Primjena metaprogramiranja u razvoju alata za analizu podatkovnih rezultata softverskog sustava

Bičak, Sebastijan

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Sebastijan Bičak

PRIMJENA METAPROGRAMIRANJA U RAZVOJU ALATA ZA ANALIZU PODATKOVNIH REZULTATA SOFTVERSKOG SUSTAVA

ZAVRŠNI RAD

Varaždin, 2023.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Sebastijan Bičak

Matični broj: 0016150730

Studij: Informacijski i poslovni sustavi

PRIMJENA METAPROGRAMIRANJA U RAZVOJU ALATA ZA ANALIZU PODATKOVNIH REZULTATA SOFTVERSKOG SUSTAVA

ZAVRŠNI RAD

Mentor :

Dr. sc. Marko Mijač

Varaždin, rujan 2023.

Sebastijan Bičak

Izjava o izvornosti

Izjavljujem da je moj završni rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

U ovom radu istražujemo primjenu metaprogramiranja u analizi rezultata iz softverskog sustava. Prikazat će se razvoj programa pomoću metaprogramiranja u .NET tehnologiji koji odgovara na jedan od problema većih i kompleksnijih sustava. U takvim sustavima vrlo je teško analizirati podatke i međurezultate, zato što takvi sustavi svoju unutarnju programsku logiku skrivaju pomoću grafičkih sučelja. Razvoj softverskog rješenja za ovaj problem pratit će metodologiju znanstvenog oblikovanja. Znanstveno oblikovanje je metodologija koja se sve češće upotrebljava u informacijskim sustavima i programskom inženjerstvu, a cilj ove metodologije je dizajnirati i izraditi artefakte koji odgovaraju određenom kontekstu problema. U ovom slučaju, naš artefakt je softversko rješenje za spomenuti problem. Kod znanstvenog oblikovanja vrlo je važno demonstrirati i evaluirati artefakt. Rješenje će se demonstrirati prema jednom programu. Taj program je ZMG Desktop koji je bio potreban za jedan od kolegija. Nakon demonstracije, navedeni artefakt bit će evaluiran s tehničke i dizajnerske perspektive te odgovara li na spomenuti problem otežane analize kod kompleksnih sustava.

Ključne riječi: metaprogramiranje, .NET tehnologija, programsko inženjerstvo, razvoj softvera, analiza softvera, znanstveno oblikovanje, artefakt

Sadržaj

1. Uvod

U složenim softverskim sustavima s kompleksnom programskom logikom, suočavamo se s izazovom analize, posebno kada korisnicima pružamo samo ograničene informacije putem grafičkih sučelja. Ovakav pristup često ograničava našu sposobnost dubljeg razumijevanja stanja objekata tijekom izvođenja softvera, otežavajući identifikaciju problema i shvaćanje kako sustav funkcionira. Grafička sučelja često prikazuju samo sažete informacije ili sakrivaju detalje kako bi olakšala upotrebu, no to često znači da korisnici nemaju pristup ključnim informacijama poput međurezultata ili detaljnih podataka potrebnih za dublje razumijevanje sustava. Nedostatak ovih informacija otežava i provjeru ispravnosti, a korisnicima je teško pratiti tok izvođenja i prepoznati greške ili nepravilnosti. Ručno stvaranje svih mogućih prikaza podataka zahtijeva značajne resurse i vremenski angažman, dodatno povećavajući složenost softverskog sustava i otežavajući njegovo održavanje.

U cilju rješavanja ovog problema, ovaj rad istražuje primjenu metaprogramiranja u razvoju softverskih rješenja prateći metodologiju znanstvenog oblikovanja. Metaprogramiranje, tehnika programiranja, omogućuje razvoj generičkog mehanizma za prikazivanje podataka tijekom izvođenja, što može značajno olakšati analizu i upravljanje složenim softverskim sustavima.

