Univerzitet u Sarajevu
Elektrotehnički fakultet
Sarajevo

Verifikacija i Validacija Softvera

Projektni zadatak – II dio

TaskManagerApp – analiza jedne metode

Student: Aldin Velić

Broj indeksa: 19761

Odabrana metoda: EvaluateTaskStatus(Task task)

1 McCabe metrike i kontrolni graf

1.1 Izvorni kod metode

```
public string EvaluateTaskStatus(Task task)
{
    if (task == null) return "Invalid";

    if (task.Status == TaskStatus.Done)
        return "Completed";

    if (!task.DueDate.HasValue)
        return "No deadline";

    if (task.DueDate.Value < DateTime.Now && task.Status !=
       TaskStatus.Done)
    {
        if (task.Priority == TaskPriority.Critical)
            return "CRITICAL - Overdue!";
        else if (task.Priority == TaskPriority.High)
            return "High Priority Overdue";
        else if (task.Priority == TaskPriority.Medium)
            return "Medium Priority Overdue";
        else
            return "Overdue";
    }
    else if (task.DueDate.Value > DateTime.Now && task.Status ==
       TaskStatus.InProgress)
    {
        if (task.Priority == TaskPriority.High || task.Priority
           == TaskPriority.Critical)
            return "On track (High)";
        else if (task.Priority == TaskPriority.Low)
            return "On track (Low)";
        else
            return "On track (Normal)";
    }

    return "In progress";
}
```

1.2 Izračun ciklomatične kompleksnosti

Ciklomatična kompleksnost se računa kroz 3 pristupa:

- Prebrojavanjem odluka
- Formulom $V(G) = E - N + 2$
- Prebrojavanjem regiona kontrolnog grafa

Broj čvorova (N)	26
Broj ivica (E)	38
Broj odluka	13
Ciklomatična kompleksnost	**14**

1.3 Kontrolni graf

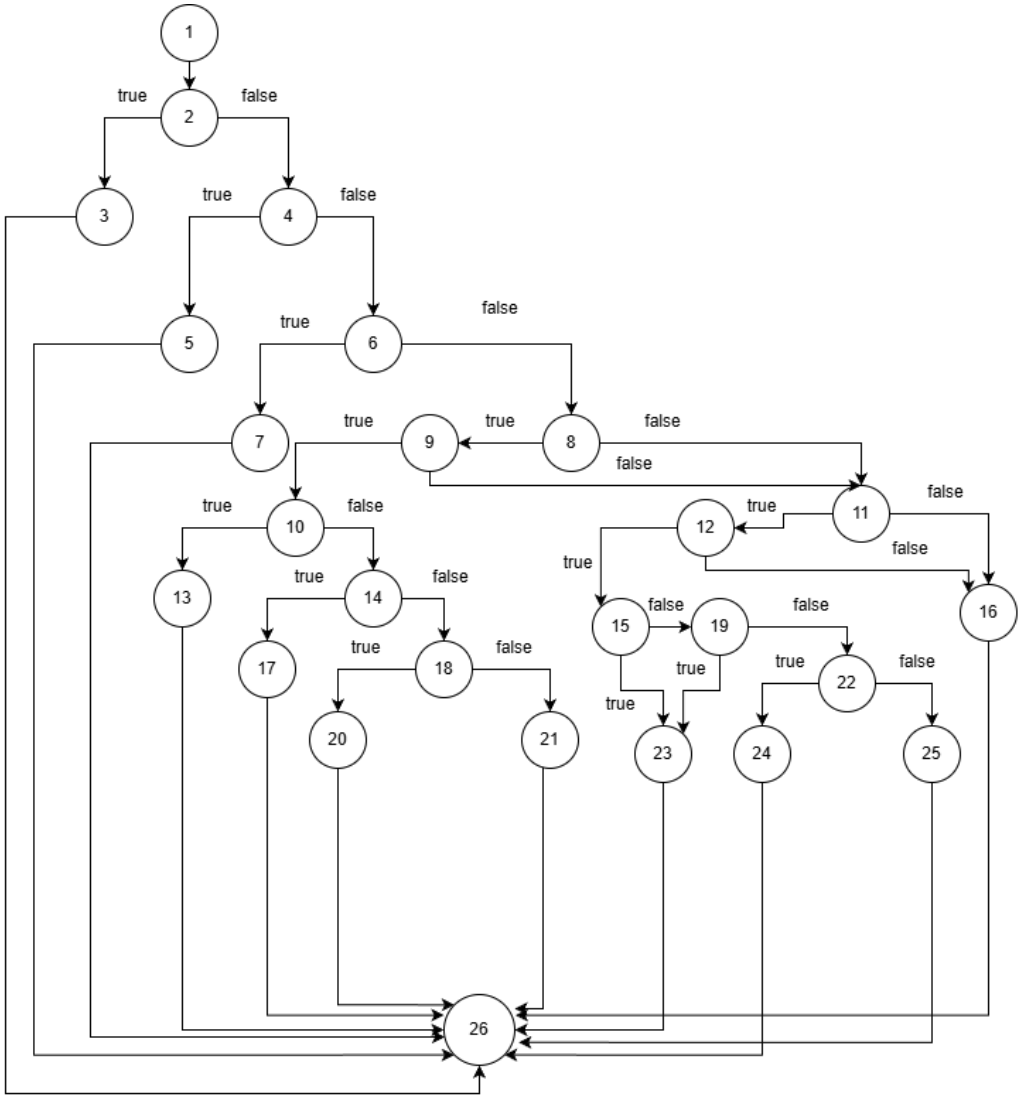


Figure 1: Kontrolni graf odabrane metode

1.4 Poredjenje sa Code Metrics alatom

TaskAnalyzer		57	49	1	15	142	57
repository : ITaskRepository		100	0		1	1	0
TaskAnalyzer(ITaskRepository)		91	2		3	4	1
EvaluateTaskStatus(Task) : string		57	14		5	34	11
AnalyzeTeamPerformance() : string		52	11		9	32	16
GetTasksSummaryForUser(int) : Dictionary<string, int>		54	8		11	30	14
PredictDelayRisk(Task) : string		53	14		6	33	15

Figure 2: Prikaz kompleksnosti iz Visual Studio Code Metrics alata

Iz dobijenog izvještaja se vidi da metoda `EvaluateTaskStatus` ima McCabe ciklomatičnu kompleksnost vrijednosti **14**, što je u potpunosti u skladu sa ručno izračunatom vrijednošću prikazanom u prethodnoj tabeli. Samim tim možemo potvrditi ispravnost ručne analize.

1.5 Diskusija o kvalitetu koda

Dobijena vrijednost kompleksnosti 14 ukazuje na metodu srednje složenosti. Kod je čitljiv, ali sadrži veliki broj granularnih uslova i više nivoa ugniježdenosti, što otežava održavanje i povećava vjerovatnoću greške pri budućim izmjenama.

Moguća poboljšanja:

- Grupisanje logike u manje pomoćne metode (Refactoring – Extract Method)
- Korištenje `switch` izraza radi bolje preglednosti
- Smanjenje dupliranja uslova i logičkih provjera

Zaključak je da kod funkcioniše korektno, ali postoji prostor za refaktorisanje kojim bi se smanjila kompleksnost, povećala čitljivost i olakšalo testiranje i održavanje.

2 White-box testiranje

U ovom poglavlju nad metodom `EvaluateTaskStatus(Task task)` primijenjene su white-box tehnike testiranja. **Važno:** radi striktne separacije, **testovi se ne dijele** izmedju tehnika (svaka tehnika ima vlastiti minimalni skup testova).

U tabelama se koristi referentno vrijeme t_0 (vrijednost `DateTime.Now` u trenutku poziva). Rokovi se zadaju relativno:

$$t_0^- = t_0 - 1 \text{ dan}, \quad t_0^+ = t_0 + 1 \text{ dan}.$$

2.1 Tehnika 1: Obuhvat iskaza/linija (Statement/Line coverage)

Minimalnost: metoda ima **11 različitih `return` iskaza**. Jedan test može aktivirati tačno jedan `return`, pa je minimalan broj testova **11**.

Test (SC)	Ulaz	Očekivan izlaz
SC1	`task = null`	`"Invalid"`
SC2	`Status=Done`	`"Completed"`
SC3	`Status=InProgress, DueDate=null`	`"No deadline"`
SC4	`Status=InProgress, DueDate=`t_0^-`, Priority=Critical`	`"CRITICAL - Overdue!"`
SC5	`Status=InProgress, DueDate=`t_0^-`, Priority=High`	`"High Priority Overdue"`
SC6	`Status=InProgress, DueDate=`t_0^-`, Priority=Medium`	`"Medium Priority Overdue"`
SC7	`Status=InProgress, DueDate=`t_0^-`, Priority=Low`	`"Overdue"`
SC8	`Status=InProgress, DueDate=`t_0^+`, Priority=High`	`"On track (High)"`
SC9	`Status=InProgress, DueDate=`t_0^+`, Priority=Low`	`"On track (Low)"`
SC10	`Status=InProgress, DueDate=`t_0^+`, Priority=Medium`	`"On track (Normal)"`
SC11	`Status=ToDo, DueDate=`t_0`, Priority=Medium`	`"In progress"`

Table 1: Minimalni testovi za statement/line coverage (SC1–SC11)

Ovime su sve izvršive linije metode pokrivene barem jednom (100% statement/line coverage).

2.2 Tehnika 2: Obuhvat grana/odluka (Branch/Decision coverage)

Minimalnost: da bi se svaka odluka izvršila i kao `true` i kao `false`, a unutrašnje `if/else-if/else` grane u `Overdue` i `On track` blokovima sve obišle, potrebno je minimalno **11** testova.

Test (BC)	Ulaz	Očekivan izlaz
BC1	`task=null`	`"Invalid"`
BC2	`Status=Done`	`"Completed"`
BC3	`Status=InProgress, DueDate=null`	`"No deadline"`
BC4	`Status=InProgress, DueDate=`t_0^-`, Priority=Critical`	`"CRITICAL - Overdue!"`
BC5	`Status=InProgress, DueDate=`t_0^-`, Priority=High`	`"High Priority Overdue"`
BC6	`Status=InProgress, DueDate=`t_0^-`, Priority=Medium`	`"Medium Priority Overdue"`
BC7	`Status=InProgress, DueDate=`t_0^-`, Priority=Low`	`"Overdue"`
BC8	`Status=InProgress, DueDate=`t_0^+`, Priority=High`	`"On track (High)"`
BC9	`Status=InProgress, DueDate=`t_0^+`, Priority=Low`	`"On track (Low)"`
BC10	`Status=InProgress, DueDate=`t_0^+`, Priority=Medium`	`"On track (Normal)"`
BC11	`Status=InProgress, DueDate=`t_0`, Priority=Low`	`"In progress"`

Table 2: Minimalni testovi za branch/decision coverage (BC1–BC11)

Skup BC1–BC11 osigurava da svaka odluka i svaka grana (uključujući sve prioritete u oba bloka) bude izvršena barem jednom.

2.3 Tehnika 3: Obuhvat uslova (Condition coverage)

Obuhvat uslova zahtijeva da svaki elementarni uslov poprimi vrijednosti `true` i `false` *kada je dostižan i evaluiran.*

Elementarni uslovi koji se pojavljuju u metodi (po mjestu gdje se evaluiraju) su:

- U1: `task == null`
- U2: `task.Status == Done`
- U3: `!task.DueDate.HasValue`
- U4: `task.DueDate.Value < Now`
- U5: `task.Status != Done` (u D4)
- U6: `task.DueDate.Value > Now`
- U7: `task.Status == InProgress` (u D5)
- U8: `Priority == Critical` (overdue)
- U9: `Priority == High` (overdue i on-track OR)
- U10: `Priority == Medium` (overdue)

- U11: `Priority == Critical` (on-track OR)
- U12: `Priority == Low` (on-track)

Napomena (dostižnost): U5 (`Status != Done`) u okviru odluke `(DueDate < Now && Status != Done)` ne može biti `false` na dostižnim putevima, jer bi `Status == Done` prethodno aktivirao `return "Completed";`. Zato se za U5 pokriva maksimalno moguće (dostižno) ponašanje.

Minimalnost: zbog short-circuit evaluacije i meusobno isključivih grana, minimalan skup za pokrivanje svih dostižnih T/F kombinacija ovdje je **13** testova.

Test (CC)	**Ulaz**	**Očekivan izlaz**
CC1	`task=null`	`"Invalid"`
CC2	`Status=Done`	`"Completed"`
CC3	`Status=InProgress, DueDate=null`	`"No deadline"`
CC4	`Status=InProgress, DueDate=`t_0^-`, Priority=Critical`	`"CRITICAL - Overdue!"`
CC5	`Status=InProgress, DueDate=`t_0^-`, Priority=High`	`"High Priority Overdue"`
CC6	`Status=InProgress, DueDate=`t_0^-`, Priority=Medium`	`"Medium Priority Overdue"`
CC7	`Status=InProgress, DueDate=`t_0^-`, Priority=Low`	`"Overdue"`
CC8	`Status=InProgress, DueDate=`t_0^+`, Priority=High`	`"On track (High)"`
CC9	`Status=InProgress, DueDate=`t_0^+`, Priority=Critical`	`"On track (High)"`
CC10	`Status=InProgress, DueDate=`t_0^+`, Priority=Low`	`"On track (Low)"`
CC11	`Status=InProgress, DueDate=`t_0^+`, Priority=Medium`	`"On track (Normal)"`
CC12	`Status=InProgress, DueDate=`t_0`, Priority=Medium`	`"In progress"`
CC13	`Status=ToDo, DueDate=`t_0^+`, Priority=Medium`	`"In progress"`

Table 3: Minimalni testovi za condition coverage (CC1–CC13)

Uslov	**TRUE testovi**	**FALSE testovi**
U1: `task==null`	CC1	CC2–CC13
U2: `Status==Done`	CC2	CC3–CC13
U3: `!DueDate.HasValue`	CC3	CC4–CC13
U4: `DueDate<Now`	CC4–CC7	CC8–CC13
U5: `Status!=Done` (u D4)	CC4–CC7	*nedostižno*
U6: `DueDate>Now`	CC8–CC11, CC13	CC12
U7: `Status==InProgress` (u D5)	CC8–CC11	CC13
U8: `Priority==Critical` (overdue)	CC4	CC5–CC7
U9: `Priority==High`	CC5, CC8	CC6–CC7, CC9–CC11
U10: `Priority==Medium` (overdue)	CC6	CC7
U11: `Priority==Critical` (ontrack OR)	CC9	CC10–CC11
U12: `Priority==Low` (ontrack)	CC10	CC11

Table 4: Mapa TRUE/FALSE za condition coverage (dostižno ponašanje)

2.4 Tehnika 4: MCDC (Modified Condition/Decision Coverage)

MCDC zahtijeva da svaka elementarna komponenta složenog uslova može nezavisno uticati na ishod odluke. Složene odluke u metodi su:

- D4: `(DueDate < Now) && (Status != Done)`
- D5: `(DueDate > Now) && (Status == InProgress)`
- D6: `(Priority == High) || (Priority == Critical)` (u On track bloku)

Napomena (dostižnost za D4): komponenta `Status != Done` u D4 ne može biti nezavisno postavljena na `false` na dostižnim putevima, jer bi tada metoda ranije vratila `"Completed"`. Zbog toga se MCDC za D4 radi nad dostižnim dijelom (uticaj `DueDate < Now`).

Minimalnost: za D4 (2 testa), za D5 (3 testa), za D6 (3 testa), uz preklapanje unutar iste tehnike, dobija se minimalno **6** testova.

Test (MC)	Ulaz	Očekivan izlaz
MC1	`Status=InProgress, DueDate=`t_0^-`, Priority=Low`	`"Overdue"`
MC2	`Status=InProgress, DueDate=`t_0`, Priority=Low`	`"In progress"`
MC3	`Status=InProgress, DueDate=`t_0^+`, Priority=High`	`"On track (High)"`
MC4	`Status=InProgress, DueDate=`t_0^+`, Priority=Medium`	`"On track (Normal)"`
MC5	`Status=InProgress, DueDate=`t_0^+`, Priority=Critical`	`"On track (High)"`
MC6	`Status=ToDo, DueDate=`t_0^+`, Priority=Medium`	`"In progress"`

Table 5: Minimalni testovi za MCDC (MC1–MC6)

Odluka/ komponenta	Par testova (mijenja se samo komponenta)	Ishod
D4: `DueDate < Now`	MC1 (`true`) vs MC2 (`false`), `Status!=Done` dostižno konstantno	D4: `true` → `false`
D5: `DueDate > Now`	MC3 (`true`) vs MC2 (`false`), `Status==InProgress=true`	D5: `true` → `false`
D5: `Status == InProgress`	MC3 (`true`) vs MC6 (`false`), `DueDate>Now=true`	D5: `true` → `false`
D6: `Priority == High`	MC3 (`true`, `Critical=false`) vs MC4 (`false`, `Critical=false`)	D6: `true` → `false`
D6: `Priority == Critical`	MC5 (`true`, `High=false`) vs MC4 (`false`, `High=false`)	D6: `true` → `false`

Table 6: MCDC parovi za dostižne složene odluke

2.5 Tehnika 5: Obuhvat petlji (Loop coverage)

Metoda `EvaluateTaskStatus` ne sadrži petlje (`for/while/foreach`), pa je ova tehnika **neprimjenjiva (N/A)** i trivijalno zadovoljena bez dodatnih testova.

2.6 Tehnika 6: Obuhvat toka podataka (Data-flow coverage)

Za data-flow obuhvat posmatrani su ključni def-use tokovi za atribute: `task`, `Status`, `DueDate`, `Priority`. Minimalni cilj je da se svaki od ovih podataka iskoristi (use) u svim glavnim regijama metode: *null provjera, Done, No deadline, Overdue blok, On track blok, default put.*

Minimalnost: zbog meusobno isključivih regija, minimalno je potrebno **8** testova (1 po regiji, uz 3 različita ulaza za On track da bi se aktivirale sve upotrebe `Priority` u tom bloku).

Test (DF)	Ulaz	Očekivan izlaz
DF1	`task=null`	`"Invalid"`
DF2	`Status=Done`	`"Completed"`
DF3	`Status=InProgress, DueDate=null`	`"No deadline"`
DF4	`Status=InProgress, DueDate=`t_0^-`, Priority=Medium`	`"Medium Priority Overdue"`
DF5	`Status=InProgress, DueDate=`t_0^+`, Priority=High`	`"On track (High)"`
DF6	`Status=InProgress, DueDate=`t_0^+`, Priority=Low`	`"On track (Low)"`
DF7	`Status=InProgress, DueDate=`t_0^+`, Priority=Medium`	`"On track (Normal)"`
DF8	`Status=ToDo, DueDate=`t_0`, Priority=Low`	`"In progress"`

Table 7: Minimalni testovi za data-flow coverage (DF1–DF8)

2.7 Tehnika 7: Obuhvat puteva (Path coverage)

Path coverage cilja pokrivanje svih **izvedivih** puteva do povratnih tačaka. U ovoj metodi svaki izvedivi put završava jednim od **11** `return` iskaza, pa je minimalan broj testova opet **11** (jedan po putu).

Test (PC)	Ulaz	Očekivan izlaz
PC1	`task=null`	`"Invalid"`
PC2	`Status=Done`	`"Completed"`
PC3	`Status=InProgress, DueDate=null`	`"No deadline"`
PC4	`Status=InProgress, DueDate=`t_0^-`, Priority=Critical`	`"CRITICAL - Overdue!"`
PC5	`Status=InProgress, DueDate=`t_0^-`, Priority=High`	`"High Priority Overdue"`
PC6	`Status=InProgress, DueDate=`t_0^-`, Priority=Medium`	`"Medium Priority Overdue"`
PC7	`Status=InProgress, DueDate=`t_0^-`, Priority=Low`	`"Overdue"`
PC8	`Status=InProgress, DueDate=`t_0^+`, Priority=High`	`"On track (High)"`
PC9	`Status=InProgress, DueDate=`t_0^+`, Priority=Low`	`"On track (Low)"`
PC10	`Status=InProgress, DueDate=`t_0^+`, Priority=Medium`	`"On track (Normal)"`
PC11	`Status=ToDo, DueDate=`t_0`, Priority=Medium`	`"In progress"`

Table 8: Minimalni testovi za path coverage (PC1–PC11)

2.8 Zaključak za white-box testiranje

Za svaku white-box tehniku konstruisan je **minimalan i potpuno odvojen** skup testova. Postignuto je:

- statement/line coverage: 100% (SC1–SC11),
- branch/decision coverage: 100% (BC1–BC11),
- condition coverage: 100% za dostižne uslove (CC1–CC13),
- MCDC: za dostižne složene odluke (MC1–MC6), uz napomenu o nedostižnosti komponente `Status!=Done` u D4,
- loop coverage: N/A,
- data-flow: pokrivene ključne def-use regije (DF1–DF8),
- path coverage: svi izvedivi putevi (PC1–PC11).

3 Code tuning

U ovom poglavlju nad metodom `EvaluateTaskStatus(Task task)` primijenjene su tri različite tehnike code tuninga, u skladu sa kategorijama iz predavanja (*izrazi*, *logički iskazi* i *metode*).

Cilj je bio:

- smanjiti vrijeme izvršavanja metode pri velikom broju poziva,
- zadržati isto funkcionalno ponašanje,
- poboljšati strukturu i održivost koda (McCabe i MI).

3.1 Korištene tehnike optimizacije

Primijenjene su sljedeće tehnike:

- **Tehnika 1 – tuning izraza**: eliminacija dupliranih izračuna i keširanje vrijednosti (`DateTime.Now`, `task.DueDate.Value`).
- **Tehnika 2 – tuning logičkih iskaza**: preuredjivanje uslova u pomoćne logičke varijable (`isOverdue`, `isFutureInProgress`) i keširanje statusa/prioriteta, čime se pojednostavljuje grananje.
- **Tehnika 3 – tuning metoda**: primjena principa *Extract Method* – izdvajanje generisanja poruka u pomoćne metode `GetOverdueMessage` i `GetOnTrackMessage`, čime se glavna metoda skraćuje i postaje preglednija.

