zahtjeve ispravno implementirane. Aktivnosti verifikacije spadaju u kategoriju statičkog ispitivanja. Tijekom statičkog ispitivanja potrebno je provjeriti da li rad koji se obavlja ispunjava zadane standarde organizacije. Analiza statičkog koda važan je dio osiguravanja funkcioniranja aplikacija kako se očekuje. To ne samo da poboljšava ukupnu brzinu razvojnog tima, već i smanjuje rizike s oslobađanjem opasnog softvera. Iako je neka analiza uvijek bolja nego nijedna analiza, statička analiza bi trebala biti integrirana u veću infrastrukturu za testiranje softvera kako bi se povećala učinkovitost prakse. Glavna prednost statičkih tehnika verificiranja je ta što oni mogu analizirati softver bez izvršenja. Ovo je posebno korisno ako softver još uvijek nije u izvršnom stanju, nema unosa ili je izvršenje skupo. Statičke tehnike mogu pronaći suptilne pogreške koje mogu biti zanimljive čak i iskusnim programerima. Nadalje, čitav proces može biti u mnogo slučaveja potpuno automatiziran i integriran u razvojni proces. Statičko ispitivanje nije zamjena za dinamičko ispitivanje, iz razloga što oba pronalaze različite vrste nedostataka. NDepend i Code Analysis alat nude kompletnu analizu i ispitivanje skupa za .NET aplikacije, kao i napredni analitički sistem. Kombinacija ovih tehnologija omogućuje timovima softverskog inžinjeringa ubrzanje isporuke softvera, istovremeno osiguravajući da njihove aplikacije funkcioniraju onako kako se i očekivalo. Ovi alati podržavaju veliki broj kodnih metrika, omogućavaju vizualizaciju ovisnosti pomoću grafova. Parametre koje smo testirali u okviru statičke analize, kao što su ciklomatična složenost i indeks održivosti su odabrani iz razloga što su se pokazali kao odlični pokazatelji kvalitete softvera, te njegove kompleksnosti.

# 8. Reference

[1] A Meneely, Perspectives on Data Science for Software Engineering, pp 283-287, 2016.

[2] Schumann J.M. (2001) Formal Methods in Software Engineering. In: Automated Theorem Proving in Software Engineering.

[3] M. Shepperd (1988) Software Engineering Journal: A critic of cyclomatic complexity as a software metric

[4] D. M. Coleman, D. Ash, B. Lowther, and P. W. Oman, “Using Metrics to Evaluate Software System Maintainability.” IEEE Computer, vol. 27, no. 8, pp. 44–49, 1994.

[5] Kurt D. Welker, Idaho National Engineering and Environmantal Laboratory: [The Software Maintainability Index Revisited](http://web.archive.org/web/20021120101304/http:/www.stsc.hill.af.mil/crosstalk/2001/08/welker.html) (August 2001)

[6] Ebeling C. E., An Introduction to Reliability and Maintainability Engineering, McGraw-Hill Companies, Inc., Boston 1997.