U početku rada bit će prikazana metodologija znanstvenog oblikovanja i njezin radni okvir. Radni okvir sastoji se od nekoliko važnih područja ili konteksta koji utječu na dizajniranje i izradu artefakata. Također, bit će prikazan radni okvir za izradu artefakta u sklopu programskog inženjerstva.

Poslije znanstvenog oblikovanja slijedi definicija i primjena metaprogramiranja u .NET tehnologiji, zato što će softversko rješenje biti izrađeno u .NET-u. Govorit će se o dinamičkoj analizi programskog koda uz pomoć refleksije i generiranju koda uz T4, CodeDOM-a i Roslyn API-ja. Najviše pažnje bit će posvećeno refleksiji, jer uz pomoć te tehnike rješavamo opisani problem.

Nakon poglavlja o metaprogramiranju, slijedi poglavlje koje prati radni okvir znanstvenog oblikovanja u programskom inženjerstvu. Prvo ćemo definirati relevantan problem, a zatim izraditi specifikaciju zahtjeva prema industrijskim standardima. Nakon toga, usredotočit ćemo se na fazu dizajna i izrade artefakta, gdje ćemo istražiti dizajnerske principe i uzorke dizajna koji pridonose kvaliteti softvera. Na kraju, bit će prikazana demonstracija i evaluacija artefakta, čime ćemo zaključiti ovaj rad.

2. Znanstveno oblikovanje

Znanstveno oblikovanje je dizajn i istraživanje artefakata u kontekstu. Artefakti koje proučavamo dizajnirani su za interakciju s kontekstom problema kako bismo poboljšali neke stvari u tom kontekstu [1]. Kako bismo uopće mogli raditi znanstveno oblikovanje (engl. *Design Science*), dobro je poznavati komponente znanstvenog oblikovanja. Jedna od komponenti je objekt studije, ono što proučavamo, a druga komponenta su dvije glavne aktivnosti.

U dizajnerskoj aktivnosti potrebno je znati što dionici žele i koji su ciljevi projekta. U drugoj aktivnosti, istražiteljskoj aktivnosti, važno je biti upoznat s kontekstom znanja projekta, jer će se to znanje iskoristiti i znanju će se doprinijeti [1].