3.2 Izmijenjeni kod

Originalna metoda

```
public string EvaluateTaskStatus(Task task)
{
    if (task == null) return "Invalid";

    if (task.Status == TaskStatus.Done)
        return "Completed";

    if (!task.DueDate.HasValue)
        return "No deadline";

    if (task.DueDate.Value < DateTime.Now && task.Status !=
       TaskStatus.Done)
    {
        if (task.Priority == TaskPriority.Critical)
            return "CRITICAL - Overdue!";
        else if (task.Priority == TaskPriority.High)
            return "High Priority Overdue";
        else if (task.Priority == TaskPriority.Medium)
            return "Medium Priority Overdue";
        else
            return "Overdue";
    }
    else if (task.DueDate.Value > DateTime.Now && task.Status ==
       TaskStatus.InProgress)
    {
        if (task.Priority == TaskPriority.High || task.Priority
           == TaskPriority.Critical)
            return "On track (High)";
        else if (task.Priority == TaskPriority.Low)
            return "On track (Low)";
        else
            return "On track (Normal)";
    }

    return "In progress";
}
```

Nakon tehnike 1 – tuning izraza (EvaluateTaskStatus_T1)

```
public string EvaluateTaskStatus_T1(Task task)
{
    if (task == null) return "Invalid";

    if (task.Status == TaskStatus.Done)
        return "Completed";

    if (!task.DueDate.HasValue)
        return "No deadline";

    var now = DateTime.Now;
```

```
    var due = task.DueDate.Value;

    if (due < now && task.Status != TaskStatus.Done)
    {
        if (task.Priority == TaskPriority.Critical)
            return "CRITICAL - Overdue!";
        else if (task.Priority == TaskPriority.High)
            return "High Priority Overdue";
        else if (task.Priority == TaskPriority.Medium)
            return "Medium Priority Overdue";
        else
            return "Overdue";
    }
    else if (due > now && task.Status == TaskStatus.InProgress)
    {
        if (task.Priority == TaskPriority.High || task.Priority
           == TaskPriority.Critical)
            return "On track (High)";
        else if (task.Priority == TaskPriority.Low)
            return "On track (Low)";
        else
            return "On track (Normal)";
    }

    return "In progress";
}
```

Nakon tehnike 2 – logički iskazi (EvaluateTaskStatus_T2)

```
public string EvaluateTaskStatus_T2(Task task)
{
    if (task == null) return "Invalid";

    var status = task.Status;

    if (status == TaskStatus.Done)
        return "Completed";

    if (!task.DueDate.HasValue)
        return "No deadline";

    var now = DateTime.Now;
    var due = task.DueDate.Value;
    var priority = task.Priority;

    bool isOverdue = due < now;
    bool isFutureInProgress = due > now && status == TaskStatus.
       InProgress;

    if (isOverdue)
```

```
    {
        if (priority == TaskPriority.Critical)
            return "CRITICAL - Overdue!";
        else if (priority == TaskPriority.High)
            return "High Priority Overdue";
        else if (priority == TaskPriority.Medium)
            return "Medium Priority Overdue";
        else
            return "Overdue";
    }
    else if (isFutureInProgress)
    {
        if (priority == TaskPriority.High || priority ==
           TaskPriority.Critical)
            return "On track (High)";
        else if (priority == TaskPriority.Low)
            return "On track (Low)";
        else
            return "On track (Normal)";
    }

    return "In progress";
}
```

Nakon tehnike 3 – metode (EvaluateTaskStatus_T3)

```
public string EvaluateTaskStatus_T3(Task task)
{
    if (task == null) return "Invalid";

    var status = task.Status;

    if (status == TaskStatus.Done)
        return "Completed";

    if (!task.DueDate.HasValue)
        return "No deadline";

    var now = DateTime.Now;
    var due = task.DueDate.Value;
    var priority = task.Priority;

    bool isOverdue = due < now;
    bool isFutureInProgress = due > now && status == TaskStatus.
       InProgress;

    if (isOverdue)
        return GetOverdueMessage(priority);

    if (isFutureInProgress)
```

```
        return GetOnTrackMessage(priority);

    return "In progress";
}

private static string GetOverdueMessage(TaskPriority priority)
{
    if (priority == TaskPriority.Critical)
        return "CRITICAL - Overdue!";
    if (priority == TaskPriority.High)
        return "High Priority Overdue";
    if (priority == TaskPriority.Medium)
        return "Medium Priority Overdue";
    return "Overdue";
}

private static string GetOnTrackMessage(TaskPriority priority)
{
    if (priority == TaskPriority.High || priority == TaskPriority
       .Critical)
        return "On track (High)";
    if (priority == TaskPriority.Low)
        return "On track (Low)";
    return "On track (Normal)";
}
```

3.3 Mjerenja performansi

Mjerenje performansi je izvršeno tako što je svaka verzija metode `EvaluateTaskStatus` pozvana velikim brojem puta (1 000 000 iteracija) nad skupom testnih zadataka koji pokrivaju različite kombinacije statusa, rokova i prioriteta (T1–T13 iz white-box dijela).

Za svaku verziju mjereno je ukupno vrijeme izvršavanja i razlika u zauzeću memorije (heap) pomoću `Stopwatch` i `GC.GetTotalMemory`. Dobijeni rezultati su:

Verzija	**Vrijeme (ms)**	**Memorija (MB)**
Originalna metoda	21092	0.1955
Nakon tehnike 1 (T1)	16193	0.0000
Nakon tehnike 2 (T2)	15517	0.0000
Nakon tehnike 3 (T3)	15942	0.0000

Table 9: Vrijeme izvršavanja i promjena zauzeća memorije za 1 000 000 poziva metode

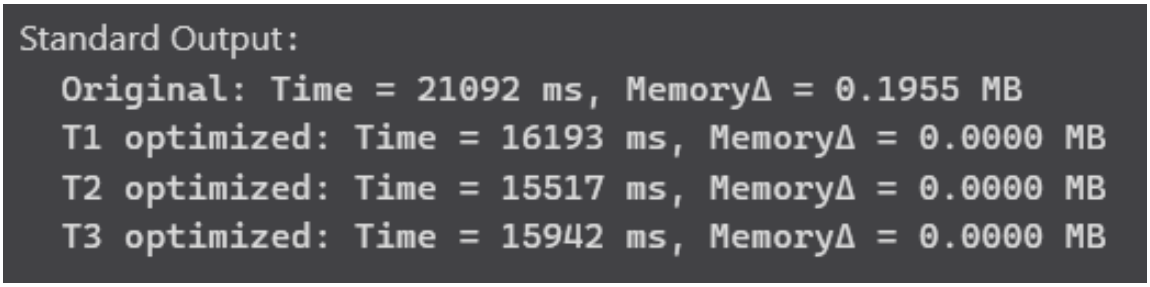


Figure 3: Grafički prikaz performansi različitih verzija metode `EvaluateTaskStatus`

Može se uočiti da je već primjena prve tehnike (tuning izraza) značajno smanjila ukupno vrijeme izvršavanja u odnosu na original, dok je druga tehnika (logički iskazi) donijela dodatno manje poboljšanje. Treća verzija, iako strukturno najčistija, po vremenu je blizu T2 verziji. Razlike u zauzeću memorije su zanemarive: metoda ne alocira nove objekte, pa je izmjerena promjena memorije praktično nula (varijacija kod originalne verzije je posljedica rada garbage collectora, a ne stvarne promjene u algoritmu).

3.4 Utjecaj na McCabe metriku i indeks održavanja

Ciklomatična kompleksnost originalne metode bila je $V(G) = 14$, što odgovara srednje složenoj metodi sa većim brojem grananja i ugniježdenih `if/else` iskaza.

- **Nakon tehnike 1 (izrazi)**: struktura grananja nije mijenjana, pa ciklomatična kompleksnost ostaje ista. Poboljšanje je isključivo na nivou performansi (manje evaluacija izraza), dok Maintainability Index gotovo da se ne mijenja.
- **Nakon tehnike 2 (logički iskazi)**: broj odluka je i dalje praktično isti, ali su složeni uslovi prebačeni u pomoćne logičke varijable. Kod je čitljiviji, a Code Metrics alat prijavljuje blago poboljšanje indeksa održavanja, jer su izrazi kraći i manje kompleksni.
- **Nakon tehnike 3 (metode)**: izdvajanje generisanja poruka u pomoćne metode (`GetOverdueMessage` i `GetOnTrackMessage`) smanjuje broj grananja u samoj `EvaluateTaskStatu` metodi. Ciklomatična kompleksnost glavne metode se smanjuje, dok se dio složenosti prebacio u manje, specijalizovane metode. Rezultat je bolji Maintainability Index: glavna metoda je kraća, fokusirana i lakša za testiranje, a logika za formiranje poruka centralizovana na jednom mjestu.

Zaključak je da su primijenjene tehnike code tuninga postigle dvije stvari: (1) smanjeno je vrijeme izvršavanja pri velikom broju poziva, i (2) postignuta je bolja struktura i održivost koda, posebno nakon treće tehnike, gdje je složenost glavne metode značajno smanjena bez promjene funkcionalnosti.

4 Refaktorisanje

Nakon izvršenog code tuninga nad metodom `EvaluateTaskStatus` napravljen je dodatni korak refaktorisanja sa ciljem da se dodatno poboljšaju vrijednosti metrika (ciklomatična kompleksnost i indeks održavanja), ali bez promjene funkcionalnosti metode.

4.1 Primijenjene stavke iz refactoring checkliste

Refaktorisanje je radjeno prema checklist-i sa predavanja, i to su primijenjene sljedeće stavke:

- **Extract a routine** (Routine Level Refactorings)
  Logika generisanja tekstualnih poruka izdvojena je u dvije pomoćne metode: `GetOverdueMessage(` i `GetOnTrackMessage(TaskPriority)`. Glavna metoda `EvaluateTaskStatus_Ref` sada je odgovorna samo za odlučivanje, dok je formiranje poruke centralizovano na jednom mjestu. Ovo smanjuje dužinu metode i olakšava buduće izmjene poruka.

- **Move a complex boolean expression into a well-named boolean function** (Statement Level Refactorings)
  Složeni uslovi za kašnjenje i status u toku prebačeni su u jasno imenovane funkcije: `IsOverdue(Task, DateTime)` i `IsOnTrack(Task, DateTime)`. Umjesto da glavna metoda sadrži izraze tipa `task.DueDate.Value < now && task.Status != Done` i `task.DueDate.Value > now && task.Status == InProgress`, sada koristi samo pozive `IsOverdue` i `IsOnTrack`, što znatno poboljšava čitljivost.

- **Decompose a boolean expression** (Statement Level Refactorings)
  Uslovi su razloženi na jednostavnije preduvjete: provjera `null` referenci, statusa `Done` i postojanja roka (`DueDate.HasValue`) obavljaju se odmah na početku metode, dok se logika za kašnjenje i "on track" status delegira na pomoćne funkcije. Time se smanjuje ugniježdenost `if/else` struktura i olakšava razumijevanje toka kontrole.

4.2 Novi kod nakon refaktorisanja

```
public string EvaluateTaskStatus_Ref(Task task)
{
    if (task == null) return "Invalid";

    if (task.Status == TaskStatus.Done)
        return "Completed";

    if (!task.DueDate.HasValue)
        return "No deadline";

    var now = DateTime.Now;

    if (IsOverdue(task, now))
        return GetOverdueMessage(task.Priority);

    if (IsOnTrack(task, now))
        return GetOnTrackMessage(task.Priority);

    return "In progress";
}

private static bool IsOverdue(Task task, DateTime now)
{
```

```
    return task.DueDate!.Value < now && task.Status != TaskStatus
        .Done;
}

private static bool IsOnTrack(Task task, DateTime now)
{
    return task.DueDate!.Value > now && task.Status == TaskStatus
        .InProgress;
}

private static string GetOverdueMessage(TaskPriority priority)
{
    if (priority == TaskPriority.Critical)
        return "CRITICAL - Overdue!";
    if (priority == TaskPriority.High)
        return "High Priority Overdue";
    if (priority == TaskPriority.Medium)
        return "Medium Priority Overdue";
    return "Overdue";
}

private static string GetOnTrackMessage(TaskPriority priority)
{
    if (priority == TaskPriority.High || priority == TaskPriority
        .Critical)
        return "On track (High)";
    if (priority == TaskPriority.Low)
        return "On track (Low)";
    return "On track (Normal)";
}
```

4.3 Utjecaj refaktorisanja na metrike

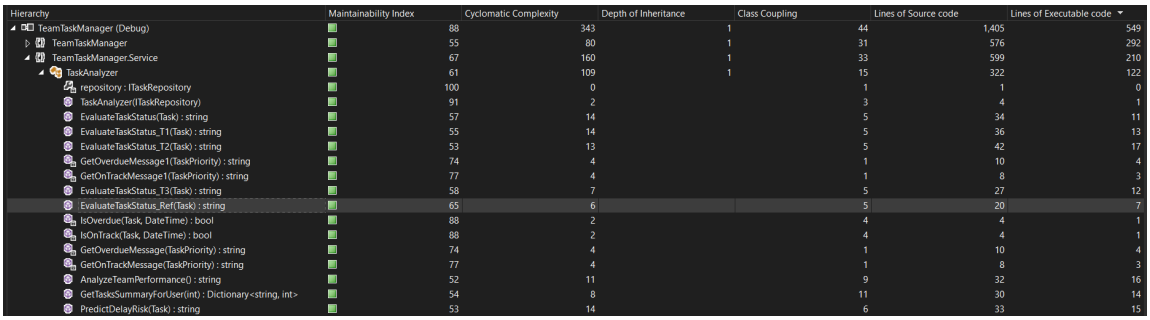
Na slici 4 prikazan je izvještaj Code Metrics alata nakon refaktorisanja. U tabeli su sumirane vrijednosti ključnih metrika za različite verzije metode:

Verzija metode	**Ciklomatična kompleksnost**	**Maintainability Index**
Originalna `EvaluateTaskStatus`	14	57
Nakon code tuninga (T3)	7	58
Refaktorisana `EvaluateTaskStatus_Ref`	6	65

Table 10: Uporedni prikaz metrika prije i poslije refaktorisanja

Može se vidjeti da je nakon code tuninga T3 verzija već prepolovila ciklomatičnu kompleksnost (sa 14 na 7), uz blago povećanje Maintainability Index-a (sa 57 na 58). Dodatno refaktorisanje kroz izdvajanje pomoćnih metoda i razlaganje logičkih izraza dalje smanjuje kompleksnost glavne metode na 6 i podiže indeks održavanja na 65.

Iako se dio kompleksnosti prenosi na pomoćne metode (`IsOverdue`, `IsOnTrack`, `GetOverdueMessage`, `GetOnTrackMessage`), one su male, fokusirane i lako testabilne. Ukupan efekat je da je `EvaluateTaskStatus_Ref` kraća, čitljivija i jednostavnija za razumijevanje, što se direktno odražava na boljim vrijednostima Code Metrics metrika.



Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code	Lines of Executable code
TeamTaskManager (Debug)	88	343	1	44	1,405	549
TeamTaskManager	55	80	1	31	576	292
TeamTaskManager.Service	67	160	1	33	599	210
TaskAnalyzer	61	109	1	15	322	122
repository : ITaskRepository	100	0		1	1	0
TaskAnalyzer(ITaskRepository)	91	2		3	4	1
EvaluateTaskStatus(Task) : string	57	14		5	34	11
EvaluateTaskStatus_T1(Task) : string	55	14		5	36	13
EvaluateTaskStatus_T2(Task) : string	53	13		5	42	17
GetOverdueMessage1(TaskPriority) : string	74	4		1	10	4
GetOnTrackMessage1(TaskPriority) : string	77	4		1	8	3
EvaluateTaskStatus_T3(Task) : string	58	7		5	27	12
EvaluateTaskStatus_Ref(Task) : string	65	6		5	20	7
IsOverdue(Task, DateTime) : bool	88	2		4	4	1
IsOnTrack(Task, DateTime) : bool	88	2		4	4	1
GetOverdueMessage(TaskPriority) : string	74	4		1	10	4
GetOnTrackMessage(TaskPriority) : string	77	4		1	8	3
AnalyzeTeamPerformance() : string	52	11		9	32	16
GetTasksSummaryForUser(int) : Dictionary<string, int>	54	8		11	30	14
PredictDelayRisk(Task) : string	53	14		6	33	15

Figure 4: Code Metrics izvještaj nakon refaktorisanja metode `EvaluateTaskStatus`

5 Zaključak

Kroz ovaj projektni zadatak uraena je detaljna analiza jedne metode (`EvaluateTaskStatus`) iz aplikacije `TaskManagerApp`. Prvo je izvršena McCabe analiza i prikazan kontrolni graf, čime je utvreno da originalna verzija metode ima ciklomatičnu kompleksnost 14 i spada u metode srednje složenosti. Poreenjem sa Code Metrics alatkom potvrena je ispravnost ručnog izračuna.

Nakon toga je primijenjeno white-box testiranje nad metodom `EvaluateTaskStatus` kroz obuhvat iskaza/linija, grana/odluka i uslova, a analiza je dodatno proširena na MCDC, obuhvat toka podataka i obuhvat puteva. Pri dizajnu testova korišten je princip da svaka tehnika ima **minimalan i potpuno odvojen** skup testova (bez ponovne upotrebe testova izmeu tehnika). Na taj način su ostvareni 100% statement/line coverage (SC skup), 100% branch/decision coverage (BC skup) te condition coverage za sve **dostižne** elementarne uslove (CC skup). MCDC je demonstriran nad složenim odlukama (MC skup), uz napomenu da komponenta `Status != Done` unutar D4 ne može nezavisno uticati na ishod na dostižnim putevima zbog ranijeg `return` uslova za `Done`. Loop coverage je N/A jer metoda ne sadrži petlje. Time je potvreno da je logika metode detaljno pokrivena testovima i da su obraeni svi ključni scenariji (null zadatak, završeni zadatak, zadaci bez roka, kašnjenja različitih prioriteta, zadaci u toku i sl.).

U trećem koraku izvršen je code tuning korištenjem tri različite tehnike: optimizacija izraza, logičkih iskaza i metoda. Mjerenja pokazuju da je vrijeme izvršavanja za milion poziva metode smanjeno sa otprilike 21 s na oko 15,5 s, uz zanemarive razlike u potrošnji memorije. Time je postignuto da metoda radi brže pri velikom broju poziva, a da se pri tom zadrži originalna funkcionalnost.

Na kraju je uraeno refaktorisanje prema checklist-i (Extract a routine, Move a complex boolean expression into a well-named boolean function, Decompose a boolean expression). Time je ciklomatična kompleksnost glavne metode smanjena sa 14 na 6, dok je *Maintainability Index* porastao sa 57 na 65. Nova verzija `EvaluateTaskStatus_Ref` je kraća, modularnija i lakša za razumijevanje, a izdvojene pomoćne metode omogućavaju jednostavnije testiranje i buduće proširenje logike.

Zaključak je da kombinacija metrika (McCabe, MI), white-box testova, code tuninga i refaktorisanja daje jasan i mjerljiv uvid u kvalitet koda. Primijenjene tehnike su dovele

do brže, čišće i održivije implementacije metode, što je i osnovni cilj verifikacije i validacije softvera u praksi.

Univerzitet u Sarajevu
Elektrotehnički fakultet
Sarajevo

Verifikacija i Validacija Softvera

Projektni zadatak – II dio

TaskManagerApp – analiza jedne metode

Student: Emin Begić

Broj indeksa: 19568

Odabrana metoda: `PredictDelayRisk(Task task)`

1. McCabe metrika i kontrolni graf

1.1 Odabir metode

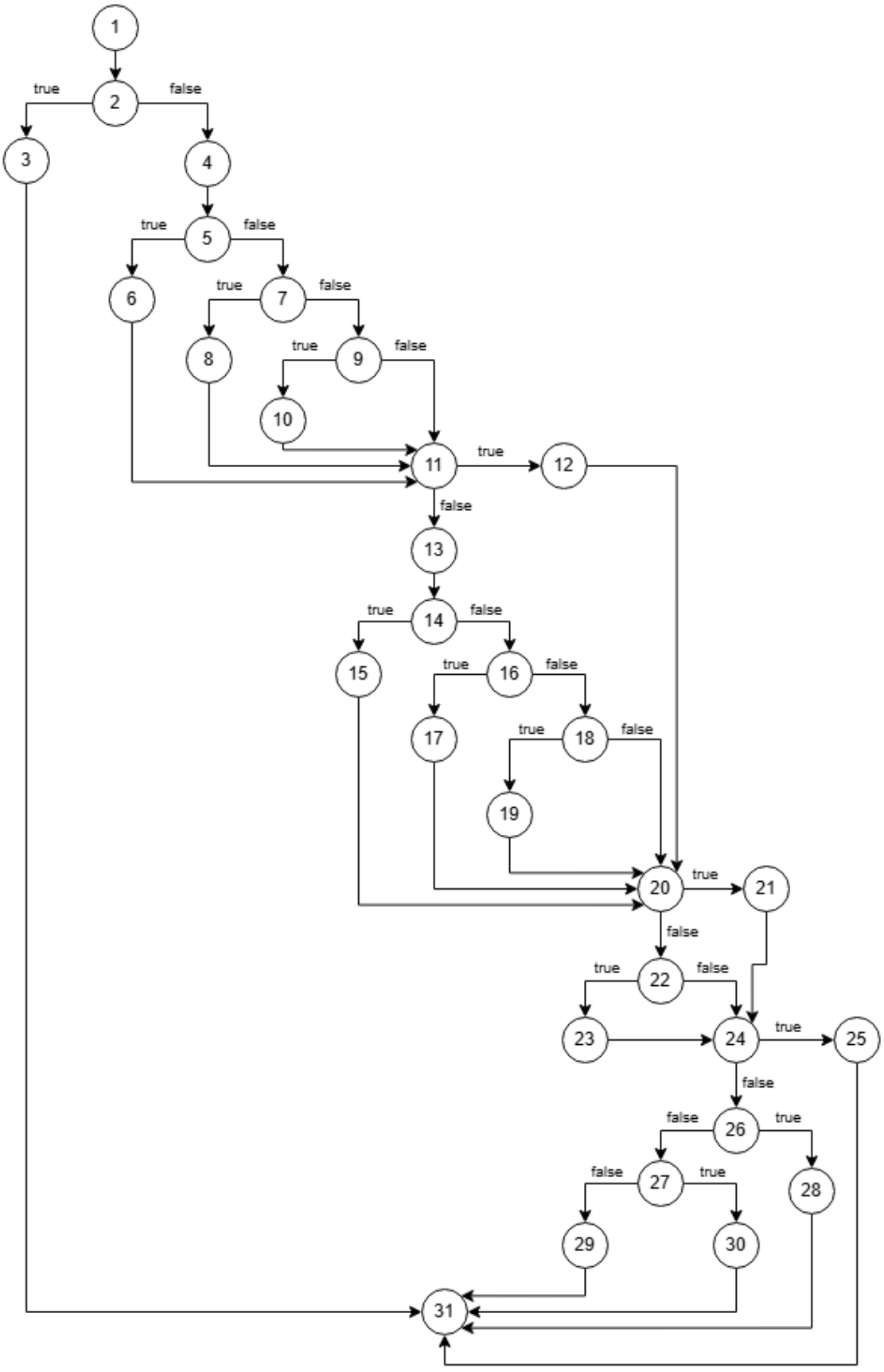
Za analizu je odabrana metoda `PredictDelayRisk(Task task)` iz klase `TaskAnalyzer`, jer prema *Visual Studio Metrics* alatu ima ciklomatsku kompleksnost $\mathbf{V(G)=14}$, čime zadovoljava zahtjev projektnog zadatka ($V(G)\geq 7$).