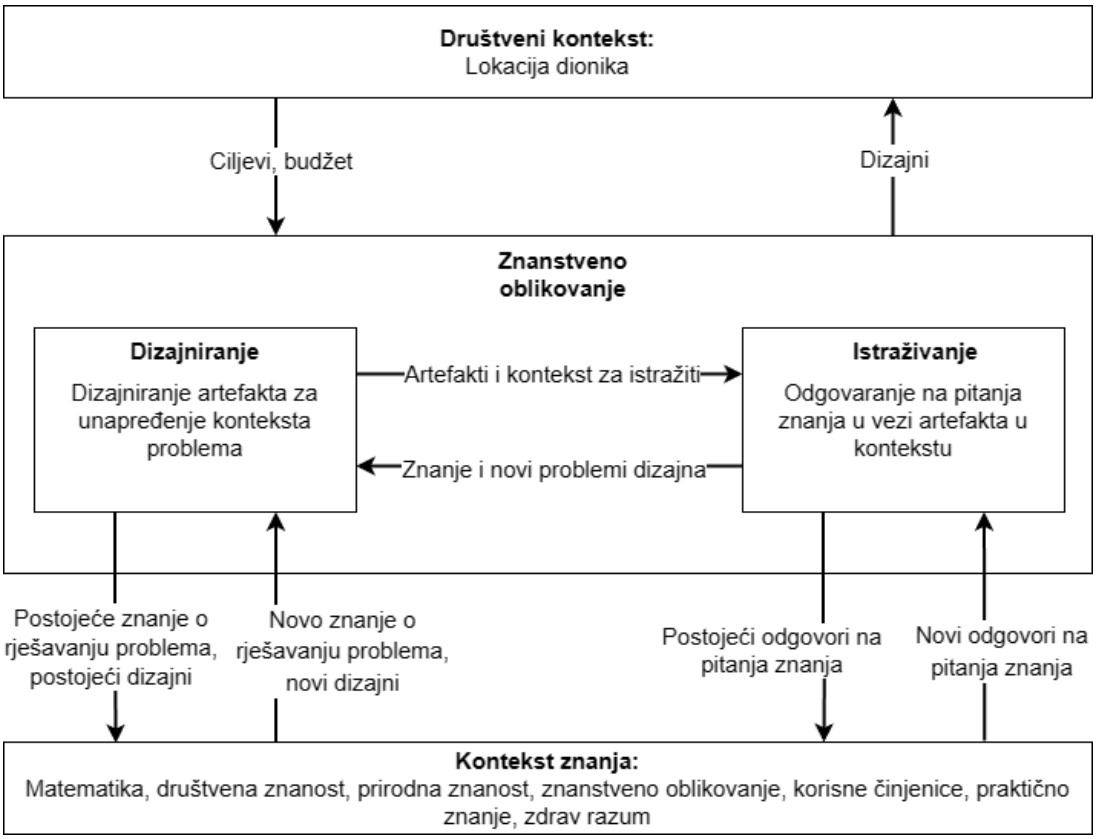
Svaki problem znanstvenog oblikovanja kojeg proučavamo je zapravo problem poboljšanja što znači da svaki problem (u slučaju ovog rada, to je problem otežane analize softvera) ima svoj sadržaj, kontekst u kojem se očekuje neko unaprjeđenje. Kako bismo razumjeli problem, moramo razumjeti kontekst tog problema. Ono što se dizajnira u dizajnerskoj aktivnosti postaje artefakt, a interakcija između artefakta i konteksta problema ima veliki utjecaj, zato što interakcija pomaže u pronalaženju odgovarajućeg rješenja. Važno je napomenuti da "artefakt može različito djelovati u različitim kontekstima problema te stoga može rješavati različite probleme u različitim kontekstima. Može čak i doprinijeti ciljevima dionika u jednom kontekstu, a stvoriti prepreke za postizanje cilja u drugom kontekstu." [1, str. 4].

2.1. Problemi istraživanja

Kako se znanstveno oblikovanje sastoji od dva dijela, dizajniranja i istraživanja, pojavljuju se i dva problema istraživanja. Jedan od problema istraživanja je problem dizajna koji zahtjeva analizu ciljeva dionika i promjenu. Rješenje je dizajn, a najčešće u praksi ima više rješenja za jedan problem. Drugi problem istraživanja su pitanja znanja. Takva pitanja ne zahtijevaju promjenu, ali traže znanje o stanju u kakvom je svijet, stoga je odgovor propozicija. Kada pokušavamo odgovoriti na pitanje, pretpostavljamo da je samo jedan odgovor. Odgovor može biti ili točan ili netočan i može dovesti do još više problema. Problem ovog rada je dizajniranje softvera koji će pomoći u otežanoj analizi kompleksnih sustava, a pitanje znanja je koliko je taj softver učinkovit u analizi. "Problemi mogu stvoriti nove probleme, a projekt znanstvenog oblikovanja nikada nije ograničen samo na jednu vrstu problema. Ovo generira ponavljanje problema dizajna i pitanja znanja u znanstvenom oblikovanju." [1, str. 6].

Znanje koje je dostupno prije projekta naziva se prethodno znanje, a znanje koje je proizvedeno kao rezultat projekta je posteriorno znanje. Znanje dobivamo iz literature, znanstvene, tehničke i profesionalne, a znanje dobivamo i slušanjem predavanja, seminara i slično od drugih osoba. Ako ne možemo pronaći odgovor, onda trebamo samostalno odraditi istraživanje za odgovor uz cijenu vremena, ponekad i novčanog iznosa [1].

2.2. Radni okvir znanstvenog oblikovanja



Slika 1: Radni okvir znanstvenog oblikovanja (Prema: Wieringa, 2014)

Društveni kontekst sadrži dionike projekta. Dionici utječu na projekt ili projekt utječe na njih. U dionike najčešće se ubrajaju korisnici, instruktori, održavatelji projekta i drugi. Također, oni uključuju i sponzore za projekt dizajna i njihova uloga je financiranje projekta i postavljanje ciljeva. Primjer sponzora može biti vlada ili netko drugi koji očekuje uspješan projekt koji će nešto unaprijediti. Kod znanstvenog oblikovanja vidimo interakciju između dizajniranja i istraživanja. Artefakti i kontekst utječu na aktivnost istraživanja, a znanje i novi problemi dizajna na aktivnost dizajniranja. Istovremeno, kontekst znanja potreban je za dizajniranje i istraživanje kao što je prikazano u slici iznad [1].

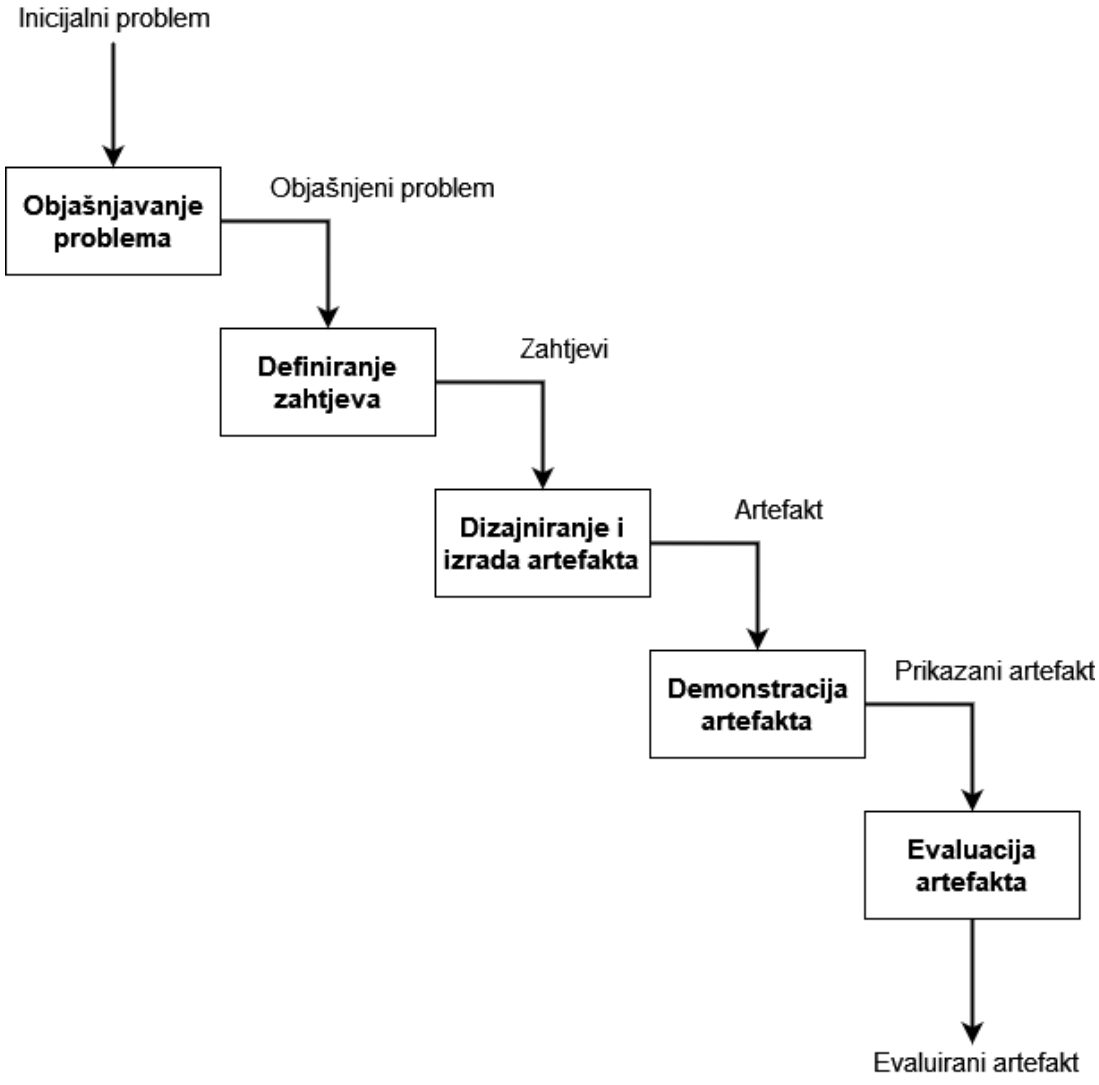
2.3. Programsko inženjerstvo i znanstveno oblikovanje