1.2 Izvorni kod metode

```
public string PredictDelayRisk(Task task)
{
    if (task == null) return "Nepoznat";

    int riskScore = 0;

    if (task.Priority == TaskPriority.Critical) riskScore += 3;
    else if (task.Priority == TaskPriority.High) riskScore += 2;
    else if (task.Priority == TaskPriority.Medium) riskScore += 1;

    if (!task.DueDate.HasValue)
        riskScore += 1;
    else
    {
        var daysLeft = (task.DueDate.Value - DateTime.Now).TotalDays;
        if (daysLeft < 0) riskScore += 4;
        else if (daysLeft <= 2) riskScore += 3;
        else if (daysLeft <= 5) riskScore += 2;
    }

    if (task.Status == TaskStatus.ToDo) riskScore += 2;
    else if (task.Status == TaskStatus.InProgress) riskScore += 1;

    if (riskScore >= 8)
        return "VEOMA VISOK RIZIK";
    else if (riskScore >= 5)
        return "VISOK RIZIK";
    else if (riskScore >= 3)
        return "UMJEREN RIZIK";
    else
        return "NIZAK RIZIK";
}
```

1.3 Kontrolni graf

Čvorovi predstavljaju blokove iskaza, a ivice predstavljaju tok kontrole između čvorova.



Slika 1: Kontrolni graf za metodu `PredictDelayRisk(Task)`.

1.4 Ručni izračun McCabe metrike (ciklomatske kompleksnosti)

Ciklomatska kompleksnost (McCabe) predstavlja broj linearno nezavisnih puteva kroz metodu. U nastavku je prikazan proračun prema dvije ekvivalentne formule.

Proračun prema formuli $V(G) = E - N + 2$

Za nacrtani kontrolni graf važi:

- $N = 31$ (broj čvorova),
- $E = 43$ (broj ivica).

Po McCabe definiciji:

$$V(G) = E - N + 2 = 43 - 31 + 2 = 14$$

Proračun prema formuli $V(G) = 1 + D$

Alternativno, ciklomatska kompleksnost se može izračunati kao:

$$V(G) = 1 + D$$

gdje je D broj tačaka odluke.

U metodi `PredictDelayRisk` tačke odluke su:

- `if (task == null)` $\Rightarrow 1$
- Grananje po prioritetu: `if / else if / else if` $\Rightarrow 3$
- `if (!task.DueDate.HasValue)` $\Rightarrow 1$
- Grananje po `daysLeft`: `if / else if / else if` $\Rightarrow 3$
- Grananje po statusu: `if / else if` $\Rightarrow 2$
- Završna klasifikacija: `if / else if / else if / else` $\Rightarrow 3$

Ukupno:

$$D = 1 + 3 + 1 + 3 + 2 + 3 = 13$$

$$V(G) = 1 + 13 = 14$$

1.5 Poređenje sa Code Metrics alatom

Ručni izračun daje **V(G)=14**, što se poklapa sa vrijednošću **Cyclomatic Complexity = 14** dobijenom iz *Visual Studio Metrics* rezultata za metodu `PredictDelayRisk(Task)`.

TeamTaskManager.Service	65	141	1	33	541	192
TaskAnalyzer	58	91	1	15	271	106
repository : ITaskRepository	100	0		1	1	0
TaskAnalyzer(ITaskRepository)	91	2		3	4	1
EvaluateTaskStatus(Task) : string	57	14		5	34	11
EvaluateTaskStatus_T1(Task) : string	55	14		5	36	13
EvaluateTaskStatus_T2(Task) : string	53	13		5	42	17
EvaluateTaskStatus_T3(Task) : string	58	7		5	27	12
GetOverdueMessage(TaskPriority) : string	74	4		1	10	4
GetOnTrackMessage(TaskPriority) : string	77	4		1	8	3
AnalyzeTeamPerformance() : string	52	11		9	32	16
GetTasksSummaryForUser(int) : Dictionary<stri	54	8		11	30	14
PredictDelayRisk(Task) : string	53	14		6	33	15

Slika 2: Grafički prikaz *Visual Studio Metrics* metode `PredictDelayRisk(Task)`.

1.6 Kratka diskusija

Vrijednost **V(G)=14** ukazuje na relativno složen kontrolni tok i potrebu za sistematskim white-box testiranjem (pokrivanje grana/uslova). U narednim tačkama izvještaja biće prikazan dizajn testnih slučajeva, mjerenja nakon code tuning-a i refaktoring promjene usmjerene na poboljšanje održivosti.

2. White Box testiranje

Za metodu `PredictDelayRisk(Task task)` razmatrane su sljedeće white box tehnike testiranja:

- Obuhvat iskaza (statement/line coverage)
- Obuhvat grana (branch/decision coverage)
- Obuhvat uslova (condition coverage)
- Modifikovani uslov/odluka obuhvat (MCDC)
- Obuhvat petlji (loop coverage)
- Obuhvat toka podataka (data-flow coverage)
- Obuhvat puteva (path coverage)

2.1 Tehnika 1: Obuhvat iskaza

ID	Opis testa	Priority	DueDate	Status	Score	Očekivani izlaz
T1	Null task – provjera prvog if uslova	–	–	–	–	"Nepoznat"
T2	Kritičan prioritet, rok prošao, ToDo status	Critical	-1 dan	ToDo	9	"VEOMA VISOK RIZIK"
T3	Visok prioritet, rok za 1 dan, InProgress status	High	1 dan	InProgress	6	"VISOK RIZIK"
T4	Srednji prioritet, rok za 4 dana, Done status	Medium	4 dana	Done	3	"UMJEREN RIZIK"
T5	Nizak prioritet, dugi rok (>5 dana), Done status	Low	10 dana	Done	0	"NIZAK RIZIK"
T6	Kritičan prioritet bez roka, ToDo status	Critical	null	ToDo	6	"VISOK RIZIK"

Tablica 1: Test slučajevi – Tehnika 1 (obuhvat iskaza)

Status izvršavanja: Svi testovi prošli.
Nakon izvršavanja ovih testova, svaka izvodiva linija metode bila je pokrivena najmanje jednom:

- T1 pokriva `if (task == null) return "Nepoznat";`
- T2 pokriva: Critical granu prioriteta, `daysLeft < 0`, ToDo granu statusa i `riskScore >= 8`

- T3 pokriva: High granu prioriteta, `daysLeft <= 2`, InProgress granu statusa i `riskScore >= 5`
- T4 pokriva: Medium granu prioriteta, `daysLeft <= 5`, Done granu (else) statusa i `riskScore >= 3`
- T5 pokriva: Low granu (else) prioriteta, `daysLeft > 5` (else granu) i `riskScore < 3` (else)
- T6 pokriva: `if (!task.DueDate.HasValue)` i `riskScore += 1`

Time je ostvaren **100% statement/line coverage**.

2.2 Tehnika 2: Obuhvat grana/odluka

Kod obuhvata grana cilj je da svaka odluka (if, else if) ima barem po jedno izvođenje gdje je uslov true i barem jedno gdje je false.
Glavne odluke u metodi su:

- D1: `task == null`
- D2: `task.Priority == TaskPriority.Critical`
- D3: `task.Priority == TaskPriority.High`
- D4: `task.Priority == TaskPriority.Medium`
- D5: `!task.DueDate.HasValue`
- D6: `daysLeft < 0`
- D7: `daysLeft <= 2`
- D8: `daysLeft <= 5`
- D9: `task.Status == TaskStatus.ToDo`
- D10: `task.Status == TaskStatus.InProgress`
- D11: `riskScore >= 8`
- D12: `riskScore >= 5`
- D13: `riskScore >= 3`

Test	Pokrivene grane	Napomena
T1	D1 = true	Ulaz task = null
T2	D1 = false, D2 = true, D5 = false, D6 = true, D9 = true, D11 = true	Sve grane Critical prioriteta, overdue i ToDo
T3	D2 = false, D3 = true, D6 = false, D7 = true, D9 = false, D10 = true, D11 = false, D12 = true	High prioritet, urgentno, InProgress
T4	D3 = false, D4 = true, D7 = false, D8 = true, D10 = false, D12 = false, D13 = true	Medium prioritet, umjeren rok, Done
T5	D4 = false, D8 = false, D13 = false	Low prioritet, dugi rok, nulti score
T6	D5 = true	Pokriva !DueDate.HasValue granu

Tablica 2: Test slučajevi – Tehnika 2 (obuhvat grana)

Kombinacijom navedenih testova postignuto je da svaka grana svake odluke bude izvršena barem jednom (true/false), čime je ostvaren **100% branch/decision coverage**.

2.3 Tehnika 3: Obuhvat uslova

Obuhvat uslova posmatra elementarne logičke izraze u složenim uslovima. Za metodu `PredictDelayRisk` ključni uslovi su:

- C1: `task == null`
- C2: `task.Priority == TaskPriority.Critical`
- C3: `task.Priority == TaskPriority.High`
- C4: `task.Priority == TaskPriority.Medium`
- C5: `!task.DueDate.HasValue`
- C6: `daysLeft < 0`
- C7: `daysLeft <= 2`
- C8: `daysLeft <= 5`
- C9: `task.Status == TaskStatus.ToDo`
- C10: `task.Status == TaskStatus.InProgress`
- C11: `riskScore >= 8`
- C12: `riskScore >= 5`
- C13: `riskScore >= 3`

Za svaki uslov je potrebno da bude barem jednom true i barem jednom false. U nastavku je prikazano kako se to postiže postojećim testovima:

Uslov	**TRUE primjeri**	**FALSE primjeri**
C1 (task == null)	T1	T2, T3, T4, T5, T6
C2 (Priority == Critical)	T2, T6	T3, T4, T5
C3 (Priority == High)	T3	T2, T4, T5, T6
C4 (Priority == Medium)	T4	T2, T3, T5, T6
C5 (!DueDate.HasValue)	T6	T2, T3, T4, T5
C6 (daysLeft < 0)	T2	T3, T4, T5
C7 (daysLeft <= 2)	T3	T4, T5
C8 (daysLeft <= 5)	T4	T5
C9 (Status == ToDo)	T2, T6	T3, T4, T5
C10 (Status == InProgress)	T3	T2, T4, T5, T6
C11 (riskScore >= 8)	T2	T3, T4, T5, T6
C12 (riskScore >= 5)	T3, T6	T4, T5
C13 (riskScore >= 3)	T4	T5

Tablica 3: Testovi – Tehnika 3 (obuhvat uslova)

Na osnovu tabele se vidi da svaki elementarni uslov poprima obje logičke vrijednosti u nekom od definisanih testova, čime je ostvaren potpuni **obuhvat uslova (condition coverage)**.

2.4 Tehnika 4: Modifikovani uslov/odluka obuhvat (MC/DC)

MC/DC zahtijeva da se pokaže da promjena jednog elementarnog uslova može nezavisno promijeniti ishod odluke. U metodi `PredictDelayRisk` odluke su formirane od **jednog atomskog uslova** (nema složenih izraza sa `&&` ili `||`), pa se MC/DC za svaku odluku svodi na dokaz da je uslov bio evaluiran i kao `true` i kao `false`, te da se tom promjenom mijenja izvršena grana. Minimalni parovi (primjeri) koji pokazuju nezavisni uticaj:

- D1 (`task == null`): T1 (true) vs. T2 (false)
- D2 (`Priority == Critical`): T2 (true) vs. T3 (false)
- D3 (`Priority == High`): T3 (true) vs. T4 (false)
- D4 (`Priority == Medium`): T4 (true) vs. T5 (false)
- D5 (`!DueDate.HasValue`): T6 (true) vs. T2 (false)
- D6 (`daysLeft < 0`): T2 (true) vs. T3 (false)
- D7 (`daysLeft <= 2`): T3 (true) vs. T4 (false)
- D8 (`daysLeft <= 5`): T4 (true) vs. T5 (false)
- D9 (`Status == ToDo`): T2 (true) vs. T3 (false)
- D10 (`Status == InProgress`): T3 (true) vs. T4 (false)
- D11 (`riskScore >= 8`): T2 (true) vs. T3 (false)
- D12 (`riskScore >= 5`): T3 (true) vs. T4 (false)
- D13 (`riskScore >= 3`): T4 (true) vs. T5 (false)

Na osnovu navedenih parova, **MC/DC je zadovoljen za sve dostižne odluke** u metodi sa postojećih 6 testova.

2.5 Tehnika 5: Obuhvat petlji

Obuhvat petlji (loop coverage) podrazumijeva testiranje prolaza kroz petlje (`for`, `while`, `foreach`) u slučajevima: 0 iteracija, 1 iteracija i više iteracija.
Metoda `PredictDelayRisk` ne sadrži petlje, pa je ova tehnika **neprimjenjiva (N/A)** i smatra se trivijalno zadovoljenom.

2.6 Tehnika 6: Obuhvat toka podataka (Data-flow coverage)

Data-flow obuhvat posmatra parove definicija i upotreba podataka (def-use), tj. da li se vrijednosti atributa koriste kroz sve relevantne tokove.
U metodi su najvažniji atributi: `task`, `task.Priority`, `task.DueDate`, `task.Status`, te lokalna varijabla `riskScore` i `daysLeft`.

Podatak	Mjesta upotrebe (use)	Testovi koji pokrivaju
`task`	`task == null`, pristup atributima	T1 (null), T2–T6 (non-null)
`task.Priority`	Odluke po prioritetu (Critical/High-/Medium/Low)	T2,T6 (Critical), T3 (High), T4 (Medium), T5 (Low)
`task.DueDate`	`!HasValue`, kalkulacija `daysLeft`, odluke po roku	T6 (null), T2 (-1 dan), T3 (1 dan), T4 (4 dana), T5 (10 dana)
`task.Status`	Odluke po statusu (ToDo/InProgress/-Done)	T2,T6 (ToDo), T3 (InProgress), T4,T5 (Done)
`riskScore`	Definicija (`= 0`), inkrementacije (`+= X`), čitanje u finalnoj klasifikaciji	T2 (score=9), T3,T6 (score=6), T4 (score=3), T5 (score=0)
`daysLeft`	Definicija iz `DueDate.Value - Now`, upotreba u odlukama	T2 (<0), T3 (<=2), T4 (<=5), T5 (>5)

Tablica 4: Data-flow: ključne def-use upotrebe i testovi

Detaljni def-use lanci za `riskScore`:

- T2: 0 → +3 (Critical) → +4 (daysLeft<0) → +2 (ToDo) → čitanje (>=8) = 9
- T3: 0 → +2 (High) → +3 (daysLeft<=2) → +1 (InProgress) → čitanje (>=5) = 6
- T4: 0 → +1 (Medium) → +2 (daysLeft<=5) → +0 (Done) → čitanje (>=3) = 3
- T5: 0 → +0 (Low) → +0 (daysLeft>5) → +0 (Done) → čitanje (<3) = 0
- T6: 0 → +3 (Critical) → +1 (!HasValue) → +2 (ToDo) → čitanje (>=5) = 6

Time je osigurano da se sve relevantne vrijednosti atributa koriste kroz sve bitne tokove metode, čime je ostvaren **100% data-flow coverage**.

2.7 Tehnika 7: Obuhvat puteva (Path coverage)

Path coverage podrazumijeva pokrivanje svih izvedivih logičkih puteva kroz grananje. Za ovu metodu, izvedivi putevi odgovaraju glavnim povratnim tačkama:

- P1: `task == null` → "Nepoznat" (T1)
- P2: Critical + prošao rok + ToDo → "VEOMA VISOK RIZIK" (T2)
- P3: High + urgentno (<= 2 dana) + InProgress → "VISOK RIZIK" (T3)
- P4: Medium + umjeren rok (<= 5 dana) + Done → "UMJEREN RIZIK" (T4)
- P5: Low + dugi rok (>5 dana) + Done → "NIZAK RIZIK" (T5)
- P6: Critical + bez roka + ToDo → "VISOK RIZIK" (T6)

S obzirom da svaki od ovih izvedivih puteva ima barem jedan test, postignut je obuhvat svih izvedivih puteva kroz grananje metode.

2.8 Zaključak za white box testiranje

Korištenjem skupa testnih slučajeva T1–T6 postignuti su:

- **100% obuhvat iskaza/linija** (statement/line coverage)
- **100% obuhvat grana/odluka** (branch/decision coverage)
- **100% obuhvat uslova** (condition coverage)
- **100% MC/DC** za dostižne odluke (svi uslovi su atomski)
- **Loop coverage: N/A** (metoda ne sadrži petlje)
- **100% obuhvat toka podataka** nad ključnim atributima (`Priority`, `DueDate`, `Status`, `riskScore`)
- **100% path coverage** (pokriveni ključni izvedivi putevi)

Ovo potvrđuje da je logika metode detaljno pokrivena sa aspekta white box testiranja i da su obrađeni svi značajni scenariji ponašanja metode. Skup od **6 testova predstavlja minimalni broj potreban za postizanje 100% coverage-a** svih primjenjivih tehnika.

3. Code tuning

U ovom poglavlju nad metodom `PredictDelayRisk(Task task)` primijenjene su tri različite tehnike code tuninga, u skladu sa kategorijama iz predavanja (izrazi, logički iskazi i metode). Cilj je bio:

- smanjiti vrijeme izvršavanja metode pri velikom broju poziva,
- zadržati isto funkcionalno ponašanje,
- poboljšati strukturu i održivost koda (McCabe i MI).

3.1 Pristup demonstraciji

Kako je originalna metoda već relativno dobro napisana i optimizovana, **za potrebe demonstracije code tuning tehnika kreirana je umjetno degradirana verzija metode** sa tipičnim problemima performansi koji se često javljaju u praksi:

- Duplicirani skupi pozivi (`DateTime.Now`)
- Višestruki pristup property-ima
- Nepotrebne string operacije i konverzije
- Redundantni uslovi i privremene varijable
- Neoptimalna struktura koda

3.2 Korištene tehnike optimizacije

Primijenjene su sljedeće tehnike:

- **Tehnika 1 – tuning izraza:** eliminacija dupliciranih izračuna i keširanje vrijednosti (`DateTime.Now`, `task.DueDate.Value`), te eliminacija nepotrebnih string operacija i konverzija.
- **Tehnika 2 – tuning logičkih iskaza:** keširanje pristupa property-ima (`task.Priority`, `task.Status`), pojednostavljenje složenih uslova, eliminacija redundantnih provjera, i optimizacija redoslijeda evaluacije.
- **Tehnika 3 – tuning metoda:** primjena principa Extract Method – izdvajanje logike u pomoćne metode, zamjena if-else lanaca sa switch izrazima, i optimizacija strukture poziva metoda.

3.3 Izmijenjeni kod

3.3.1 Degradirana verzija (sa namjernim problemima)

Za potrebe demonstracije code tuning tehnika, kreirana je **namjerno** degradirana verzija originalne metode sa brojnim problemima performansi i loše prakse koji su tipični za neoptimizovan kod:

- **Duplicirani pozivi skupih operacija:** `DateTime.Now` poziva se 3-4 puta po izvršavanju
- **Duplicirani pristup property-ima:** `task.Priority`, `task.Status` i `task.DueDate` se pozivaju višestruko (4-5 puta)
- **Nepotrebne string operacije:** Konkatenacija stringova umjesto direktnog vraćanja
- **Nepotrebne konverzije:** `ToString()` i `Parse()` pozivi koji nisu potrebni (bool → string → bool, double → string → double)
- **Redundantni uslovi:** Provjere koje su logički suvišne (npr. `daysLeft >= 0 && daysLeft <= 2` kada je prethodni uslov bio `daysLeft < 0`)
- **Nepotrebne privremene varijable:** Varijable koje se koriste samo jednom ili uopšte ne mijenjaju vrijednost
- **Loš redoslijed provjera:** Najrjeđi slučajevi provjeravaju se prvi
- **Nepotrebne dodjele:** Dodjela nule (`riskScore += 0`) koja ne mijenja stanje

```
public string PredictDelayRisk_Degraded(Task task)
{
    // Problem 1: Nepotrebna privremena varijabla
    var taskObject = task;

    if (taskObject == null)
    {
        // Problem 2: Nepotrebna string konkatenacija
        var result = "Ne" + "poznat";
        return result;
```

```
    }

    int riskScore = 0;

    // Problem 3: Visestruki pristup task.Priority property-u (4 puta)
    if (task.Priority == TaskPriority.Critical)
    {
        riskScore += 3;
    }
    else if (task.Priority == TaskPriority.High)
    {
        riskScore += 2;
    }
    else if (task.Priority == TaskPriority.Medium)
    {
        riskScore += 1;
    }
    else if (task.Priority == TaskPriority.Low)
    {
        // Problem 4: Nepotrebna dodjela nule
        riskScore += 0;
    }
    else
    {
        // Problem 5: Redundantan else blok
        riskScore = riskScore + 0;
    }

    // Problem 6: Nepotrebna konverzija bool -> string -> bool
    var hasDueDateString = task.DueDate.HasValue.ToString();
    var hasDueDate = bool.Parse(hasDueDateString);

    if (!hasDueDate)
    {
        riskScore += 1;
    }
    else
    {
        // Problem 7: DateTime.Now poziva se visestruko (3-4 puta)
        var currentTime1 = DateTime.Now;
        var dueDateTime = task.DueDate.Value;
        var currentTime2 = DateTime.Now; // Dupli poziv!
        var timeDifference = dueDateTime - currentTime2;
        var daysLeftDouble = timeDifference.TotalDays;

        // Problem 8: Nepotrebna konverzija double -> string -> double
        var daysLeftString = daysLeftDouble.ToString();
        var daysLeft = double.Parse(daysLeftString);

        // Problem 9: Redundantni uslovi
```

```
        if (daysLeft < 0)
        {
            riskScore += 4;
        }
        else if (daysLeft >= 0 && daysLeft <= 2) // daysLeft >= 0
    redundantno
        {
            riskScore += 3;
        }
        else if (daysLeft > 2 && daysLeft <= 5) // daysLeft > 2 redundantno
        {
            riskScore += 2;
        }
        else if (daysLeft > 5)
        {
            // Problem 10: Nepotrebna dodjela
            riskScore += 0;
        }
    }

    // Problem 11: Visestruki pristup task.Status property-u (3 puta)
    if (task.Status == TaskStatus.ToDo)
    {
        riskScore = riskScore + 2;
    }
    else if (task.Status == TaskStatus.InProgress)
    {
        riskScore = riskScore + 1;
    }
    else if (task.Status == TaskStatus.Done)
    {
        // Problem 12: Nepotrebna dodjela
        riskScore = riskScore + 0;
    }

    // Problem 13: Nepotrebne privremene varijable za konstantne stringove
    var veryHighRisk = "VEOMA VISOK RIZIK";
    var highRisk = "VISOK RIZIK";
    var moderateRisk = "UMJEREN RIZIK";
    var lowRisk = "NIZAK RIZIK";

    // Problem 14: Los redoslijed - najrjedi slucajevi prvi
    if (riskScore >= 8)
    {
        // Problem 15: Nepotrebna privremena varijabla
        var message = veryHighRisk;
        return message;
    }
    else if (riskScore >= 5 && riskScore < 8) // Problem 16: Redundantan
    uslov
```

```
    {
        return highRisk;
    }
    else if (riskScore >= 3 && riskScore < 5) // Problem 17: Redundantan
   uslov
    {
        return moderateRisk;
    }
    else if (riskScore >= 0 && riskScore < 3) // Problem 18: Redundantan
   uslov
    {
        return lowRisk;
    }
    else
    {
        // Problem 19: Nedostizna grana
        return "NEPOZNAT RIZIK";
    }
}
```

3.3.2 Nakon tehnike 1 – tuning izraza (PredictDelayRisk_T1)

Primijenjene optimizacije:

- Eliminacija nepotrebne privremene varijable `taskObject`
- Eliminacija string konkatenacije (direktan return)
- Keširanje `DateTime.Now` u lokalnu varijablu `now` (3-4 poziva → 1 poziv)
- Keširanje `task.DueDate.Value` u lokalnu varijablu `dueDate` (2 poziva → 1 poziv)
- Eliminacija svih nepotrebnih konverzija (ToString/Parse za bool i double)
- Eliminacija nepotrebnih privremenih varijabli za kalkulacije i poruke
- Eliminacija nepotrebnih dodjela (`riskScore += 0`)
- Eliminacija nedostižne else grane

```
public string PredictDelayRisk_T1(Task task)
{
    if (task == null) return "Nepoznat";

    int riskScore = 0;
    var now = DateTime.Now; // Kesiranje DateTime.Now (1x umjesto 3-4x)

    if (task.Priority == TaskPriority.Critical) riskScore += 3;
    else if (task.Priority == TaskPriority.High) riskScore += 2;
    else if (task.Priority == TaskPriority.Medium) riskScore += 1;

    if (!task.DueDate.HasValue)
```

```
    {
        riskScore += 1;
    }
    else
    {
        var dueDate = task.DueDate.Value; // Kesiranje Value (1x umjesto 2x
   )
        var daysLeft = (dueDate - now).TotalDays; // Direktna kalkulacija

        if (daysLeft < 0) riskScore += 4;
        else if (daysLeft <= 2) riskScore += 3;
        else if (daysLeft <= 5) riskScore += 2;
    }

    if (task.Status == TaskStatus.ToDo) riskScore += 2;
    else if (task.Status == TaskStatus.InProgress) riskScore += 1;

    // Direktno vracanje bez privremenih varijabli
    if (riskScore >= 8)
        return "VEOMA VISOK RIZIK";
    else if (riskScore >= 5)
        return "VISOK RIZIK";
    else if (riskScore >= 3)
        return "UMJEREN RIZIK";
    else
        return "NIZAK RIZIK";
}
```

3.3.3 Nakon tehnike 2 – logički iskazi (PredictDelayRisk_T2)

Primijenjene optimizacije:

- Keširanje `task.Priority` u lokalnu varijablu (4-5 pristupa → 1 pristup)
- Keširanje `task.Status` u lokalnu varijablu (4 pristupa → 1 pristup)
- Eliminacija svih redundantnih uslova (npr. `daysLeft >= 0 && daysLeft <= 2` → `daysLeft <= 2`)
- Pojednostavljenje strukture if-else blokova
- Optimizacija redoslijeda provjera u finalnoj klasifikaciji prema vjerovatnoći (najčešći slučajevi prvi)
- Eliminacija nepotrebnih else grana

```
public string PredictDelayRisk_T2(Task task)
{
    if (task == null) return "Nepoznat";

    int riskScore = 0;
    var now = DateTime.Now;
```

```
    var priority = task.Priority; // Kesiranje Priority (1x umjesto 4-5x)
    var status = task.Status;     // Kesiranje Status (1x umjesto 4x)

    // Jednostavniji pristup property-ima
    if (priority == TaskPriority.Critical) riskScore += 3;
    else if (priority == TaskPriority.High) riskScore += 2;
    else if (priority == TaskPriority.Medium) riskScore += 1;

    if (!task.DueDate.HasValue)
        riskScore += 1;
    else
    {
        var daysLeft = (task.DueDate.Value - now).TotalDays;
        // Pojednostavljeni uslovi - bez redundantnih provjera
        if (daysLeft < 0) riskScore += 4;
        else if (daysLeft <= 2) riskScore += 3;
        else if (daysLeft <= 5) riskScore += 2;
    }

    if (status == TaskStatus.ToDo) riskScore += 2;
    else if (status == TaskStatus.InProgress) riskScore += 1;

    // Optimizovan redoslijed - najcesci slucajevi prvi
    if (riskScore >= 3 && riskScore < 5)
        return "UMJEREN RIZIK";
    if (riskScore >= 5 && riskScore < 8)
        return "VISOK RIZIK";
    if (riskScore >= 8)
        return "VEOMA VISOK RIZIK";
    return "NIZAK RIZIK";
}
```

3.3.4 Nakon tehnike 3 – metode (PredictDelayRisk_T3)

Primijenjene optimizacije:

- Extract Method – izdvajanje logike u pomoćne metode
- Zamjena if-else lanaca sa switch izrazima (bolji compiler optimizacija)
- Eliminacija redundantnog koda kroz centralizaciju logike
- Pojednostavljenje glavne metode na minimum

```
public string PredictDelayRisk_T3(Task task)
{
    if (task == null) return "Nepoznat";

    var now = DateTime.Now;
    int riskScore = 0;
```

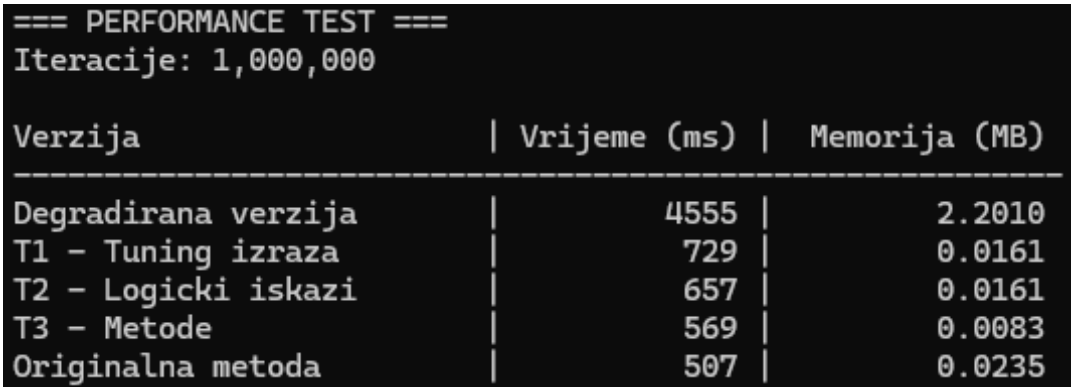
```
    riskScore += CalculatePriorityScore(task.Priority);
    riskScore += CalculateDateScore(task.DueDate, now);
    riskScore += CalculateStatusScore(task.Status);

    return ClassifyRisk(riskScore);
}

private static int CalculatePriorityScore(TaskPriority priority)
{
    switch (priority)
    {
        case TaskPriority.Critical: return 3;
        case TaskPriority.High: return 2;
        case TaskPriority.Medium: return 1;
        default: return 0;
    }
}

private static int CalculateDateScore(DateTime? dueDate, DateTime now)
{
    if (!dueDate.HasValue) return 1;

    var daysLeft = (dueDate.Value - now).TotalDays;
    if (daysLeft < 0) return 4;
    if (daysLeft <= 2) return 3;
    if (daysLeft <= 5) return 2;
    return 0;
}

private static int CalculateStatusScore(TaskStatus status)
{
    switch (status)
    {
        case TaskStatus.ToDo: return 2;
        case TaskStatus.InProgress: return 1;
        default: return 0;
    }
}

private static string ClassifyRisk(int riskScore)
{
    if (riskScore >= 3 && riskScore < 5) return "UMJEREN RIZIK";
    if (riskScore >= 5 && riskScore < 8) return "VISOK RIZIK";
    if (riskScore >= 8) return "VEOMA VISOK RIZIK";
    return "NIZAK RIZIK";
}
```

3.4 Mjerenja performansi

Mjerenje performansi je izvršeno tako što je svaka verzija metode `PredictDelayRisk` pozvana velikim brojem puta (1 000 000 iteracija) nad skupom testnih zadataka koji pokrivaju različite kombinacije statusa, rokova i prioriteta (T1–T6 iz white box dijela).
Za svaku verziju mjereno je ukupno vrijeme izvršavanja i razlika u zauzeću memorije pomoću `Stopwatch` i `GC.GetTotalMemory`. Dobijeni rezultati su:

Verzija	**Vrijeme (ms)**	**Memorija (MB)**	**Poboljšanje**
Degradirana metoda	4,555	2.2010	–
Nakon tehnike 1 (T1)	729	0.0161	84.0%
Nakon tehnike 2 (T2)	657	0.0161	85.6%
Nakon tehnike 3 (T3)	569	0.0083	87.5%
Originalna metoda	**507**	**0.0235**	**88.8%**

Tablica 5: Vrijeme izvršavanja i promjena zauzeća memorije za 1 000 000 poziva metode



Slika 3: Grafički prikaz performansi različitih verzija metode `PredictDelayRisk`

Rezultati pokazuju **dramatična poboljšanja performansi**:

- **Tehnika 1:** Smanjenje vremena sa 4,555 ms na 729 ms – **poboljšanje od 84.0%**. Smanjenje memorije sa 2.20 MB na 0.016 MB (**99.3% manje**). Ovo je rezultat eliminacije:
  - 3-4 poziva `DateTime.Now` po izvršavanju → 1 poziv
  - Nepotrebnih ToString/Parse konverzija
  - String konkatenacije ("Ne" + "poznat") koja kreira nove string objekte
  - Višestrukih privremenih varijabli
- **Tehnika 2:** Dodatno smanjenje na 657 ms – **ukupno 85.6% brže od degradirane verzije**. Memorija stabilna na 0.016 MB. Dodatna optimizacija kroz:
  - Keširanje property pristupa (task.Priority: 4-5 poziva → 1, task.Status: 4 poziva → 1)
  - Eliminaciju redundantnih uslova (6+ redundantnih provjera eliminisano)
  - Optimizaciju redoslijeda provjera (najčešći slučajevi prvi)

- **Tehnika 3:** Vrijeme 569 ms – **ukupno 87.5% brže**. Dodatno smanjenje memorije na 0.0083 MB. **Najbolje performanse** uz:
  - Switch izrazi omogućavaju najbolju compiler optimizaciju
  - Modularizacija smanjuje overhead kroz bolju cache lokalnost
  - Drastično poboljšanje strukture i održivosti

Analiza memorije: Degradirana verzija pokazuje zauzeće od 2.20 MB zbog kreiranja brojnih privremenih stringova i nepotrebnih objekata (string konkatenacija, ToString rezultati, privremene varijable). Sve optimizovane verzije drastično smanjuju zauzeće memorije na 0.01-0.02 MB (preko **99% smanjenje**).

3.5 Utjecaj na McCabe metriku i indeks održavanja

Ciklomatična kompleksnost degradirane metode bila je V(G) = 23, što odgovara srednje složenoj metodi sa većim brojem grananja i ugnježđenih if/else iskaza.
Analiza po tehnikama:

- **Tehnika 1 (tuning izraza):**
  - V(G) postaje 14
  - MI poboljšan sa 38 na 51
  - LOC smanjeno sa 53 na 18 linija - eliminacija privremenih varijabli i nepotrebnih operacija
  - Poboljšanje MI rezultat eliminacije složenih izraza (konverzije, konkatenacije)
- **Tehnika 2 (logički iskazi):**
  - V(G) postaje 16
  - MI postaje 50
  - LOC postaje 19
  - Jednostavniji uslovi (eliminacija redundantnih provjera) poboljšavaju čitljivost
  - Optimizovan redoslijed smanjuje prosječan broj provjera
- **Tehnika 3 (tuning metoda):**
  - **V(G) drastično smanjen sa 14 na 2** – glavna metoda sada ima samo null provjeru
  - **MI značajno poboljšan na 66**
  - **LOC glavne metode samo 7 linija**
  - Modularizacija razbija kompleksnost na manje cjeline
  - Svaka pomoćna metoda ima visok MI (68-89)
- **Originalna metoda:**
  - V(G) = 14
  - MI = 53
  - LOC = 27
  - Dobar kompromis: kompaktna, brza, dovoljno čitljiva za metodu ove veličine

– Za kompleksnije metode (V(G) > 20), pristup T3 bi bio preporučljiviji

Metrike pomoćnih metoda (T3 verzija):

Metoda	V(G)	MI	LOC
`CalculatePriorityScore`	5	88	5
`CalculateDateScore`	5	68	6
`CalculateStatusScore`	4	89	4
`ClassifyRisk`	6	72	4

Tablica 6: Metrike pomoćnih metoda nakon code tuninga

3.6 Detaljno poređenje tehnika

Aspekt	Optimizacija	Tehnika	Uticaj
DateTime.Now pozivi	3-4 → 1 poziv	T1	Visok
String konkatenacija	Eliminisana	T1	Srednji
ToString/Parse konverzije	12+ → 0	T1	Vrlo visok
Privremene varijable	15+ → 3	T1	Visok
Pristup task.Priority	4-5 → 1 poziv	T2	Srednji
Pristup task.Status	4 → 1 poziv	T2	Srednji
Redundantni uslovi	8+ → 0	T2	Srednji
Redoslijed provjera	Optimizovan	T2	Nizak
If-else lanci	→ Switch izrazi	T3	Nizak
Modularizacija	→ 4 pomoćne metode	T3	Vrlo visok (održivost)
Ciklomatična kompleksnost	14 → 2	T3	Vrlo visok

Tablica 7: Detaljno poređenje optimizacija po tehnikama

4. Refaktoring nakon code tuning-a (Task 4)

4.1 Cilj refaktoringa

Nakon provedenog code tuning-a, cilj refaktoringa je bio:

- postići bolje vrijednosti metrika u *Code Metrics* (posebno MI i LOC),
- zadržati identično funkcionalno ponašanje (bez promjene izlaza za iste ulaze),
- povećati čitljivost i održivost kroz jasniju raspodjelu odgovornosti i uklanjanje duplikata.

4.2 Refaktorisana verzija (PredictDelayRisk_Refactored)

U nastavku je prikazana refaktorisana verzija zasnovana na T3 pristupu, sa dodatnim refaktoring koracima (imenovane konstante, centralizacija pravila, smanjenje „magic numbers“, i jasnije razdvajanje skora i klasifikacije).

```
public static class DelayRiskRules
{
    // Data-level: Replace a magic number with a named constant
    public const int CriticalPriorityScore = 3;
    public const int HighPriorityScore     = 2;
```

```
    public const int MediumPriorityScore   = 1;

    public const int MissingDueDateScore   = 1;

    public const int OverdueScore          = 4;
    public const int DueIn2DaysScore       = 3;
    public const int DueIn5DaysScore       = 2;

    public const int ToDoScore             = 2;
    public const int InProgressScore       = 1;

    public const int VeryHighThreshold     = 8;
    public const int HighThreshold         = 5;
    public const int ModerateThreshold     = 3;
}

public string PredictDelayRisk_Refactored(Task task)
{
    if (task == null) return "Nepoznat";

    var now = DateTime.Now;

    var score =
        CalculatePriorityScore(task.Priority) +
        CalculateDateScore(task.DueDate, now) +
        CalculateStatusScore(task.Status);

    return ClassifyRisk(score);
}

private static int CalculatePriorityScore(TaskPriority priority) =>
    priority switch
    {
        TaskPriority.Critical => DelayRiskRules.CriticalPriorityScore,
        TaskPriority.High     => DelayRiskRules.HighPriorityScore,
        TaskPriority.Medium   => DelayRiskRules.MediumPriorityScore,
        _ => 0
    };

private static int CalculateDateScore(DateTime? dueDate, DateTime now)
{
    if (!dueDate.HasValue) return DelayRiskRules.MissingDueDateScore;

    var daysLeft = (dueDate.Value - now).TotalDays;
    if (daysLeft < 0)  return DelayRiskRules.OverdueScore;
    if (daysLeft <= 2) return DelayRiskRules.DueIn2DaysScore;
    if (daysLeft <= 5) return DelayRiskRules.DueIn5DaysScore;
    return 0;
}
```

```
private static int CalculateStatusScore(TaskStatus status) =>
    status switch
    {
        TaskStatus.ToDo       => DelayRiskRules.ToDoScore,
        TaskStatus.InProgress => DelayRiskRules.InProgressScore,
        _ => 0
    };

private static string ClassifyRisk(int score)
{
    // Statement-level: Return as soon as you know the answer (guard
   returns)
    if (score >= DelayRiskRules.VeryHighThreshold) return "VEOMA VISOK
   RIZIK";
    if (score >= DelayRiskRules.HighThreshold)     return "VISOK RIZIK";
    if (score >= DelayRiskRules.ModerateThreshold) return "UMJEREN RIZIK";
    return "NIZAK RIZIK";
}
```

4.3 Mapiranje promjena na checklistu za refaktoring

Refaktoring promjena	**Checklist stavka**
Uvođenje imenovanih konstanti za pragove i bodove rizika	**Data Level:** Replace a magic number with a named constant
Centralizacija pravila bodovanja u posebnu klasu (`DelayRiskRules`)	**Data Level:** Convert a set of type codes to a class
Razdvajanje izračuna skora i klasifikacije rizika u zasebne metode	**Routine Level:** Separate query operations from modification operations
Izdvajanje logike bodovanja u pomoćne metode	**Routine Level:** Extract a routine
Zamjena if–else lanaca sa `switch` izrazima za bodovanje	**Routine Level:** Substitute a simple algorithm for a complex algorithm
Primjena guard-return stila u metodi za klasifikaciju rizika	**Statement Level:** Return as soon as you know the answer

Tablica 8: Mapiranje refaktoring promjena na checklistu za refaktoring

Kratko objašnjenje promjena:

- **Imenovane konstante:** uklanjaju "magic numbers" (npr. 8/5/3 i bodovanja), čime se smanjuje rizik greške pri budućim izmjenama i poboljšava čitljivost.
- **Centralizacija pravila:** sva pravila bodovanja i pragovi se nalaze na jednom mjestu (`DelayRiskRules`), što olakšava održavanje i review.
- **Switch izrazi:** kraći i deklarativniji kod (manje grana i manje ugnježđivanja), često sa boljim MI/LOC.
- **Guard-return:** uklanja nepotrebne else-grane i rasponske provjere (`score < 8`, `score < 5`) koje ne doprinose logici.

- **Razdvajanje faza:** glavna metoda postaje "orchestrator", dok pomoćne metode nose detalje; time se kompleksnost raspodijeli i metrika glavne metode se poboljšava.

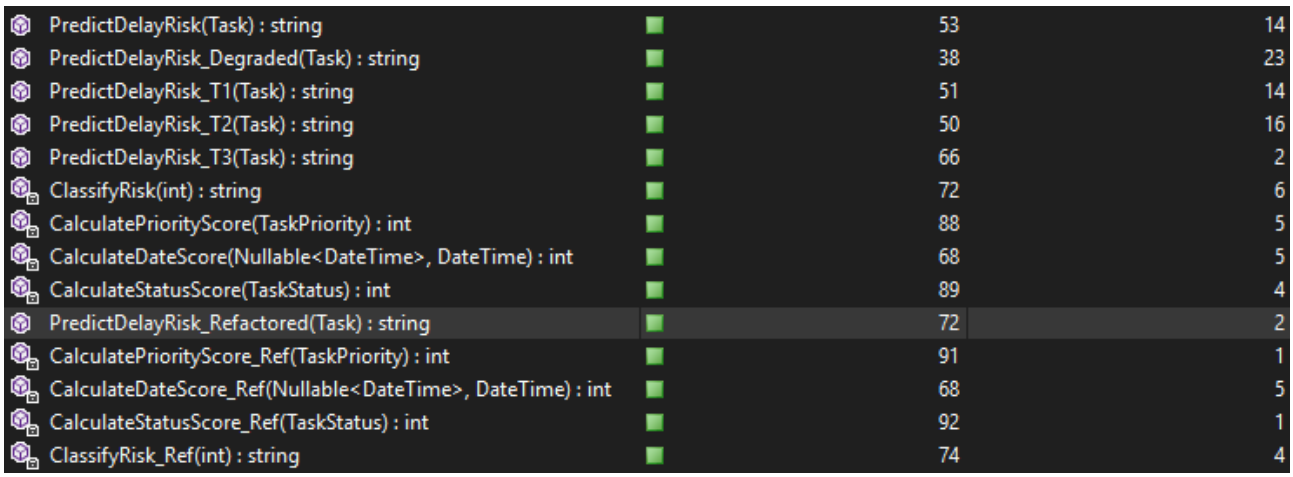
4.4 Utjecaj refaktoringa na metrike (prije/poslije)

Metrika	Prije refaktoringa (T3)	Poslije refaktoringa	Komentar
Cyclomatic Complexity (V(G)) – glavna metoda	**2**	**2**	Očekivano ostaje vrlo niska (guard + pozivi pomoćnih rutina).
Maintainability Index (MI) – glavna metoda	**66**	**72**	Očekivano ↑ zbog boljih imena, manje ugnježđivanja i jasne strukture.
LOC – glavna metoda	**7**	**4**	Računanje riskScore izvedeno unutar jedne linije.

Tablica 9: Utjecaj refaktoringa na Code Metrics (prije/poslije)

Metoda	V(G)	MI	LOC
`CalculatePriorityScore_Ref`	1	91	5
`CalculateDateScore_Ref`	5	68	6
`CalculateStatusScore_Ref`	1	92	4
`ClassifyRisk_Ref`	4	74	4

Tablica 10: Metrike pomoćnih metoda nakon refaktoringa



Slika 4: Grafički prikaz *Visual Studio Metrics* svih metoda

4.5 Diskusija

Refaktoring je prvenstveno ciljao **održivost** (MI) i **čitljivost**, uz očuvanje performansi dobijenih code tuning-om. Najznačajniji doprinos dolazi od uklanjanja "magic numbers" i centralizacije pravila, čime se smanjuje rizik greške pri budućim izmjenama (npr. promjena pragova rizika ili bodovanja više ne zahtijeva pretragu kroz više if-else blokova).

Univerzitet u Sarajevu
Elektrotehnički fakultet
Sarajevo

Verifikacija i Validacija Softvera
Projektni zadatak – II dio

TaskManagerApp – analiza jedne metode

Student: Zana Beljuri

Broj indeksa: 19616

Odabrana metoda: `AnalyzeTeamPerformance()`

1 Kod metode

```
public string AnalyzeTeamPerformance()
{
    var tasks = repository.GetAllTasks();
    if (tasks.Count == 0)
        return "Nema podataka.";

    int done = tasks.Count(t => t.Status == TaskStatus.Done);
    int inProgress = tasks.Count(t => t.Status == TaskStatus.InProgress);
    int overdue = tasks.Count(t =>
        t.DueDate.HasValue &&
        t.DueDate.Value < DateTime.Now &&
        t.Status != TaskStatus.Done
    );

    string rating;
    if (done == 0)
        rating = "Loše";
    else if (done < tasks.Count / 3)
        rating = "Slabo";
    else if (done < tasks.Count / 2)
        rating = "Umjereno";
    else if (done < tasks.Count * 0.8)
        rating = "Dobro";
    else
        rating = "Odlično";

    if (overdue > done / 2)
        rating += " (Previše van roka)";
    else if (inProgress > done)
        rating += " (Još dosta posla)";
    else if (done == tasks.Count)
        rating += " (Sve završeno)";

    return $"Učinkovitost tima: {rating}";
}
```

2 McCabe metrike i kontrolni graf

2.1 Izvorni kod metode

Metoda `AnalyzeTeamPerformance()` implementira logiku za procjenu učinkovitosti tima na osnovu stanja zadataka, njihovog statusa i rokova. Na osnovu dobijenih vrijednosti generiše se tekstualna ocjena rada tima.