Mnogi ljudi bave se programiranjem kako bi si pojednostavili rad. Primjerice, znanstvenici i inženjeri pišu programe koji procesiraju podatke ili osobe koje se programiranjem bave u svoje slobodno vrijeme. Ali u većini slučajeva, programsko inženjerstvo treba smatrati profesionalnom aktivnošću gdje se izrađuje poslovni softver ili softverski proizvod. Primjer softverskog proizvoda je informacijski sustav u jednoj tvrtki. Ključan dio profesionalnog softvera uključuje i druge programere, developere. S time na umu, treba pažljivo osmisliti softver koji će biti pouzdan, održiv te ponovno iskoristiv. "Programsko inženjerstvo je inženjerska disciplina koja

se bavi svim aspektima proizvodnje softvera, počevši od početne koncepcije pa sve do rada i održavanja." [2, str. 21].

Mnogi ljudi misle da je softver druga riječ za računalni program, ali kada pomislimo na programsko inženjerstvo tada softver nije samo računalni program, nego se uključuje i dokumentacija, biblioteke, web stranice te bilo koja konfiguracija kako bi programi bili korisni. Temeljem toga, profesionalno razvijeni softverski sustav često se sastoji više od jednog programa [2].

Znanstveno oblikovanje u programskom inženjerstvu odvija se prema radnom okviru koji je različit od tipičnog okvira znanstvenog oblikovanja kao što je prikazano na slici 1. Sastoji se od pet aktivnosti koje su usko povezane s programskim inženjerstvom [3].



Slika 2: Radni okvir znanstvenog oblikovanja u programskom inženjerstvu (Prema: Mijač, García-Cabot i Strahonja, 2021)

U prvoj aktivnosti, objašnjavanje problema, problem definiramo i njega stavljamo u kontekst. Moramo prikazati problem koji je relevantan praktičarima i istraživačima. Kako bismo mogli napraviti i dizajnirati rješenje, istražujemo moguće uzroke problema [3].

U drugoj aktivnosti, definiranje zahtjeva, definiramo funkcionalne i nefunkcionalne zahtjeve koji pomažu u dizajniranju i stvaranju mogućeg rješenja. Takvo rješenje je artefakt. Prvo objasnimo općenite karakteristike artefakta, a kasnije objasnimo detaljnije zahtjeve. Glavni izvori za stvaranje rješenja, artefakta su pretraživanje literature, iskustvo istraživača i prototipiranje [3].

U trećoj aktivnosti, koja je iterirajuća, izrađujemo artefakt prema specificiranim zahtjevima iz druge aktivnosti. Stvaramo ideje i dizajn koji kasnije upotrebljavamo za stvaranje artefakta [3].

Nakon što smo napravili artefakt, trebamo ga demonstrirati na nekoliko primjera i provjeriti zadovoljava li specifikaciju zahtjeva što znači da ga evaluiramo i procjenjujemo. Evaluacija je vrlo važna, zato što dizajnu, artefaktu daje znanstveni dio. Pogodna svojstva za procjenu su tehnička izvedivost i efektivnost. Tehnička izvedivost pokazuje da je moguće izraditi artefakt iz tehničkog pogleda, a efektivnost koliko dobro rješenje odgovora problemu [3].

3. Pristupi metaprogramiranja u .NET-u

Metaprogramiranje se često opisuje kao sposobnost jednog programa da generira ili stvori novi program. Druga definicija metaprogramiranje smatra tehnikom programiranja koju jedan program koristi kako bi mogao tretirati drugi program kao varijablu. Ponekad program može tretirati samog sebe kao varijablu kako bi se mogao modificirati [4]. Slijedi jednostavan primjer metaprogramiranja u praksi. Program je napisan u Bash-u, koji je ljuska za izvršavanje naredbi u Linux operacijskom sustavu, djelujući kao sučelje za komunikaciju s jezgrom (engl. *Kernel*) sustava.

```
1 #!/bin/bash
2 echo "#!/bin/bash" > noviProgram.sh
3 echo "echo 'Hello World iz svijeta metaprogramiranja!'" >>
    noviProgram.sh
```

U primjeru iznad, vidimo kako će program zapisati unutar nove datoteke noviProgram.sh naredbu echo 'Hello World iz svijeta metaprogramiranja!'. Kada pokrenemo noviProgram.sh unutar terminala kod Linux operacijskog sustava, vidi se "echo" poruka.

Naravno, metaprogramiranje nije toliko jednostavno, ali generalna ideja je da se kompleksni sustavi mogu učinkovitije pratiti, analizirati i nadograđivati. Kompajleri, asembleri, linkeri i programi za otklanjanje pogrešaka (engl. *Debugger*) su sve primjeri metaprograma koji uzimaju napisani programski kod i njega prevađaju ili analiziraju. Debugger uglavnom analizira, a kompajler prevađa programski kod.

Dva pojma koja je korisno poznavati u svijetu metaprogramiranja su introspekcija, gdje program gleda samog sebe i daje izvješće, te refleksija, gdje program samog sebe modificira [5].

Najčešće se metaprogramiranje odvija kroz skriptiranje, kao na prethodnom primjeru u Bash-u, ali može se koristiti i u skriptnim programskim jezicima poput Rubyja, Pythona, pa čak i u JavaScriptu uz pomoć HTML-a. Ruby je poznat po svojim sposobnostima metaprogramiranja, a sintaksa jezika vrlo je jasna i čitljiva. Druga česta primjena metaprogramiranja je u generatorima koda. Na primjer, kada putem grafičkog sučelja stvaramo novu klasu unutar projekta u Visual Studiju. Također, metaprogramiranje se može koristiti unutar deklarativnog programiranja pomoću F# [6]. "Deklarativno programiranje je paradigma programiranja u kojoj programer definira što program treba postići bez definiranja kako to treba implementirati. Drugim riječima, pristup se fokusira na ono što treba postići umjesto da daje upute kako to postići." [7].

Velika prednost metaprogramiranja leži u razumijevanju sustava ili programa. Omogućuje nam jednostavniju izmjenu podataka neovisno o tome koliko je sustav kompleksan, ali uz cijenu kompleksnosti pisanja metaprograma. Ujedno, takvo znanje o metaprogramiranju i sposobnosti modificiranja programa prilikom izvođenja može izazvati štetne posljedice vezane uz sigurnosnu ranjivost sustava. Kao rezultat toga, nužno je poduzeti mjere kako bi se osigurao integritet i osigurala sigurnost sustava.

Metaprogramiranje u .NET tehnologiji najčešće se odvija uz pomoć vanjskih biblioteka ili biblioteka koje su već smještene unutar .NET tehnologije. Primjer jedne već ukomponirane biblioteke je System.Reflection. U nastavku slijede potpoglavlja vezana uz opis biblioteka i njihovih sposobnosti metaprogramiranja unutar .NET-a.