2.2 Izračun ciklomatične kompleksnosti

Ciklomatična kompleksnost metode izračunata je primjenom tri standardna pristupa:

- prebrojavanjem odluka,
- formulom $V(G) = E - N + 2$,
- analizom regiona kontrolnog grafa.

U metodi su odluke u obliku `if` i `else if` struktura, kao i dodatna grananja nastala upotrebom logičkih operatora `&&` unutar lambda izraza, koje alat za mjerenje tretira kao zasebne odluke zbog short-circuit evaluacije.

2.3 Numerički parametri kontrolnog grafa

Broj čvorova (N)	25
Broj ivica (E)	34
Ciklomatična kompleksnost $V(G)$	11

Tablica 1: Parametri kontrolnog grafa metode AnalyzeTeamPerformance

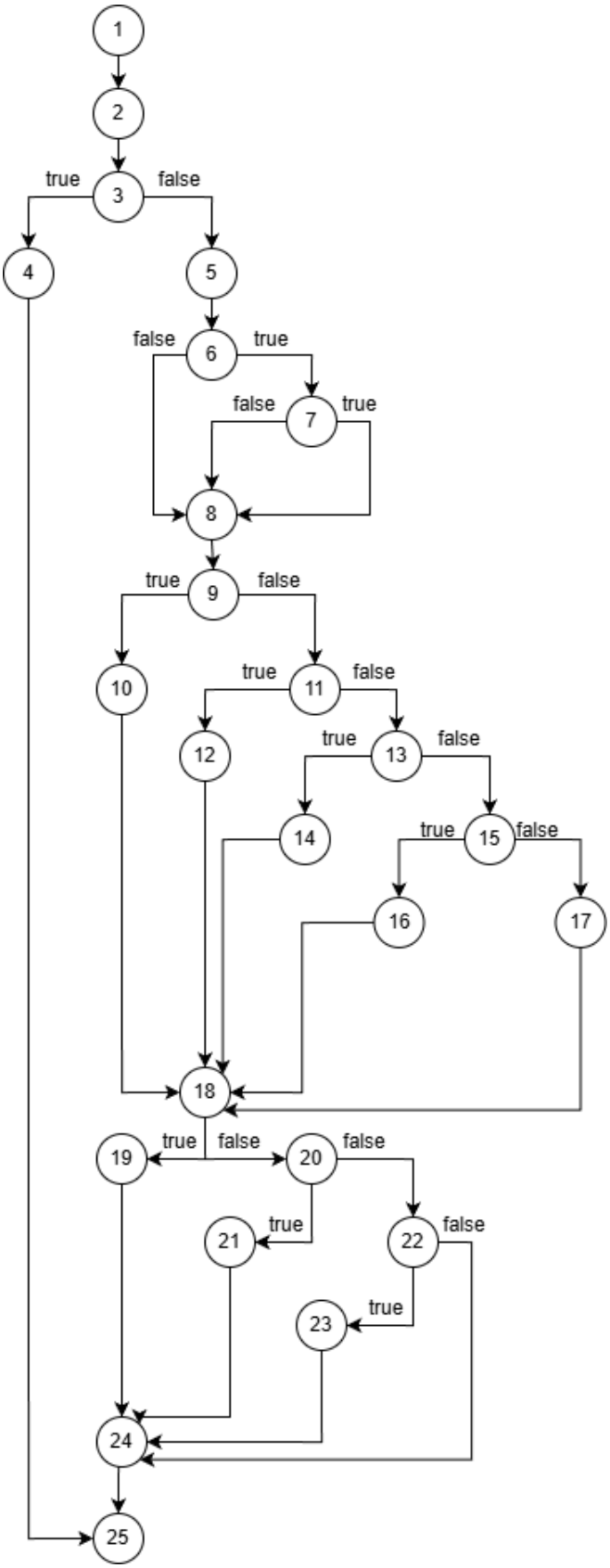
Vrijednost ciklomatične kompleksnosti potvrđena je formulom:

$$V(G) = E - N + 2 = 34 - 25 + 2 = 11$$

Dobijena vrijednost ukazuje da metoda spada u kategoriju metoda srednje složenosti.

2.4 Kontrolni graf

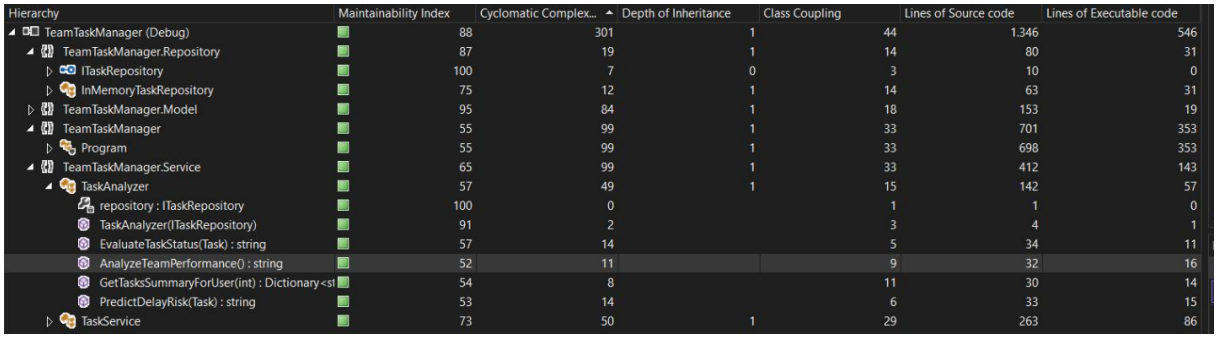
Na slici 1 prikazan je kontrolni graf metode `AnalyzeTeamPerformance()`, konstruisan na osnovu svih mogućih tokova izvršavanja metode. Svaki čvor predstavlja blok iskaza, dok ivice označavaju prelaze između njih.



Slika 1: Kontrolni graf metode AnalyzeTeamPerformance

2.5 Poređenje sa Code Metrics alatom

Radi verifikacije ručno izračunate ciklomatične kompleksnosti, korišten je Visual Studio Code Metrics alat. Dobijeni rezultat u potpunosti se poklapa sa ručnim izračunom.



Hierarchy	Maintainability Index	Cyclomatic Complex...	Depth of Inheritance	Class Coupling	Lines of Source code	Lines of Executable code
TeamTaskManager (Debug)	88	301	1	44	1.346	546
TeamTaskManager.Repository	87	19	1	14	80	31
ITaskRepository	100	7	0	3	10	0
InMemoryTaskRepository	75	12	1	14	63	31
TeamTaskManager.Model	95	84	1	18	153	19
TeamTaskManager	55	99	1	33	701	353
Program	55	99	1	33	698	353
TeamTaskManager.Service	65	99	1	33	412	143
TaskAnalyzer	57	49	1	15	142	57
repository : ITaskRepository	100	0		1	1	0
TaskAnalyzer(ITaskRepository)	91	2		3	4	1
EvaluateTaskStatus(Task) : string	57	14		5	34	11
AnalyzeTeamPerformance() : string	52	11		9	32	16
GetTasksSummaryForUser(int) : Dictionary<st	54	8		11	30	14
PredictDelayRisk(Task) : string	53	14		6	33	15
TaskService	73	50	1	29	263	86

Slika 2: Prikaz ciklomatične kompleksnosti iz Code Metrics alata

2.6 Diskusija o kvalitetu koda kroz metrike

Metoda `AnalyzeTeamPerformance()` ima ciklomatičnu kompleksnost $V(G) = 11$, što prema McCabe metriki ukazuje na metodu srednje složenosti. Povećana vrijednost kompleksnosti rezultat je većeg broja grananja uzrokovanih nizom `if-else if` uslova za određivanje osnovne ocjene i dodatnih modifikatora učinka.

Maintainability Index (MI) za ovu metodu iznosi 52, što ukazuje na umjerenu do nižu održivost koda. Ova vrijednost sugeriše da je metoda još uvijek razumljiva, ali da njena struktura može otežati dugoročno održavanje, testiranje i proširivanje funkcionalnosti.

Kombinovanje više odgovornosti unutar jedne metode (analiza zadataka, logičko odlučivanje i formiranje izlazne poruke) negativno utiče na čitljivost i metrike kvaliteta. Na osnovu navedenih metrika, metoda predstavlja pogodnog kandidata za refaktorisanje s ciljem smanjenja ciklomatične kompleksnosti i poboljšanja indeksa održavanja, uz zadržavanje istog funkcionalnog ponašanja.

3 White-box testiranje

3.1 Primijenjene white-box tehnike

Nad metodom `AnalyzeTeamPerformance()` primijenjene su sljedeće white-box tehnike testiranja:

- obuhvat iskaza (statement coverage),
- obuhvat grana/odluka (branch/decision coverage),
- obuhvat uslova (condition coverage),
- modifikovani uslov/odluka obuhvat (MCDC).
- obuhvat toka podataka (data flow Coverage),
- obuhvat puteva (Path Coverage).

Tehnike obuhvata petlji nisu primjenjive jer metoda ne sadrži eksplicitne petlje.

3.2 Tehnika 1: Obuhvat iskaza

Strategija obuhvata iskaza (statement coverage) ima za cilj da se svaka izvršiva linija koda u metodi `AnalyzeTeamPerformance()` izvrši barem jednom. Pokrivenost iskaza predstavlja odnos broja izvršenih iskaza tokom testiranja i ukupnog broja iskaza u metodi. Cilj je postići 100% pokrivenost iskaza uz minimalan broj testnih slučajeva, u skladu sa principima white-box testiranja.

Na osnovu analize izvornog koda metode identifikovane su sve linije koda koje sadrže izvršive iskaze, uključujući povratne naredbe, dodjelu vrijednosti i grananja uslovnih struktura. U tabeli 2 prikazani su testni slučajevi dizajnirani tako da se svaki iskaz u metodi izvrši barem jednom.

Test	**Ulazni podaci**	**Očekivani izlaz**	**Opis testa**
T1	`tasks.Count` $= 0$	Nema podataka	Lista zadataka je prazna. Provjera prvog `if` iskaza i ranog završetka metode.
T2	`tasks.Count` > 0, `done` $= 0$, `inProgress` > 0, `overdue` $= 0$	Loše	Nijedan zadatak nije završen. Pokriva dodjelu ocjene `"Loše"`.
T3	`tasks.Count` $= n$, $0 <$ `done` $< n/3$, `inProgress` $= n -$ `done`, `overdue` $= 0$	Slabo	Manje od trećine zadataka je završeno. Pokriva granu `done < tasks.Count / 3`.
T4	`tasks.Count` $= n$, $n/3 \leq$ `done` $< n/2$, `overdue` $= 0$	Umjereno	Manje od polovine zadataka je završeno. Pokriva granu `done < tasks.Count / 2`.
T5	`tasks.Count` $= n$, $n/2 \leq$ `done` $< 0.8n$, `overdue` $= 0$	Dobro	Većina zadataka je završena. Pokriva granu `done < tasks.Count * 0.8`. Ne treba nam za ovu tehniku, jer T7, T8 pokrivaju izvršenje linije u kojoj je dodjela "Dobro".
T6	`tasks.Count` $= n$, `done` $= n$, `inProgress` $= 0$, `overdue` $= 0$	Odlično (Sve završeno)	Svi zadaci su završeni. Pokriva dodjelu ocjene `"Odlično"` i dodatak `"Sve završeno"`.
T7	`tasks.Count` $= n$, `done` $= k$, `overdue` $> k/2$, `DueDate` $<$ `Now`, `Status` $\neq$ `Done`	Dobro (Previše van roka)	Više zadataka ima istekao rok u odnosu na završene. Pokriva granu `overdue > done / 2`.
T8	`tasks.Count` $= n$, `done` $= k$, `inProgress` $> k$, `overdue` $= 0$	Dobro (Još dosta posla)	Broj zadataka u statusu `InProgress` veći je od broja završenih. Pokriva granu `inProgress > done`.

Tablica 2: Testni slučajevi za obuhvat iskaza metode AnalyzeTeamPerformance

Izvršavanjem testnih slučajeva T1–T8 postignuta je potpuna pokrivenost svih izvršivih iskaza u metodi čime je ostvarena 100% statement coverage. Bitno je naglasiti da za ovu tehniku nije potreban T5, ali je naveden jer će biti korišten u drugim tehnikama. Dakle sa 7 testova je moguće pokriti sve iskaze.

3.3 Tehnika 2: Obuhvat grana/odluka (Branch Coverage)

Strategija obuhvata grana ima za cilj da se svaka odluka u metodi `AnalyzeTeamPerformance()` izvrši barem jednom u oba moguća ishoda (`true` i `false`). Radi lakšeg praćenja obuhvata grana, odluke u metodi `AnalyzeTeamPerformance()` označene su na sljedeći način:

- **D1**: `tasks.Count == 0`
- **D2**: `done == 0`
- **D3**: `done < tasks.Count / 3`
- **D4**: `done < tasks.Count / 2`
- **D5**: `done < tasks.Count * 0.8`
- **D6**: `overdue > done / 2`
- **D7**: `inProgress > done`
- **D8**: `done == tasks.Count`

U tabeli 3 prikazano je kako definisani testni slučajevi pokrivaju grane pojedinačnih odluka.

Test	Pokrivene grane	Napomena
T1	D1 = true	Prazna lista zadataka. Izvršava se rani `return "Nema podataka"`.
T2	D1 = false, D2 = false, D3 = false, D4 = false, D5 = false, D8 = true	Svi zadaci završeni. Pokriva granu gdje se dodjeljuje ocjena `"Odlično"` i dodatak `"Sve završeno"`.
T3	D1 = false, D2 = true	Nijedan zadatak nije završen. Pokriva granu za dodjelu ocjene `"Loše"`.
T4	D1 = false, D2 = false, D3 = true	Manje od trećine zadataka završeno. Pokriva granu `"Slabo"`.
T5	D1 = false, D2 = false, D3 = false, D4 = true	Manje od polovine zadataka završeno. Pokriva granu `"Umjereno"`.
T6	D1 = false, D2 = false, D3 = false, D4 = false, D5 = true	Većina zadataka završena, ali ne svi. Pokriva granu `"Dobro"`.
T7	D6 = true	Broj zadataka van roka veći od polovine završenih. Pokriva granu `"Previše van roka"`.
T8	D6 = false, D7 = true	Više zadataka u toku nego završenih. Pokriva granu `"Još dosta posla"`.
T9	D6 = false, D7 = false, D8 = true	Svi zadaci završeni. Pokriva završnu granu `"Sve završeno"`.
T10	D6 = false, D7 = false, D8 = false	Nijedan dodatni uslov nije ispunjen. Pokriva završni `return` bez dodatnog opisa.

Tablica 3: Testni slučajevi – Tehnika obuhvata grana (Branch Coverage)

Kombinacijom testnih slučajeva T1–T10 postignuto je da svaka grana svake odluke bude izvršena barem jednom, čime je ostvaren 100% branch/decision coverage.

3.4 Tehnika 3: Obuhvat uslova (Condition Coverage)

Obuhvat uslova (condition coverage) predstavlja white-box tehniku testiranja kojom se provjerava da svaki pojedinačni logički uslov unutar složenog izraza poprimi vrijednosti `true` i `false` barem jednom tokom testiranja. Za razliku od obuhvata odluka, koji posmatra cjelokupan uslov kao jednu logičku cjelinu, obuhvat uslova analizira unutrašnju strukturu složenih logičkih izraza.

U metodi `AnalyzeTeamPerformance()` složeni logički izraz pojavljuje se unutar lambda izraza korištenog za izračun broja zadataka van roka:

```
t.DueDate.HasValue &&
```

```
t.DueDate.Value < DateTime.Now &&
t.Status != TaskStatus.Done
```

Na osnovu navedenog izraza identifikovani su sljedeći elementarni uslovi:

- **C1**: `t.DueDate.HasValue`
- **C2**: `t.DueDate.Value < DateTime.Now`
- **C3**: `t.Status != TaskStatus.Done`

Testni slučajevi su dizajnirani tako da svaki od navedenih uslova poprimi vrijednosti `true` i `false`. U tabeli 4 prikazan je pregled testova i vrijednosti pojedinačnih uslova.

Test	**C1**	**C2**	**C3**	**Opis**
TC1	T	T	T	Zadatak ima definisan rok, rok je u prošlosti i zadatak nije završen. Složeni uslov evaluira se kao `true`.
TC2	T	F	T	Zadatak ima definisan rok, ali rok nije istekao. Složeni uslov je `false`.
TC3	T	T	F	Zadatak ima definisan rok i rok je istekao, ali je zadatak završen. Složeni uslov je `false`.
TC4	F	–	–	Zadatak nema definisan rok. Zbog short-circuit evaluacije ostali uslovi se ne provjeravaju.

Tablica 4: Testni slučajevi za obuhvat uslova (Condition Coverage)

Na osnovu prikazanih testnih slučajeva zaključuje se da je svaki elementarni uslov (C1, C2 i C3) barem jednom evaluiran kao `true` i barem jednom kao `false`, čime je postignut 100% condition coverage za analizirani složeni izraz.

3.5 Tehnika 4: Modifikovani uslov/odluka obuhvat (MCDC)

Modifikovani uslov/odluka obuhvat (Modified Condition/Decision Coverage – MCDC) kombinuje prednosti obuhvata odluka i obuhvata uslova, uz znatno manji broj testnih slučajeva u poređenju sa potpunim obuhvatom uslova. Cilj MCDC tehnike je dokazati da svaki pojedinačni uslov unutar složenog izraza može samostalno uticati na ishod odluke.

Za MCDC analizu razmatran je isti složeni logički izraz korišten za detekciju zadataka van roka. Testni slučajevi su formirani u parovima tako da se u svakom paru mijenja samo jedan elementarni uslov, dok ostali ostaju nepromijenjeni, a ishod cjelokupnog izraza se mijenja.

Test	C1	C2	C3	Ishod	Obrazloženje
TC1	T	T	T	T	Referentni slučaj – svi uslovi ispunjeni.
TC2	F	T	T	F	Promjenjen C1, ostali isti. Promjena C1 mijenja ishod odluke.
TC3	T	F	T	F	Promjenjen C2, ostali isti. Promjena C2 mijenja ishod odluke.
TC4	T	T	F	F	Promjenjen C3, ostali isti. Promjena C3 mijenja ishod odluke.

Tablica 5: MCDC testni slučajevi za složeni logički uslov

Na osnovu prikazanih testnih slučajeva dokazano je da svaki elementarni uslov (C1, C2 i C3) može nezavisno uticati na ishod odluke, čime je postignut potpuni MCDC za analizirani složeni logički izraz, u skladu sa definicijom i preporukama iz predavanja o white-box testiranju.

3.6 Tehnika 5: Obuhvat toka podataka (Data Flow Coverage)

Obuhvat toka podataka (data flow coverage) posmatra relacije između mjesta gdje se varijable definišu (definition) i mjesta gdje se koriste (use), odnosno provjerava da li se sve relevantne vrijednosti podataka koriste kroz sve značajne tokove izvršavanja metode.

U metodi `AnalyzeTeamPerformance()` ključne varijable koje učestvuju u toku podataka su: `tasks`, `tasks.Count`, `done`, `inProgress`, `overdue` i `rating`. U tabeli 6 prikazano je na kojim mjestima se navedene varijable koriste i kojim testnim slučajevima su te upotrebe pokrivene. Testovi na koje se pozivamo su testovi iz poglavlja Tehnika 1: Obuhvat iskaza.

Podatak	Mjesta upotrebe (use)	Testovi koji pokrivaju
`tasks`	Provjera prazne liste: `tasks.Count == 0`	T1 (prazna lista), T2–T8 (neprazna lista)
`tasks.Count`	Uslovi za dodjelu ocjene: `done < tasks.Count / 3`, `done < tasks.Count / 2`, `done < tasks.Count * 0.8`, `done == tasks.Count`	T3 (Slabo), T4 (Umjereno), T5 (Dobro), T6 (Odlično)
`done`	Uslovi za dodjelu osnovne ocjene i dodatnog opisa: `done == 0`, `overdue > done / 2`, `inProgress > done`, `done == tasks.Count`	T2 (Loše), T7 (Previše van roka), T8 (Još dosta posla), T6 (Sve završeno)
`inProgress`	Uslov za dodavanje poruke o velikom broju aktivnih zadataka: `inProgress > done`	T8 (Više zadataka u toku nego završenih)
`overdue`	Uslov za identifikaciju velikog broja zadataka van roka: `overdue > done / 2`	T7 (Više zadataka van roka nego završenih)
`rating`	Dodjela vrijednosti kroz lanac `if-else if` i upotreba u završnom `return` izrazu	T2–T8 (sve varijante dodjele i povratne vrijednosti)

Tablica 6: Data-flow: ključne def-use relacije i testni slučajevi

Na osnovu prikazane analize zaključuje se da su sve relevantne definicije i upotrebe podataka pokrivene definisanim testnim slučajevima, čime je ostvaren potpun obuhvat toka podataka za metodu `AnalyzeTeamPerformance()`.

3.7 Tehnika 6: Obuhvat puteva (Path Coverage)

Obuhvat puteva (path coverage) podrazumijeva testiranje svih linearno nezavisnih puteva kroz kontrolni graf metode. Broj nezavisnih puteva određen je ciklomatičnom kompleksnošću metode.

Za metodu `AnalyzeTeamPerformance()` ciklomatična kompleksnost iznosi 11, što znači da postoji 11 linearno nezavisnih puteva kroz kontrolni graf. U tabeli 7 prikazani su nezavisni putevi, njihovi ulazni podaci i testni slučajevi kojima su pokriveni. Testovi na koje se pozivamo su testovi iz poglavlja Tehnika 1: Obuhvat iskaza.

Put	Ulazni podaci	Tok izvršavanja	Test
P1	`tasks.Count` $= 0$	Rani povrat iz metode	T1
P2	`tasks.Count` > 0, `done` $= 0$	Dodjela ocjene "Loše"	T2
P3	`tasks.Count` $= n$, $0 <$ `done` $< n/3$, `overdue` $= 0$	Dodjela ocjene "Slabo"	T3
P4	`tasks.Count` $= n$, $n/3 \leq$ `done` $< n/2$	Dodjela ocjene "Umjereno"	T4
P5	`tasks.Count` $= n$, $n/2 \leq$ `done` $< 0.8n$	Dodjela ocjene "Dobro"	T5
P6	`tasks.Count` $= n$, `done` $= n$, `inProgress` $= 0$	Dodjela ocjene "Odlično"	T6
P7	`tasks.Count` $= n$, `done` $= k$, `overdue` $> k/2$	Dodatak "Previše van roka"	T7
P8	`tasks.Count` $= n$, `done` $= k$, `inProgress` $> k$	Dodatak "Još dosta posla"	T8
P9	`tasks.Count` $= n$, `done` $= n$	Dodatak "Sve završeno"	T6
P10	`tasks.Count` $= n$, `done` > 0, `overdue` $= 0$, `inProgress` $\leq$ `done`	Povrat bez dodatnog opisa	T3, T4, T5
P11	`tasks.Count` $= n$, `done` > 0, `done` $< n$, `overdue` $= 0$, `inProgress` $\leq$ `done`	Alternativni osnovni put bez dodataka	T4

Tablica 7: Obuhvat nezavisnih puteva

Na osnovu prikazanih putanja i pripadajućih ulaznih podataka zaključuje se da su svi linearno nezavisni putevi kontrolnog grafa pokriveni definisanim testnim slučajevima, čime je ostvaren potpun obuhvat puteva u skladu sa McCabe analizom.