3.1. Refleksija

Refleksija je koncept koji je dugo prisutan u mnogim programskim jezicima, prije nego što je .NET postojao. Refleksija omogućuje programerima čitanje sadržaja programa i izvršavanja njegovog koda. Dubina promišljanja koju jezici i platforme nude varira, ali općenito bilo koji sustav koji omogućuje pregled i pozivanje koda tijekom izvođenja koristi neki oblik refleksije [6].

Refleksija uključuje klasifikacije koje pristupaju informacijama o komponentama, modulima, članovima, argumentima i drugim entitetima u programskom kodu istražujući njihove metapodatke. Ove klasifikacije također omogućuju upravljanje instancama učitanih klasa, na primjer, za povezivanje događaja ili pozivanje metoda [8].

System.Reflection pruža vrlo korisne metode koje se mogu iskoristiti za pregledavanje metapodataka raznih tipova podataka. U nastavku bit će opisane samo najvažnije metode i jednostavni primjeri za shvaćanje što System.Reflection omogućuje programerima.

Uzmimo sljedeći scenarij. Imamo klasu Student koja sadrži nekoliko posebnih svojstva i nekoliko metoda. Neka od svojstva imaju i metaatribute. Cilj nam je pročitati svojstva i metode instance objekta klase Student. Za primjer, slijedi implementacija klase Student.

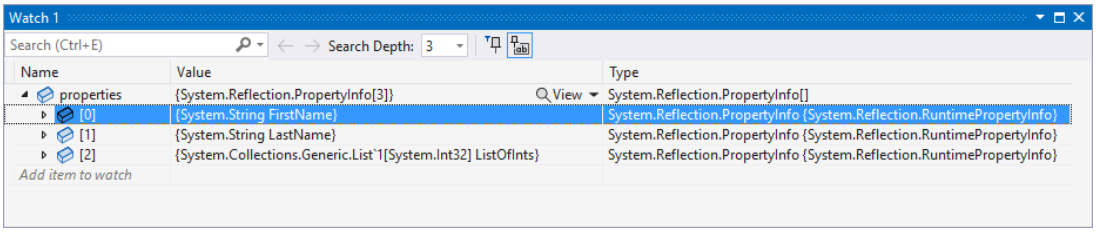
```
1  public class Student
2  {
3      [Browsable(true)] // metaatribut
4      public string FirstName { get; set; }
5
6      public string LastName { get; set; }
7      public List<int> ListOfInts { get; set; }
8
9      public void GetProjects()
10     {
11
12     }
13 }
```

Klasa sadrži tri svojstva i jednu metodu GetProjects. Svojstvo FirstName ima metaatribut Browsable(true). Neke od najvažnijih klasa koje će pomoću u dohvaćanju svojstava i metoda su PropertyInfo, MethodInfo i MemberInfo.

3.1.1. PropertyInfo

ProperyInfo je klasa koja nam pruža sve informacije o svojstvu (engl. *Property*). Slijedi primjer dohvaćanja svih svojstava unutar instance tipa Student.

```
1 using System.Reflection;
2
3 object item = new Student()
4 {
5     FirstName = "Ime"
6 };
7 PropertyInfo[] properties = item.GetType().GetProperties(
    BindingFlags.Instance | BindingFlags.Public | BindingFlags.
    NonPublic);
```



Slika 3: Watch panel za properties (Izvor: autorski rad, 2023)

Možemo vidjeti kako se dohvaćaju sva svojstva i spremaju se u polje properties tipa PropertyInfo[]. Kod metode GetProperties koja poprima BindingFlags parametar, važno je staviti BindingFlags argumente za instancu, javna i/ili privatna, zaštićena svojstva. Slijedi primjer kako se može dohvatiti ime i vrijednost svojstva.