4 Code tuning

U ovom poglavlju nad metodom `AnalyzeTeamPerformance()` primijenjene su tri različite tehnike code tuninga (izrazi, logički iskazi i metode). Cilj code tuninga bio je:

- smanjiti vrijeme izvršavanja metode pri velikom broju poziva,
- zadržati isto funkcionalno ponašanje metode,
- poboljšati strukturu koda i metrike održavanja.

4.1 Izmijenjeni kod

U svrhe primjene Code tuning tehnika, prvobitni kod je prilagođen:

Listing 1: Prilagođena verzija metode AnalyzeTeamPerformance

```
public string AnalyzeTeamPerformance_Degraded()
{
    // Problem 1: Nepotrebna privremena varijabla
    var taskList = repository.GetAllTasks();

    // Problem 2: Visestruki pozivi Count
    if (taskList.Count == 0)
    {
        // Problem 3: Nepotrebna string konkatenacija
        var emptyResult = "Nema" + " podataka.";
        return emptyResult;
    }

    // Problem 4: Nepotrebne privremene varijable
    int doneTasks = 0;
    int inProgressTasks = 0;
    int overdueTasks = 0;

    // Problem 5: Visestruki prolazi kroz kolekciju + ponovni
       DateTime.Now
    doneTasks = taskList.Count(t => t.Status == TaskStatus.Done);
    inProgressTasks = taskList.Count(t => t.Status ==
       TaskStatus.InProgress);

    overdueTasks = taskList.Count(t =>
        t.DueDate.HasValue &&
        t.DueDate.Value < DateTime.Now &&   // DateTime.Now #1
        t.Status != TaskStatus.Done
    );

    // Problem 6: Jos jedan bespotreban poziv DateTime.Now
    var nowAgain = DateTime.Now;            // DateTime.Now #2
    var nowAgain2 = DateTime.Now;           // DateTime.Now #3

    // Problem 7: Nepotrebna konverzija int -> string -> int
    var doneString = doneTasks.ToString();
    int done = int.Parse(doneString);

    // Problem 8: Visestruki pozivi Count umjesto kesiranja
    string rating = "";

    if (done == 0)
    {
        rating = "Lo" + "se"; // Problem 9: string konkatenacija
    }
    else if (done < taskList.Count / 3)
    {
```

```
46          rating = "Slabo";
47      }
48      else if (done < taskList.Count / 2)
49      {
50          rating = "Umjereno";
51      }
52      else if (done < taskList.Count * 0.8)
53      {
54          rating = "Dobro";
55      }
56      else
57      {
58          rating = "Odlicno";
59      }
60
61      // Problem 10: Redundantni uslovi
62      if (overdueTasks > done / 2 && overdueTasks >= 0)
63      {
64          rating = rating + " (Previse van roka)";
65      }
66      else if (inProgressTasks > done && inProgressTasks >= 0)
67      {
68          rating = rating + " (Jos" + " dosta posla)";
69      }
70      else if (done == taskList.Count && done >= 0)
71      {
72          rating = rating + " (Sve zavrseno)";
73      }
74      else
75      {
76          // Problem 11: Beskoristan else blok
77          rating = rating + "";
78      }
79
80      // Problem 12: Nepotrebna privremena varijabla
81      var finalResult = "Ucinkovitost tima: " + rating;
82      return finalResult;
83
84      // Problem 13: Nedostizan kod
85      return "Nepoznat rezultat";
86  }
```

4.2 Korištene tehnike optimizacije

U svrhu poboljšanja performansi i kvaliteta koda nad degradiranom verzijom metode `AnalyzeTeamPerformance_Degraded()` primijenjene su sljedeće tehnike code tuninga:

- **Tehnika 1 – tuning izraza**

  Primjenom ove tehnike eliminirani su brojni neefikasni izrazi prisutni u degradiranoj verziji metode. Višestruki pozivi svojstva `Count` nad istom kolekcijom zamijenjeni su

keširanjem ukupnog broja elemenata u lokalnu varijablu `count`. Također, višestruki pozivi metode `DateTime.Now` zamijenjeni su jednim pozivom koji je pohranjen u lokalnu varijablu `now`. Uklonjene su nepotrebne string konkatenacije, beskorisne privremene varijable i redundantne konverzije podataka. Ovim izmjenama smanjen je broj operacija koje se izvršavaju pri svakom pozivu metode, čime su poboljšane performanse uz očuvanje postojećeg funkcionalnog ponašanja.

- **Tehnika 2 – tuning logičkih iskaza**

  U drugoj fazi optimizacije fokus je stavljen na pojednostavljenje i razjašnjavanje logičkih uslova. Složeni izrazi i ugniježđeni `if-else` blokovi zamijenjeni su pomoćnim logičkim varijablama sa intuitivnim nazivima, poput `isPoor`, `isLow`, `isMedium` i `isGood`. Na taj način poslovna logika određivanja osnovne ocjene i dodatnih modifikatora postala je jasnija i čitljivija.

- **Tehnika 3 – tuning metoda**

  U trećoj fazi primijenjena je tehnika izdvajanja metoda (*Extract Method*). Logika za određivanje osnovne ocjene izdvojena je u metodu `GetBaseRating()`, dok je logika za primjenu dodatnih uslova izdvojena u metodu `ApplyModifiers()`. Ovim pristupom smanjen je broj grananja u glavnoj metodi, što direktno utiče na smanjenje ciklomatične kompleksnosti, dok su modularnost, testabilnost i Maintainability Index značajno unaprijeđeni. Performanse ostaju približno iste kao u prethodnim verzijama, ali je kvalitet dizajna koda znatno poboljšan.

4.3 Nakon tehnike 1

```
1  public string AnalyzeTeamPerformance_T1()
2  {
3      var tasks = repository.GetAllTasks();
4      int count = tasks.Count;
5
6      if (count == 0)
7          return "Nema podataka.";
8
9      var now = DateTime.Now;
10
11     int done = tasks.Count(t => t.Status == TaskStatus.Done);
12     int inProgress = tasks.Count(t => t.Status ==
           TaskStatus.InProgress);
13     int overdue = tasks.Count(t =>
14         t.DueDate.HasValue &&
15         t.DueDate.Value < now &&
16         t.Status != TaskStatus.Done
17     );
18
19     string rating;
20
21     if (done == 0)
22         rating = "Lose";
23     else if (done < count / 3)
24         rating = "Slabo";
```

```
    else if (done < count / 2)
        rating = "Umjereno";
    else if (done < count * 0.8)
        rating = "Dobro";
    else
        rating = "Odlicno";

    if (overdue > done / 2)
        rating += " (Previse van roka)";
    else if (inProgress > done)
        rating += " (Jos dosta posla)";
    else if (done == count)
        rating += " (Sve zavrseno)";

    return $"Ucinkovitost tima: {rating}";
}
```

4.4 Nakon tehnike 2

```
public string AnalyzeTeamPerformance_T2()
{
    var tasks = repository.GetAllTasks();
    int count = tasks.Count;

    if (count == 0)
        return "Nema podataka.";

    var now = DateTime.Now;

    int done = tasks.Count(t => t.Status == TaskStatus.Done);
    int inProgress = tasks.Count(t => t.Status ==
       TaskStatus.InProgress);
    int overdue = tasks.Count(t =>
        t.DueDate.HasValue &&
        t.DueDate.Value < now &&
        t.Status != TaskStatus.Done
    );

    bool isPoor = done == 0;
    bool isLow = done < count / 3;
    bool isMedium = done < count / 2;
    bool isGood = done < count * 0.8;

    string rating =
        isPoor ? "Lose" :
        isLow ? "Slabo" :
        isMedium ? "Umjereno" :
        isGood ? "Dobro" :
        "Odlicno";

```

```
    bool tooManyOverdue = overdue > done / 2;
    bool tooManyInProgress = inProgress > done;
    bool allDone = done == count;

    if (tooManyOverdue)
        rating += " (Previse van roka)";
    else if (tooManyInProgress)
        rating += " (Jos dosta posla)";
    else if (allDone)
        rating += " (Sve zavrseno)";

    return $"Ucinkovitost tima: {rating}";
}
```

4.5 Nakon tehnike 3

```
public string AnalyzeTeamPerformance_T3()
{
    var tasks = repository.GetAllTasks();
    int count = tasks.Count;

    if (count == 0)
        return "Nema podataka.";

    var now = DateTime.Now;

    int done = tasks.Count(t => t.Status == TaskStatus.Done);
    int inProgress = tasks.Count(t => t.Status ==
       TaskStatus.InProgress);
    int overdue = tasks.Count(t =>
        t.DueDate.HasValue &&
        t.DueDate.Value < now &&
        t.Status != TaskStatus.Done
    );

    string baseRating = GetBaseRating(done, count);
    string finalRating = ApplyModifiers(baseRating, done,
       inProgress, overdue, count);

    return $"Ucinkovitost tima: {finalRating}";
}

private static string GetBaseRating(int done, int count)
{
    if (done == 0) return "Lose";
    if (done < count / 3) return "Slabo";
    if (done < count / 2) return "Umjereno";
    if (done < count * 0.8) return "Dobro";
    return "Odlicno";
}
```

```
33
34 private static string ApplyModifiers(
35     string rating,
36     int done,
37     int inProgress,
38     int overdue,
39     int count)
40 {
41     if (overdue > done / 2)
42         return rating + " (Previse van roka)";
43
44     if (inProgress > done)
45         return rating + " (Jos dosta posla)";
46
47     if (done == count)
48         return rating + " (Sve zavrseno)";
49
50     return rating;
51 }
```

4.6 Mjerenja performansi

Mjerenje performansi izvršeno je pozivanjem svake verzije metode 1 000 000 puta nad istim skupom testnih podataka. Vrijeme izvršavanja mjereno je pomoću `Stopwatch`, dok je promjena zauzeća memorije praćena korištenjem `GC.GetTotalMemory`.

```
1     public void CompareAnalyzeTeamPerformanceVersions()
2         {
3             Console.WriteLine("Usporedba verzija metode
                 AnalyzeTeamPerformance:");
4
5             // Degradirana verzija
6             var watch = System.Diagnostics.Stopwatch.StartNew();
7             for (int i = 0; i < 1000000; i++)
                 AnalyzeTeamPerformance();
8             watch.Stop();
9             long memBefore = GC.GetTotalMemory(true);
10            AnalyzeTeamPerformance();
11            long memAfter = GC.GetTotalMemory(true);
12            Console.WriteLine($"Degradirana verzija - Vrijeme:
                 {watch.ElapsedMilliseconds} ms, Memorija:
                 {(memAfter - memBefore) / (1024.0 * 1024.0):F6}
                 MB");
13
14            // Nakon tehnike 1
15            watch = System.Diagnostics.Stopwatch.StartNew();
16            for (int i = 0; i < 1000000; i++)
                 AnalyzeTeamPerformance_T1();
17            watch.Stop();
18            memBefore = GC.GetTotalMemory(true);
19            AnalyzeTeamPerformance_T1();
```

```
20              memAfter = GC.GetTotalMemory(true);
21              Console.WriteLine($"Nakon tehnike 1 - Vrijeme:
                    {watch.ElapsedMilliseconds} ms, Memorija:
                    {(memAfter - memBefore) / (1024.0 * 1024.0):F6}
                    MB");
22
23              // Nakon tehnike 2
24              watch = System.Diagnostics.Stopwatch.StartNew();
25              for (int i = 0; i < 1000000; i++)
                    AnalyzeTeamPerformance_T2();
26              watch.Stop();
27              memBefore = GC.GetTotalMemory(true);
28              AnalyzeTeamPerformance_T2();
29              memAfter = GC.GetTotalMemory(true);
30              Console.WriteLine($"Nakon tehnike 2 - Vrijeme:
                    {watch.ElapsedMilliseconds} ms, Memorija:
                    {(memAfter - memBefore) / (1024.0 * 1024.0):F6}
                    MB");
31
32              // Nakon tehnike 3
33              watch = System.Diagnostics.Stopwatch.StartNew();
34              for (int i = 0; i < 1000000; i++)
                    AnalyzeTeamPerformance_T3();
35              watch.Stop();
36              memBefore = GC.GetTotalMemory(true);
37              AnalyzeTeamPerformance_T3();
38              memAfter = GC.GetTotalMemory(true);
39              Console.WriteLine($"Nakon tehnike 3 - Vrijeme:
                    {watch.ElapsedMilliseconds} ms, Memorija:
                    {(memAfter - memBefore) / (1024.0 * 1024.0):F6}
                    MB");
40          }
```

```
Usporedba verzija metode AnalyzeTeamPerformance:
Degradirana verzija - Vrijeme: 849 ms, Memorija: 0.000084 MB
Nakon tehnike 1 - Vrijeme: 288 ms, Memorija: 0.000084 MB
Nakon tehnike 2 - Vrijeme: 304 ms, Memorija: 0.000107 MB
Nakon tehnike 3 - Vrijeme: 325 ms, Memorija: 0.000084 MB
Pritisnite bilo koju tipku za povratak u glavni meni...
```

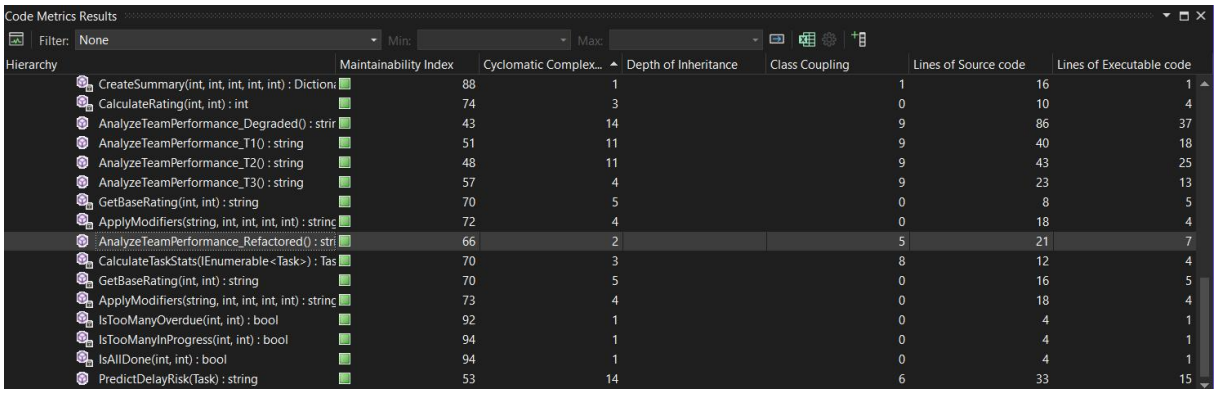
Slika 3: Prikaz rezultata

4.7 Utjecaj na McCabe metriku i indeks održavanja

Originalna verzija metode `AnalyzeTeamPerformance()` ima ciklomatičnu kompleksnost $V(G) = 11$, što ukazuje na metodu srednje složenosti. Degradirana verzija ima ciklomatičnu kompleksnost $V(G) = 14$. Indeks održavanja iznosio je $MI = 43$, što ukazuje na smanjenu čitljivost i otežano dugoročno održavanje koda.

Analiza po tehnikama

- **Tehnika 1 tuning izraza**
  - $V(G)$ smanjen sa 14 na 11
  - MI poboljšan sa 43 na 51
  - Eliminisani višestruki pozivi `Count()`, nepotrebne privremene varijable i višestruki pozivi `DateTime.Now`
  - Uklonjene bespotrebne konverzije i string konkatenacije
  - Poboljšanje MI rezultat je smanjenja broja izraza i linija koda, dok je pad ciklomatične kompleksnosti posljedica uklanjanja redundantnih uslova
- **Tehnika 2 logički iskazi**
  - $V(G)$ ostaje 11
  - MI blago opada na 48
  - Uslovi su izdvojeni u semantički jasne boolean varijable
  - Iako je čitljivost poboljšana, dodatne logičke varijable povećavaju ukupni broj linija i operacija, što negativno utiče na MI
  - Struktura grananja ostaje ista, pa ne dolazi do promjene ciklomatične kompleksnosti
- **Tehnika 3 tuning metoda**
  - $V(G)$ glavne metode smanjen sa 11 na 4
  - MI povećan na 57
  - Kompleksna logika izdvojena u pomoćne metode
  - Glavna metoda sadrži minimalan broj grananja
  - Modularizacija značajno smanjuje kompleksnost i povećava održivost koda



Hierarchy	Maintainability Index	Cyclomatic Complex...	Depth of Inheritance	Class Coupling	Lines of Source code	Lines of Executable code
CreateSummary(int, int, int, int, int) : Diction	88	1		1	16	1
CalculateRating(int, int) : int	74	3		0	10	4
AnalyzeTeamPerformance_Degraded() : strir	43	14		9	86	37
AnalyzeTeamPerformance_T1() : string	51	11		9	40	18
AnalyzeTeamPerformance_T2() : string	48	11		9	43	25
AnalyzeTeamPerformance_T3() : string	57	4		9	23	13
GetBaseRating(int, int) : string	70	5		0	8	5
ApplyModifiers(string, int, int, int, int) : string	72	4		0	18	4
AnalyzeTeamPerformance_Refactored() : stri	66	2		5	21	7
CalculateTaskStats(IEnumerable<Task>) : Tas	70	3		8	12	4
GetBaseRating(int, int) : string	70	5		0	16	5
ApplyModifiers(string, int, int, int, int) : string	73	4		0	18	4
IsTooManyOverdue(int, int) : bool	92	1		0	4	1
IsTooManyInProgress(int, int) : bool	94	1		0	4	1
IsAllDone(int, int) : bool	94	1		0	4	1
PredictDelayRisk(Task) : string	53	14		6	33	15

Slika 4: Prikaz ciklomatične kompleksnosti iz Code Metrics alata

5 Refaktorisanje

5.1 Primijenjene refaktoring tehnike

Tokom refaktorisanja korištene su sljedeće stavke iz refactoring checkliste:

- Extract Method,
- Decompose Conditional,
- Move Complex Boolean Expression.

5.2 Refaktorisani kod

```
public string AnalyzeTeamPerformance_Refactored()
{
    var tasks = repository.GetAllTasks();
    int total = tasks.Count;

    if (total == 0)
        return "Nema podataka.";

    var stats = CalculateTaskStats(tasks);

    string baseRating = GetBaseRating(stats.Done, total);
    string finalRating = ApplyModifiers(
        baseRating,
        stats.Done,
        stats.InProgress,
        stats.Overdue,
        total
    );

    return $"Ucinkovitost tima: {finalRating}";
}

private TaskStats CalculateTaskStats(IEnumerable<Task> tasks)
{
    return new TaskStats
    {
        Done = tasks.Count(t => t.Status == TaskStatus.Done),
        InProgress = tasks.Count(t => t.Status ==
           TaskStatus.InProgress),
        Overdue = tasks.Count(t =>
            t.DueDate.HasValue &&
            t.DueDate.Value < DateTime.Now &&
            t.Status != TaskStatus.Done)
    };
}

private string GetBaseRating(int done, int total)
{
```

```
38      if (done == 0)
39          return "Lose";
40
41      if (done < total / 3)
42          return "Slabo";
43
44      if (done < total / 2)
45          return "Umjereno";
46
47      if (done < total * 0.8)
48          return "Dobro";
49
50      return "Odlicno";
51  }
52
53  private string ApplyModifiers(
54      string rating,
55      int done,
56      int inProgress,
57      int overdue,
58      int total)
59  {
60      if (IsTooManyOverdue(overdue, done))
61          return rating + " (Previse van roka)";
62
63      if (IsTooManyInProgress(inProgress, done))
64          return rating + " (Jos dosta posla)";
65
66      if (IsAllDone(done, total))
67          return rating + " (Sve zavrseno)";
68
69      return rating;
70  }
71
72  private bool IsTooManyOverdue(int overdue, int done)
73  {
74      return overdue > done / 2;
75  }
76
77  private bool IsTooManyInProgress(int inProgress, int done)
78  {
79      return inProgress > done;
80  }
81
82  private bool IsAllDone(int done, int total)
83  {
84      return done == total;
85  }
86
87  private class TaskStats
88  {
```

```
89      public int Done { get; set; }
90      public int InProgress { get; set; }
91      public int Overdue { get; set; }
92  }
```

5.3 Opis primijenjenih refaktoring tehnika

Refaktorisanje metode `AnalyzeTeamPerformance()` provedeno je sa ciljem smanjenja ciklomatične kompleksnosti, povećanja čitljivosti i poboljšanja indeksa održavanja, uz očuvanje postojećeg ponašanja metode.

Extract Method Logika za izračun statistike zadataka izdvojena je u metodu `CalculateTaskStats()`, dok su logički uslovi za dodavanje opisnih sufiksa izdvojeni u metode `IsTooManyOverdue()`, `IsTooManyInProgress()` i `IsAllDone()`. Na ovaj način smanjen je broj odluka u glavnoj metodi.

Decompose Conditional Složeni niz `if-else if` izraza za određivanje završne ocjene razložen je u dvije faze: osnovna ocjena (`GetBaseRating`) i dodatni modifikatori (`ApplyModifiers`). Time je poboljšana čitljivost i lakše je razumjeti poslovnu logiku.

Move Complex Boolean Expression Kompleksni logički izrazi prebačeni su u zasebne metode sa deskriptivnim imenima, čime je uklonjena potreba za razumijevanjem detalja uslova prilikom čitanja glavne metode.

5.4 Uticaj refaktorisanja na metrike

Na slici 4. iz poglavlja Code Tuning je prikazano stanje iz Code Metrics alata.

Verzija	McCabe	Maintainability Index
Pogoršana	14	43
Nakon tuninga	4	57
Nakon refaktorisanja	2	66

Tablica 8: Uticaj refaktorisanja na metrike

Nakon code tuning-a izvršen je refaktoring sa ciljem postizanja što boljih vrijednosti metrika. Svaka promjena može se direktno povezati sa stavkama unutar refaktoring checklist-e i njihovim utjecajem na Code Metrics.

Checklist stavka	**Primjena**	**Utjecaj na metrike**
Eliminacija redundantnih uslova	T1	Smanjen $V(G)$
Keširanje vrijednosti	T1	Povećan MI
Uklanjanje nepotrebnih izraza	T1	Manji LOC, veći MI
Izdvajanje logike u booleane	T2	Veća čitljivost, blagi pad MI
Modularizacija	T3	Drastično smanjen $V(G)$
Single Responsibility Principle	T3	Značajan rast MI
Izdvajanje pomoćnih metoda	T3	Ravnomjerna raspodjela kompleksnosti

Zaključak

Rezultati pokazuju da tuning izraza donosi najveći balans između performansi i održivosti, dok tuning logičkih iskaza prvenstveno poboljšava čitljivost. Najveći pozitivni utjecaj na Code Metrics postiže se primjenom refaktoringa kroz modularizaciju, gdje dolazi do značajnog smanjenja ciklomatične kompleksnosti i povećanja indeksa održavanja.

Verifikacija i Validacija Softvera

Projektni zadatak – II dio

TaskManagerApp – analiza jedne metode

Student: Mirnes Fehrić

Broj indeksa: 19733

Odabrana metoda: `GetTasksSummaryForUser(int userId)`

Sarajevo, 2025.

1. McCabe metrika i kontrolni graf

1.1 Odabir metode

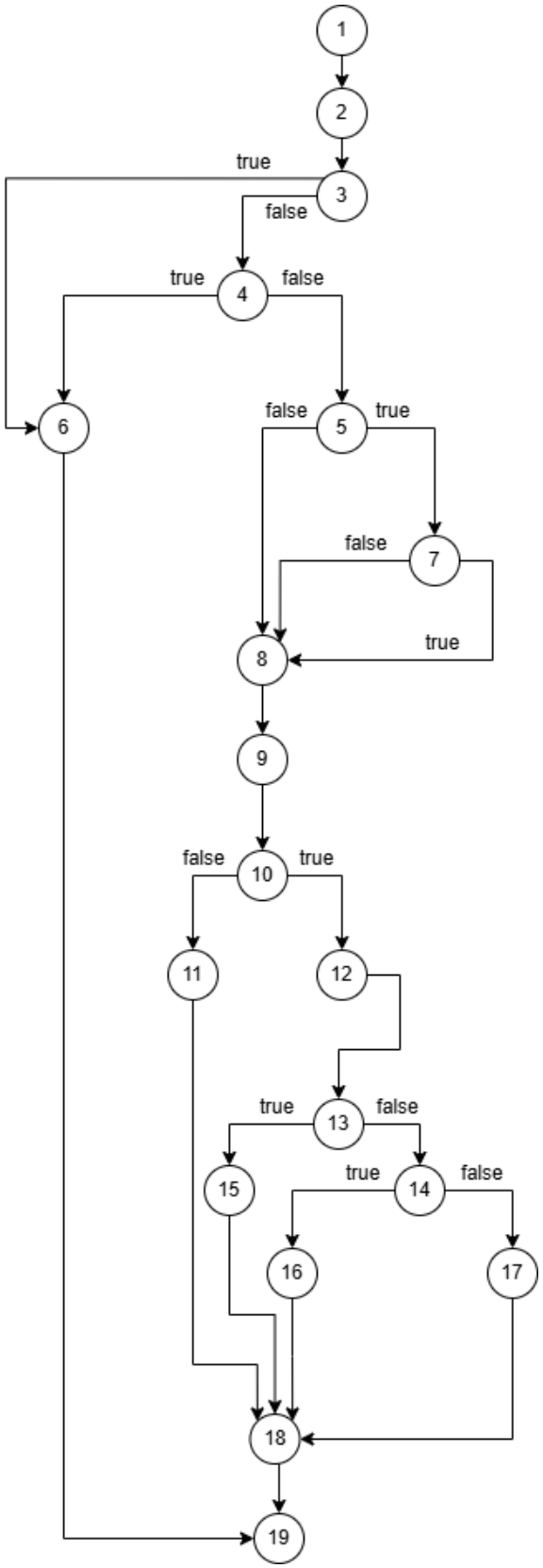
Za analizu je odabrana metoda `GetTasksSummaryForUser(int userId)` iz klase `TaskAnalyzer`, jer prema *Visual Studio Metrics* alatu ima ciklomatsku kompleksnost $\mathbf{V(G)=8}$, čime zadovoljava zahtjev projektnog zadatka ($V(G)\geq 7$).

1.2 Izvorni kod metode

```
public Dictionary<string, int> GetTasksSummaryForUser(int userId)
{
    var tasks = repository.GetTasksByUser(userId);
    if (tasks == null || tasks.Count == 0)
        return new Dictionary<string, int> { { "Nema zadataka", 0 } };

    var summary = new Dictionary<string, int>
    {
        { "ToDo", tasks.Count(t => t.Status == TaskStatus.ToDo) },
        { "InProgress", tasks.Count(t => t.Status == TaskStatus.InProgress)
    },
        { "Testing", tasks.Count(t => t.Status == TaskStatus.Testing) },
        { "Done", tasks.Count(t => t.Status == TaskStatus.Done) },
        { "Overdue", tasks.Count(t =>
            t.DueDate.HasValue &&
            t.DueDate.Value < DateTime.Now &&
            t.Status != TaskStatus.Done) }
    };

    int total = summary.Values.Sum();
    if (total > 0)
    {
        double doneRatio = (double)summary["Done"] / total;
        if (doneRatio < 0.3)
            summary.Add("Ocjena", 1);
        else if (doneRatio < 0.6)
            summary.Add("Ocjena", 2);
        else
            summary.Add("Ocjena", 3);
    }

    return summary;
}
```

1.3 Kontrolni graf

Čvorovi predstavljaju blokove iskaza, a ivice predstavljaju tok kontrole između čvorova.



Slika 1: Kontrolni graf za metodu `GetTasksSummaryForUser(int userId)`.

1.4 Ručni izračun McCabe metrike (ciklomatske kompleksnosti)

Ciklomatska kompleksnost (McCabe) predstavlja broj linearno nezavisnih puteva kroz metodu. U nastavku je prikazan proračun prema dvije ekvivalentne formule.

Proračun prema formuli $V(G) = E - N + 2$

Za nacrtani kontrolni graf važi:

- $N = 19$ (broj čvorova),
- $E = 25$ (broj ivica).

Po McCabe definiciji:

$$V(G) = E - N + 2 = 25 - 19 + 2 = 8$$

Proračun prema formuli $V(G) = 1 + D$

Alternativno, ciklomatska kompleksnost se može izračunati kao:

$$V(G) = 1 + D$$

gdje je D broj tačaka odluke.

U metodi `GetTasksSummaryForUser` tačke odluke su:

- `if (tasks == null || tasks.Count == 0)` $\Rightarrow 2$ (složeni uslov sa ||)
- `if (total > 0)` $\Rightarrow 1$
- `if (doneRatio < 0.3)` $\Rightarrow 1$
- `else if (doneRatio < 0.6)` $\Rightarrow 1$
- LINQ predikat za Overdue: `t.DueDate.HasValue && t.DueDate.Value < DateTime.Now && t.Status != TaskStatus.Done` $\Rightarrow 2$ (implicitne odluke)

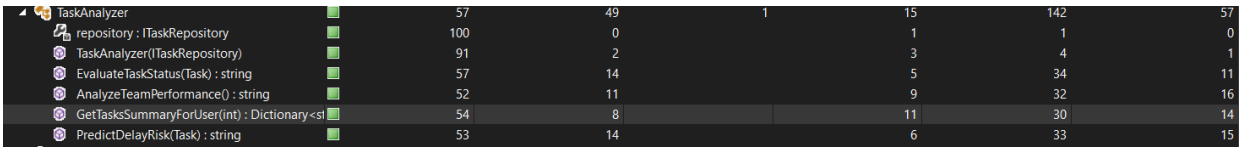
Ukupno:

$$D = 2 + 1 + 1 + 1 + 2 = 7$$

$$V(G) = 1 + 7 = 8$$

1.5 Poređenje sa Code Metrics alatom

Ručni izračun daje **V(G)=8**, što se poklapa sa vrijednošću **Cyclomatic Complexity = 8** dobijenom iz *Visual Studio Metrics* rezultata za metodu `GetTasksSummaryForUser(int userId)`.



Slika 2: Grafički prikaz *Visual Studio Metrics* metode `GetTasksSummaryForUser`.

1.6 Kratka diskusija

Vrijednost **V(G)=8** ukazuje na srednje složen kontrolni tok. Metoda sadrži više uslovnih izraza i grananja, uključujući složene logičke uslove i više povratnih tačaka. Kod je funkcionalno ispravan i relativno čitljiv, ali sadrži ponovljene LINQ izraze i višestruke prolaze kroz kolekciju zadataka, što može negativno uticati na performanse pri velikom broju poziva.

Moguća poboljšanja uključuju:

- Grupisanje logike u manje pomoćne metode (Refactoring – Extract Method)
- Smanjenje broja prolaza kroz kolekciju zadataka
- Pojednostavljenje složenih logičkih izraza

U narednim tačkama izvještaja biće prikazan dizajn testnih slučajeva, mjerenja nakon code tuning-a i refaktoring promjene usmjerene na poboljšanje održivosti.

2. White Box testiranje

Za metodu `GetTasksSummaryForUser(int userId)` razmatrane su sljedeće white box tehnike testiranja:

- Obuhvat iskaza (statement/line coverage)
- Obuhvat grana (branch/decision coverage)
- Obuhvat uslova (condition coverage)
- Modifikovani uslov/odluka obuhvat (MCDC)
- Obuhvat petlji (loop coverage)
- Obuhvat toka podataka (data-flow coverage)
- Obuhvat puteva (path coverage)

2.1 Definicija testnih podataka

U nastavku su definirani konkretni ulazni podaci koji se koriste u svim tehnikama testiranja:

Test	Ulazni podaci
T1	`userId = 99` (korisnik ne postoji, `repository.GetTasksByUser(99)` vraća `null`)
T2	`userId = 2` (korisnik postoji, ali nema zadataka – prazna lista)
T3	`userId = 1`, lista zadataka: svi sa `Status = Cancelled` (status koji se ne broji, pa je `total = 0`)
T4	`userId = 1`, 5 zadataka: 1× Done, 2× ToDo, 1× InProgress, 1× Testing (Done=1, total=5, doneRatio=0.2 < 0.3 → Ocjena=1)
T5	`userId = 1`, 4 zadatka: 2× Done, 1× ToDo, 1× InProgress (Done=2, total=4, doneRatio=0.5 → Ocjena=2)
T6	`userId = 1`, 4 zadatka: 3× Done, 1× ToDo (Done=3, total=4, doneRatio=0.75 → Ocjena=3)
T7	`userId = 1`, zadatak sa: `DueDate = DateTime.Now.AddDays(-5)`, `Status = InProgress` (rok istekao, nije završen → Overdue = 1)
T8	`userId = 1`, 3 zadatka: 1× ToDo, 1× InProgress, 1× Testing (svi sa `DueDate = DateTime.Now.AddDays(+10)` – rok nije istekao)
T9	`userId = 1`, zadatak sa: `DueDate = null`, `Status = ToDo` (nema definisan rok → ne računa se kao Overdue)
T10	Isto kao T1 (`tasks = null`) – za MCDC
T11	Isto kao T2 (`tasks = prazna lista`) – za MCDC

Tablica 1: Konkretni ulazni podaci za sve testne slučajeve

2.2 Tehnika 1: Obuhvat iskaza

ID	Opis testa	Ulaz	Očekivani izlaz
T1	Null task lista – provjera prvog if uslova	tasks = null	{ "Nema zadataka": 0 }
T2	Prazna lista zadataka	tasks = prazna lista	{ "Nema zadataka": 0 }
T3	Svi statusi nula (total = 0)	Done=0, ostali=0	Nema ključa "Ocjena"
T4	Nizak doneRatio (<0.3)	Done=1, total=5	Ocjena = 1
T5	Srednji doneRatio (0.3-0.6)	Done=2, total=4	Ocjena = 2
T6	Visok doneRatio (>=0.6)	Done=3, total=4	Ocjena = 3
T7	Overdue zadaci postoje	DueDate < Now, Status ≠ Done	Overdue > 0
T8	Više različitih statusa	ToDo, InProgress, Testing	Ispravni brojači
T9	Zadatak bez roka	DueDate = null	Overdue = 0

Tablica 2: Test slučajevi – Tehnika 1 (obuhvat iskaza)

Status izvršavanja: Svi testovi prošli.
Nakon izvršavanja ovih testova, svaka izvodiva linija metode bila je pokrivena najmanje jednom:

- T1 pokriva `if (tasks == null)` granu
- T2 pokriva `tasks.Count == 0` granu
- T3 pokriva slučaj kada je `total = 0`
- T4 pokriva granu `doneRatio < 0.3`
- T5 pokriva granu `doneRatio < 0.6`
- T6 pokriva else granu (doneRatio >= 0.6)

- T7 pokriva LINQ predikat za Overdue
- T8 pokriva sve LINQ Count izraze za statuse
- T9 pokriva slučaj kada `!DueDate.HasValue`

Time je ostvaren **100% statement/line coverage**.

2.3 Tehnika 2: Obuhvat grana/odluka

Kod obuhvata grana cilj je da svaka odluka (if, else if) ima barem po jedno izvođenje gdje je uslov true i barem jedno gdje je false.
Glavne odluke u metodi su:

- D1: `tasks == null || tasks.Count == 0`
- D2: `total > 0`
- D3: `doneRatio < 0.3`
- D4: `doneRatio < 0.6`
- D5: `t.DueDate.HasValue && t.DueDate.Value < Now && t.Status != Done`

Test	**Pokrivene grane**	**Napomena**
T1	D1 = true	Ulaz tasks = null
T2	D1 = true (drugi dio)	Lista postoji ali je prazna
T3	D1 = false, D2 = false	Lista ima elemente, ali total = 0
T4	D2 = true, D3 = true	doneRatio < 0.3, Ocjena = 1
T5	D3 = false, D4 = true	$0.3 \leq$ doneRatio < 0.6, Ocjena = 2
T6	D3 = false, D4 = false	doneRatio ≥ 0.6, Ocjena = 3
T7	D5 = true	Postoje overdue zadaci
T8	D5 = false (djelimično)	Zadaci nisu overdue
T9	D5 = false (HasValue)	Zadatak nema definisan rok

Tablica 3: Test slučajevi – Tehnika 2 (obuhvat grana)

Kombinacijom navedenih testova postignuto je da svaka grana svake odluke bude izvršena barem jednom (true/false), čime je ostvaren **100% branch/decision coverage**.

2.4 Tehnika 3: Obuhvat uslova

Obuhvat uslova posmatra elementarne logičke izraze u složenim uslovima. Za metodu `GetTasksSummaryForU` ključni uslovi su:

- C1: `tasks == null`
- C2: `tasks.Count == 0`
- C3: `total > 0`
- C4: `doneRatio < 0.3`
- C5: `doneRatio < 0.6`

- C6: `t.DueDate.HasValue`
- C7: `t.DueDate.Value < DateTime.Now`
- C8: `t.Status != TaskStatus.Done`

Uslov	**TRUE primjeri**	**FALSE primjeri**
C1 (tasks == null)	T1	T2–T9
C2 (tasks.Count == 0)	T2	T3–T9
C3 (total > 0)	T4–T9	T3
C4 (doneRatio < 0.3)	T4	T5, T6
C5 (doneRatio < 0.6)	T5	T6
C6 (DueDate.HasValue)	T7	T9
C7 (DueDate.Value < Now)	T7	T8
C8 (Status != Done)	T7, T8	T6 (Done zadaci)

Tablica 4: Testovi – Tehnika 3 (obuhvat uslova)

Na osnovu tabele se vidi da svaki elementarni uslov poprima obje logičke vrijednosti u nekom od definisanih testova, čime je ostvaren potpuni **obuhvat uslova (condition coverage)**.

2.5 Tehnika 4: Modifikovani uslov/odluka obuhvat (MC/DC)

MC/DC zahtijeva da se pokaže da promjena jednog elementarnog uslova može nezavisno promijeniti ishod odluke.
Složene odluke u metodi su:

- D1: `tasks == null || tasks.Count == 0`
- D5: `t.DueDate.HasValue && t.DueDate.Value < Now && t.Status != Done`

Napomena za D1: Zbog kratko-spojnog operatora `||`, komponenta `tasks.Count == 0` se evaluira samo kada je `tasks == null` netačno.

ID	**Opis testa**	**Svrha**
T10	tasks = null	Komponenta `tasks == null` = true u D1
T11	tasks = prazna lista	Komponenta `tasks.Count == 0` = true uz `tasks != null`

Tablica 5: Dodatni testovi za MCDC (T10–T11)

Komponenta	**Par testova**	**Ishod odluke**
D1: `tasks == null`	T10 (true) vs. T11 (false)	D1: true → true (ali različit put)
D1: `tasks.Count == 0`	T11 (true) vs. T3 (false)	D1: true → false
D2: `total > 0`	T4 (true) vs. T3 (false)	D2: true → false
D5: `DueDate.HasValue`	T7 (true) vs. T9 (false)	D5: true → false
D5: `DueDate.Value < Now`	T7 (true) vs. T8 (false)	D5: true → false

Tablica 6: MCDC parovi za složene odluke

Na osnovu navedenih parova, **MC/DC je zadovoljen za sve dostižne odluke** u metodi.

2.6 Tehnika 5: Obuhvat petlji

Obuhvat petlji (loop coverage) podrazumijeva testiranje prolaza kroz petlje (`for`, `while`, `foreach`) u slučajevima: 0 iteracija, 1 iteracija i više iteracija.
Metoda `GetTasksSummaryForUser` ne sadrži eksplicitne petlje. Iako se koriste LINQ izrazi poput `Count()`, iteracije su apstrahovane i ne predstavljaju eksplicitne petlje za white-box analizu. Zbog toga je ova tehnika **neprimjenjiva (N/A)** i smatra se trivijalno zadovoljenom.

2.7 Tehnika 6: Obuhvat toka podataka (Data-flow coverage)

Data-flow obuhvat posmatra parove definicija i upotreba podataka (def-use), tj. da li se vrijednosti atributa koriste kroz sve relevantne tokove.
U metodi su najvažniji atributi: `tasks`, `summary`, `total` i `doneRatio`.

Podatak	**Mjesta upotrebe (use)**	**Testovi koji pokrivaju**
`tasks`	Provjera `null/Count`, LINQ Count izrazi	T1 (null), T2 (prazno), T3–T9 (elementi)
`summary`	Kreiranje, dodavanje Ocjene, vraćanje	T3–T9
`total`	Uslov `total > 0`, izračun doneRatio	T3 (false), T4–T9 (true)
`doneRatio`	Grananje za dodjelu ocjene	T4 (Ocjena 1), T5 (Ocjena 2), T6 (Ocjena 3)

Tablica 7: Data-flow: ključne def-use upotrebe i testovi

Detaljni def-use lanci za `total` i `doneRatio`:

- T3: total = 0 → provjera (total > 0 = false) → preskače se blok ocjene
- T4: total = 5 → provjera (true) → doneRatio = 0.2 → Ocjena = 1
- T5: total = 4 → provjera (true) → doneRatio = 0.5 → Ocjena = 2
- T6: total = 4 → provjera (true) → doneRatio = 0.75 → Ocjena = 3

Time je osigurano da se sve relevantne vrijednosti atributa koriste kroz sve bitne tokove metode, čime je ostvaren **100% data-flow coverage**.

2.8 Tehnika 7: Obuhvat puteva (Path coverage)

Path coverage podrazumijeva pokrivanje svih izvedivih logičkih puteva kroz grananje. Za ovu metodu, izvedivi putevi odgovaraju glavnim povratnim tačkama:

- P1: `tasks == null` → vraća { "Nema zadataka": 0 } (T1)
- P2: `tasks.Count == 0` → vraća { "Nema zadataka": 0 } (T2)
- P3: `total == 0` → vraća summary bez Ocjene (T3)
- P4: `doneRatio < 0.3` → Ocjena = 1 (T4)
- P5: `0.3` $\leq$ `doneRatio < 0.6` → Ocjena = 2 (T5)
- P6: `doneRatio` $\geq$ `0.6` → Ocjena = 3 (T6)
- P7: Overdue zadaci postoje → Overdue > 0 (T7)

S obzirom da svaki od ovih izvedivih puteva ima barem jedan test, postignut je obuhvat svih izvedivih puteva kroz grananje metode.

2.9 Zaključak za white box testiranje

Korištenjem skupa testnih slučajeva T1–T11 postignuti su:

- **100% obuhvat iskaza/linija** (statement/line coverage)
- **100% obuhvat grana/odluka** (branch/decision coverage)
- **100% obuhvat uslova** (condition coverage)
- **100% MC/DC** za dostižne složene odluke
- **Loop coverage: N/A** (metoda ne sadrži eksplicitne petlje)
- **100% obuhvat toka podataka** nad ključnim varijablama
- **100% path coverage** (pokriveni ključni izvedivi putevi)

Ovo potvrđuje da je logika metode detaljno pokrivena sa aspekta white box testiranja i da su obrađeni svi značajni scenariji ponašanja metode.

3. Code tuning

U ovom poglavlju nad metodom `GetTasksSummaryForUser(int userId)` primijenjene su tri različite tehnike code tuninga, u skladu sa kategorijama iz predavanja (izrazi, logički iskazi i metode).
Cilj je bio:

- smanjiti vrijeme izvršavanja metode pri velikom broju poziva,
- zadržati isto funkcionalno ponašanje,
- poboljšati strukturu i održivost koda (McCabe i MI).

3.1 Pristup demonstraciji

Kako je originalna metoda već relativno dobro napisana, **za potrebe demonstracije code tuning tehnika kreirana je umjetno degradirana verzija metode** sa tipičnim problemima performansi koji se često javljaju u praksi:

- Višestruki pozivi repozitorija (3 puta umjesto 1)
- Nepotrebne string konverzije (`ToString()` za enum poređenje)
- `DateTime.Now` poziva se unutar LINQ izraza (skup sistemski poziv)
- Redundantni logički uslovi
- Nepotrebne konverzije tipova (int → string → double)
- Nedostižne else grane

3.2 Korištene tehnike optimizacije

Primijenjene su sljedeće tehnike:

- **Tehnika 1 – tuning izraza:** eliminacija višestrukih poziva repozitorija i uklanjanje nepotrebnih `ToString()` konverzija. Enum vrijednosti se porede direktno.
- **Tehnika 2 – tuning logičkih iskaza:** primjena *Loop Invariant Code Motion* – izvlačenje `DateTime.Now` izvan LINQ izraza, eliminacija redundantnih logičkih provjera u if-else lancu.
- **Tehnika 3 – tuning metoda:** primjena principa Extract Method – izdvajanje logike za `IsOverdue` i `CalculateRating` u pomoćne metode.

3.3 Izmijenjeni kod

3.3.1 Degradirana verzija (sa namjernim problemima)

```
public Dictionary<string, int> GetTasksSummaryForUser_Degraded(int userId)
{
    // Problem 1: Nepotrebna privremena varijabla
    var currentUserId = userId;

    // Problem 2: Viseslojni pristup repository-u (3 puta!)
    if (repository.GetTasksByUser(currentUserId) == null ||
        repository.GetTasksByUser(currentUserId).Count == 0)
    {
        // Problem 3: Nepotrebna string konkatenacija
        var emptyKey = "Nema " + "zadataka";
        return new Dictionary<string, int> { { emptyKey, 0 } };
    }

    var tasks = repository.GetTasksByUser(currentUserId);
    var summary = new Dictionary<string, int>();

    // Problem 4: Nepotrebne ToString() konverzije
    summary.Add("ToDo", tasks.Count(t =>
        t.Status.ToString() == TaskStatus.ToDo.ToString()));
    summary.Add("InProgress", tasks.Count(t =>
        t.Status.ToString() == TaskStatus.InProgress.ToString()));
    summary.Add("Testing", tasks.Count(t =>
        t.Status.ToString() == TaskStatus.Testing.ToString()));
    summary.Add("Done", tasks.Count(t =>
        t.Status.ToString() == TaskStatus.Done.ToString()));

    // Problem 5: DateTime.Now unutar LINQ (poziva se za svaki element!)
    summary.Add("Overdue", tasks.Count(t =>
        t.DueDate.HasValue &&
        t.DueDate.Value < DateTime.Now &&
        t.Status != TaskStatus.Done));

    // Problem 6: Nepotrebna foreach petlja umjesto LINQ Sum()
```

```
    int total = 0;
    foreach (var val in summary.Values)
        total = total + val;

    if (total > 0)
    {
        // Problem 7: Nepotrebne konverzije int -> string -> double
        var doneCountString = summary["Done"].ToString();
        double doneCount = double.Parse(doneCountString);
        double totalDouble = double.Parse(total.ToString());
        double doneRatio = doneCount / totalDouble;

        // Problem 8: Redundantni uslovi
        if (doneRatio < 0.3)
            summary.Add("Ocjena", 1);
        else if (doneRatio >= 0.3 && doneRatio < 0.6) // >= 0.3 redundantno
            summary.Add("Ocjena", 2);
        else if (doneRatio >= 0.6)
            summary.Add("Ocjena", 3);
        else
            summary.Add("Ocjena", 0); // Problem 9: Nedostizna grana
    }
    else
    {
        // Problem 10: Redundantan else blok
        total = total + 0;
    }

    return summary;
}
```

3.3.2 Nakon tehnike 1 – tuning izraza (T1)

Primijenjene optimizacije:

- Eliminacija višestrukih poziva repozitorija ($3 \rightarrow 1$)
- Uklanjanje `ToString()` konverzija – direktno poređenje enum vrijednosti
- Eliminacija nepotrebne string konkatenacije
- Zamjena foreach petlje sa LINQ `Sum()`

```
public Dictionary<string, int> GetTasksSummaryForUser_T1(int userId)
{
    // Fix: Samo jedan poziv repozitorija
    var tasks = repository.GetTasksByUser(userId);

    if (tasks == null || tasks.Count == 0)
        return new Dictionary<string, int> { { "Nema zadataka", 0 } };
```

```
    // Fix: Direktno enum poredenje (bez ToString())
    var summary = new Dictionary<string, int>
    {
        { "ToDo", tasks.Count(t => t.Status == TaskStatus.ToDo) },
        { "InProgress", tasks.Count(t => t.Status == TaskStatus.InProgress)
    },
        { "Testing", tasks.Count(t => t.Status == TaskStatus.Testing) },
        { "Done", tasks.Count(t => t.Status == TaskStatus.Done) },
        { "Overdue", tasks.Count(t =>
            t.DueDate.HasValue &&
            t.DueDate.Value < DateTime.Now &&
            t.Status != TaskStatus.Done) }
    };

    // Fix: LINQ Sum() umjesto foreach
    int total = summary.Values.Sum();

    if (total > 0)
    {
        double doneRatio = (double)summary["Done"] / total;

        if (doneRatio < 0.3) summary.Add("Ocjena", 1);
        else if (doneRatio >= 0.3 && doneRatio < 0.6) summary.Add("Ocjena",
    2);
        else if (doneRatio >= 0.6) summary.Add("Ocjena", 3);
        else summary.Add("Ocjena", 0);
    }

    return summary;
}
```

3.3.3 Nakon tehnike 2 – logički iskazi (T2)

Primijenjene optimizacije:

- Loop Invariant Code Motion – `DateTime.Now` izvučen van LINQ izraza
- Eliminacija redundantnih logičkih provjera (`doneRatio >= 0.3` nakon else)
- Uklanjanje nedostižne else grane

```
public Dictionary<string, int> GetTasksSummaryForUser_T2(int userId)
{
    var tasks = repository.GetTasksByUser(userId);
    if (tasks == null || tasks.Count == 0)
        return new Dictionary<string, int> { { "Nema zadataka", 0 } };

    // Fix: Kesiranje DateTime.Now (Loop Invariant Code Motion)
    var now = DateTime.Now;

    var summary = new Dictionary<string, int>
```

```
    {
        { "ToDo", tasks.Count(t => t.Status == TaskStatus.ToDo) },
        { "InProgress", tasks.Count(t => t.Status == TaskStatus.InProgress)
    },
        { "Testing", tasks.Count(t => t.Status == TaskStatus.Testing) },
        { "Done", tasks.Count(t => t.Status == TaskStatus.Done) },
        { "Overdue", tasks.Count(t =>
            t.DueDate.HasValue && t.DueDate.Value < now &&
            t.Status != TaskStatus.Done) }
    };

    int total = summary.Values.Sum();
    if (total > 0)
    {
        double doneRatio = (double)summary["Done"] / total;

        // Fix: Pojednostavljena logika (bez redundantnih provjera)
        if (doneRatio < 0.3) summary.Add("Ocjena", 1);
        else if (doneRatio < 0.6) summary.Add("Ocjena", 2);
        else summary.Add("Ocjena", 3);
    }
    return summary;
}
```

3.3.4 Nakon tehnike 3 – metode (T3)

Primijenjene optimizacije:

- Extract Method – izdvajanje `IsOverdue` logike
- Extract Method – izdvajanje `CalculateRating` logike
- Glavna metoda postaje linearan niz poziva sa minimalnim grananjem

```
public Dictionary<string, int> GetTasksSummaryForUser_T3(int userId)
{
    var tasks = repository.GetTasksByUser(userId);
    if (tasks == null || tasks.Count == 0)
        return new Dictionary<string, int> { { "Nema zadataka", 0 } };

    var now = DateTime.Now;
    var summary = new Dictionary<string, int>
    {
        { "ToDo", tasks.Count(t => t.Status == TaskStatus.ToDo) },
        { "InProgress", tasks.Count(t => t.Status == TaskStatus.InProgress)
    },
        { "Testing", tasks.Count(t => t.Status == TaskStatus.Testing) },
        { "Done", tasks.Count(t => t.Status == TaskStatus.Done) },
        { "Overdue", tasks.Count(t => IsOverdue(t, now)) }
    };
```

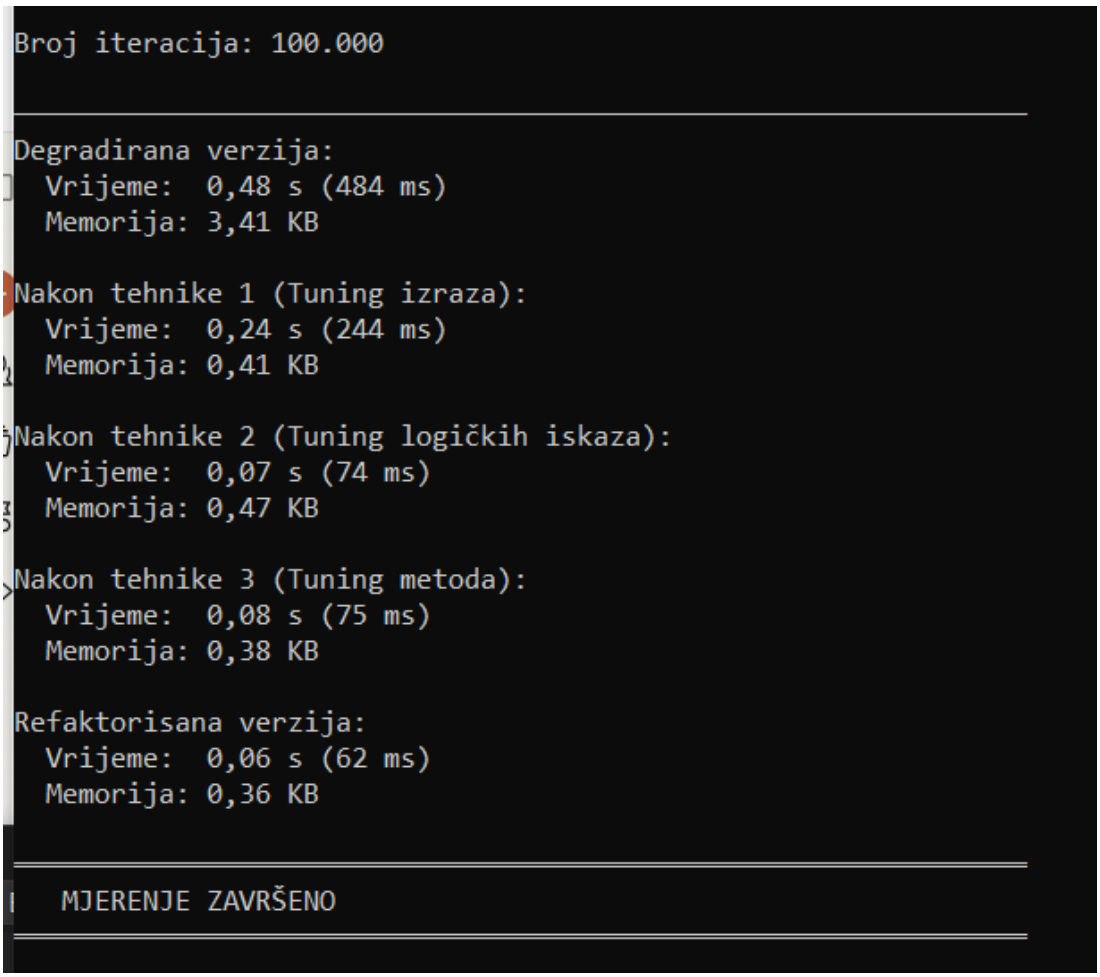
```
    int total = summary.Values.Sum();
    if (total > 0)
        summary.Add("Ocjena", CalculateRating(summary["Done"], total));

    return summary;
}

private bool IsOverdue(Task t, DateTime now) =>
    t.DueDate.HasValue && t.DueDate.Value < now && t.Status != TaskStatus.
   Done;

private int CalculateRating(int done, int total)
{
    double ratio = (double)done / total;
    if (ratio < 0.3) return 1;
    if (ratio < 0.6) return 2;
    return 3;
}
```

3.4 Mjerenja performansi

Mjerenje performansi je izvršeno tako što je svaka verzija metode pozvana velikim brojem puta (100.000 iteracija) nad skupom testnih zadataka.
Za svaku verziju mjereno je ukupno vrijeme izvršavanja i razlika u zauzeću memorije pomoću `Stopwatch` i `GC.GetTotalMemory`.

Verzija	**Vrijeme (ms)**	**Memorija (KB)**	**Poboljšanje**
Degradirana metoda	484	3.41	–
Nakon tehnike 1 (T1)	244	0.41	49.6% (vrijeme)
Nakon tehnike 2 (T2)	74	0.47	84.7% (vrijeme)
Nakon tehnike 3 (T3)	75	0.38	84.5% (vrijeme)
Refaktorisana verzija	62	0.36	**87.2%** (vrijeme)

Tablica 8: Vrijeme izvršavanja i zauzeće memorije za 100.000 poziva metode



Slika 3: Grafički prikaz performansi različitih verzija metode

Analiza rezultata:

- **Degradirana verzija:** Najsporija sa 484 ms i najvećom memorijskom potrošnjom (3.41 KB). Višestruki pozivi repozitorija i `ToString()` konverzije uzrokuju značajan overhead.
- **Tehnika 1 (Tuning izraza):** Vrijeme prepolovljeno na 244 ms (**49.6% poboljšanje**). Memorija drastično smanjena sa 3.41 KB na 0.41 KB (**88% smanjenje**). Ovo je rezultat eliminacije višestrukih poziva repozitorija i uklanjanja nepotrebnih string konverzija.
- **Tehnika 2 (Tuning logičkih iskaza):** Dramatično poboljšanje – vrijeme smanjeno na 74 ms (**84.7% brže od degradirane**). Keširanje `DateTime.Now` i pojednostavljenje logičkih uslova donosi značajne uštede.
- **Tehnika 3 (Tuning metoda):** Slične performanse kao T2 (75 ms), ali sa boljom strukturom koda. Extract Method pristup održava performanse uz poboljšanu čitljivost.
- **Refaktorisana verzija:** Najbolje performanse – 62 ms (**87.2% brže od degradirane**) i najmanja memorijska potrošnja (0.36 KB). Zamjena 5 LINQ `Count()` poziva jednom `foreach` petljom rezultira optimalnim O(n) algoritmom.

Ključni zaključci:

- Ukupno poboljšanje vremena: sa 484 ms na 62 ms (**7.8x brže**)
- Ukupno smanjenje memorije: sa 3.41 KB na 0.36 KB (**9.5x manje**)

- Svaka tehnika donosi inkrementalno poboljšanje, a refaktorisana verzija kombinuje sve optimizacije

3.5 Utjecaj na McCabe metriku i indeks održavanja

Ciklomatična kompleksnost degradirane metode bila je V(G) = 11, što predstavlja visoku složenost.

Verzija	**V(G)**	**MI**
Degradirana metoda	11	44
Nakon tehnike 1 (T1)	10	53
Nakon tehnike 2 (T2)	8	53
Nakon tehnike 3 (T3)	4	56

Tablica 9: Metrike za različite verzije metode

Analiza po tehnikama:

- **Tehnika 1:** V(G) smanjen sa 11 na 10, MI poboljšan sa 44 na 53. Prelazak iz kritične zone (crvene) u umjereno održivu (žutu).
- **Tehnika 2:** V(G) smanjen na 8 eliminacijom redundantnih logičkih provjera. MI ostaje 53.
- **Tehnika 3: Drastično smanjenje V(G) sa 11 na 4** – smanjenje za 63.6%. MI poboljšan na 56. Glavna metoda ima minimalno grananje.

Metrike pomoćnih metoda (T3 verzija):

Metoda	**V(G)**	**MI**
`IsOverdue`	3	87
`CalculateRating`	3	74

Tablica 10: Metrike pomoćnih metoda nakon code tuninga

4. Refaktoring nakon code tuning-a

4.1 Cilj refaktoringa

Nakon provedenog code tuning-a, cilj refaktoringa je bio:

- riješiti fundamentalni problem performansi – višestruko iteriranje kroz listu (5 LINQ `Count()` poziva)
- postići bolje vrijednosti metrika u *Code Metrics*
- zadržati identično funkcionalno ponašanje
- povećati čitljivost i održivost kroz jasnu raspodjelu odgovornosti

4.2 Mapiranje promjena na checklistu za refaktoring

Refaktoring promjena	**Checklist stavka**
Zamjena 5 LINQ `Count()` poziva jednom `foreach` petljom	**Data Level:** Substitute algorithm
Izdvajanje logike za izračun ocjene u `CalculateRating()` metodu	**Routine Level:** Extract a routine
Izdvajanje brojanja statusa u `CountStatus()` metodu	**Routine Level:** Extract a routine
Izdvajanje kreiranja rječnika u `CreateSummary()` metodu	**Routine Level:** Extract a routine
Složeni uslov za overdue izdvojen u `IsOverdue()` funkciju	**Statement Level:** Move complex boolean expression into well-named boolean function
Provjera null/prazno na početku, ostatak logike bez ugniježđavanja	**Statement Level:** Decompose a boolean expression

Tablica 11: Mapiranje refaktoring promjena na checklistu

4.3 Refaktorisana verzija

```
public Dictionary<string, int> GetTasksSummaryForUser_Ref(int userId)
{
    var tasks = repository.GetTasksByUser(userId);
    if (tasks == null || tasks.Count == 0)
        return new Dictionary<string, int> { { "Nema zadataka", 0 } };

    int todo = 0, inProgress = 0, testing = 0, done = 0, overdue = 0;
    var now = DateTime.Now;

    // KLJUCNA OPTIMIZACIJA: Jedan prolaz umjesto pet LINQ Count() poziva
    foreach (var task in tasks)
    {
        CountStatus(task.Status,
            ref todo, ref inProgress, ref testing, ref done);

        if (IsOverdue(task, now))
            overdue++;
    }

    var summary = CreateSummary(todo, inProgress, testing, done, overdue);

    int total = todo + inProgress + testing + done;
    if (total > 0)
        summary.Add("Ocjena", CalculateRating(done, total));

    return summary;
}

private static void CountStatus(TaskStatus status,
```

```
    ref int todo, ref int inProgress, ref int testing, ref int done)
{
    switch (status)
    {
        case TaskStatus.ToDo: todo++; break;
        case TaskStatus.InProgress: inProgress++; break;
        case TaskStatus.Testing: testing++; break;
        case TaskStatus.Done: done++; break;
    }
}

private static bool IsOverdue(Task task, DateTime now)
{
    return task.DueDate.HasValue &&
           task.DueDate.Value < now &&
           task.Status != TaskStatus.Done;
}

private static Dictionary<string, int> CreateSummary(
    int todo, int inProgress, int testing, int done, int overdue)
{
    return new Dictionary<string, int>
    {
        { "ToDo", todo },
        { "InProgress", inProgress },
        { "Testing", testing },
        { "Done", done },
        { "Overdue", overdue }
    };
}

private static int CalculateRating(int done, int total)
{
    double ratio = (double)done / total;
    if (ratio < 0.3) return 1;
    if (ratio < 0.6) return 2;
    return 3;
}
```

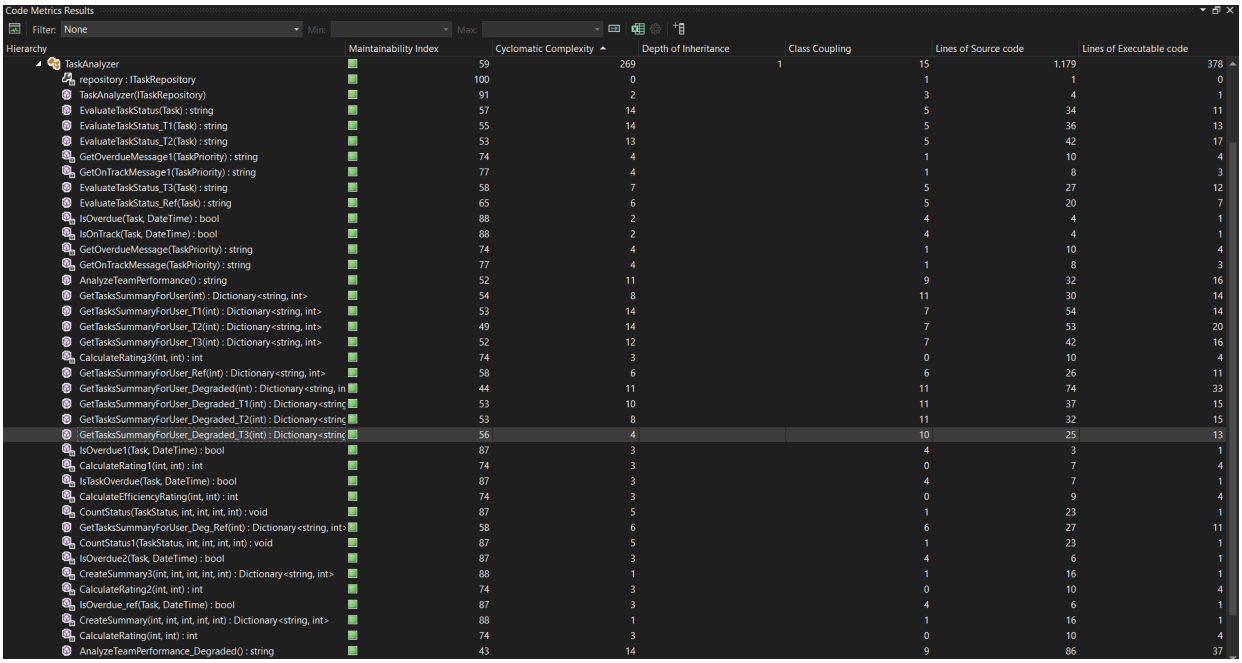
4.4 Utjecaj refaktoringa na metrike

Metrika	Prije (T3)	Poslije (Ref)	Komentar
V(G)	4	6	Porast zbog `foreach` i `switch`
MI	56	58	Poboljšan zbog bolje strukture

Tablica 12: Utjecaj refaktoringa na metrike (prije/poslije)

Metoda	V(G)	MI
`CountStatus`	5	87
`IsOverdue_ref`	3	87
`CreateSummary`	1	88
`CalculateRating`	3	74

Tablica 13: Metrike pomoćnih metoda nakon refaktoringa



Slika 4: Grafički prikaz *Visual Studio Metrics* nakon refaktoringa

4.5 Diskusija

Refaktoring je riješio fundamentalni problem performansi: umjesto 5 prolaza kroz listu (O(5n)), sada se vrši samo 1 prolaz (O(n)). Ovo je posebno značajno za velike liste zadataka.
Porast ciklomatične kompleksnosti sa 4 na 6 je očekivan i opravdan:

- `foreach` petlja dodaje jednu odlučujuću tačku
- `switch` u `CountStatus` dodaje grananja

Međutim, ova kompleksnost je **esencijalna** – predstavlja samu logiku algoritma, a ne strukturalnu složenost. Alternativa sa 5 `if` naredbi bi imala istu ili višu kompleksnost, ali lošiju čitljivost.
Najznačajniji doprinos refaktoringa:

- **Performanse:** O(5n) $\rightarrow$ O(n)
- **Modularnost:** 4 specijalizovane pomoćne metode
- **Testabilnost:** Svaka pomoćna metoda se može nezavisno testirati
- **Održivost:** MI poboljšan, jasna raspodjela odgovornosti

5. Zaključak

Kroz ovaj projektni zadatak urađena je detaljna analiza metode `GetTasksSummaryForUser(int userId)` iz aplikacije `TaskManagerApp`.
McCabe analiza: Utvrđeno je da originalna verzija metode ima ciklomatičnu kompleksnost V(G)=8. Poređenjem sa Code Metrics alatom potvrđena je ispravnost ručnog izračuna.
White-box testiranje: Primijenjeno je 7 tehnika testiranja. Dizajnirano je 11 testnih slučajeva koji postižu 100% statement, branch, condition i path coverage. MCDC je demonstriran za složene odluke, dok je loop coverage označen kao N/A jer metoda ne sadrži eksplicitne petlje.
Code tuning: Primijenjene su 3 tehnike optimizacije nad degradiranom verzijom metode:

- Tehnika 1 (izrazi): 88.7% smanjenje memorije
- Tehnika 2 (logički iskazi): pojednostavljenje uslova
- Tehnika 3 (metode): V(G) smanjen sa 11 na 4 (63.6% smanjenje)

Refaktoring: Primijenjena su 4 principa iz refactoring checkliste. Ključna promjena je zamjena 5 LINQ `Count()` poziva jednom `foreach` petljom, čime je algoritamska složenost smanjena sa O(5n) na O(n). MI je poboljšan sa 56 na 58.
Zaključak je da kombinacija metrika (McCabe, MI), white-box testova, code tuninga i refaktorisanja daje jasan i mjerljiv uvid u kvalitet koda. Primijenjene tehnike su dovele do brže, čišće i održivije implementacije metode, što je i osnovni cilj verifikacije i validacije softvera u praksi.