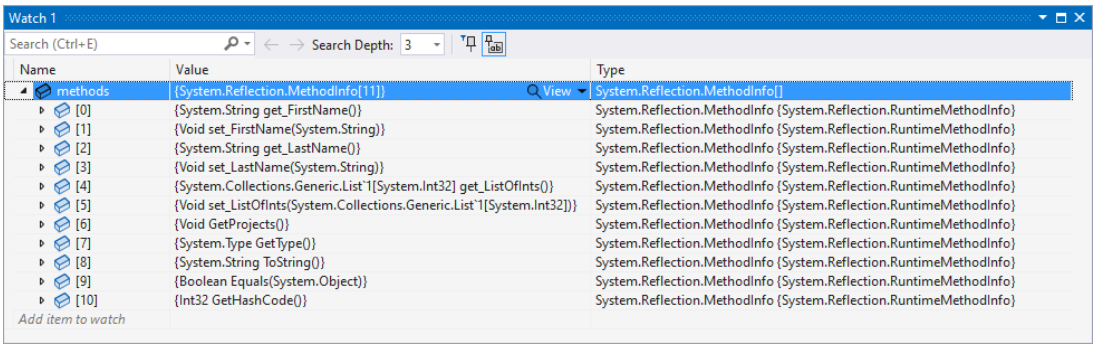
```
1 var svojstvo = properties[0];
2 string imeSvojstva = svojstvo.Name;
3 var vrijednostSvojstva = svojstvo.GetValue(item); // item
    oznacava instancu tipa Student
```

3.1.2. MethodInfo

Klasa koja sadrži sve podatke o metodama koje ima instancirani objekt je MethodInfo. Da bismo dohvatili metapodatke o metodama definiranima u nekoj klasi, možemo koristiti metodu GetMethods. Također, dohvaćene metode mogu se izvršiti pomoću metode Invoke. Slijedi primjer dohvaćanja metoda.

```
1 using System.Reflection;
2
3 object item = new Student()
4 {
5     FirstName = "Ime"
```

```
6  };
7  MethodInfo[] methods = item.GetType().GetMethods();
```



Slika 4: Watch panel za methods (Izvor: autorski rad, 2023)

Metoda GetMethods vraća samo metode koje imaju javnu vidljivost. Vratit će se metoda GetProjects koju sadrži klasa Student, ali uz tu metodu vratit će se i Getter i Setter metode koje su zaslužne za dohvaćanje i postavljenje vrijednosti svojstava. Naziv Getter metode svojstva FirstName je get_FirstName, a Setter metode je set_FirstName. Takvo nazivlje vrijedi za svako svojstvo.

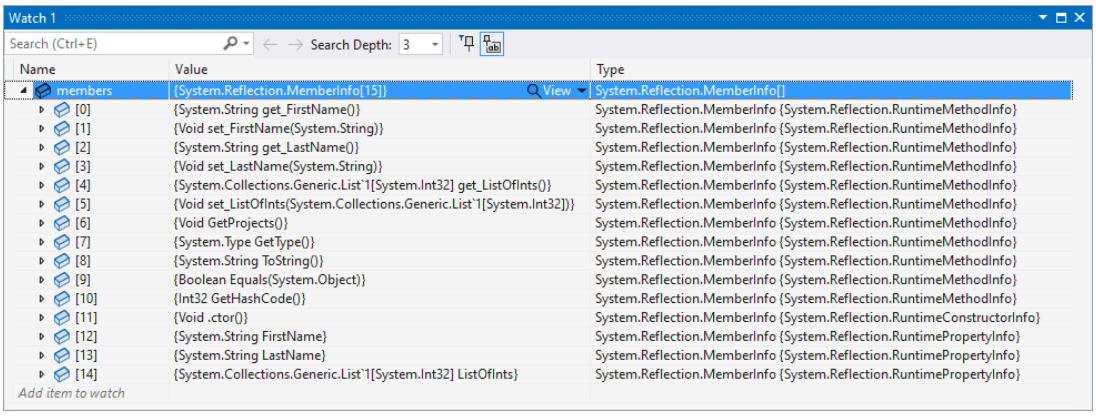
3.1.3. MemberInfo

MemberInfo je klasa čija je svrha dobiti sve članove, a to su sva svojstva, metode i polja (engl. *Field*). Slijedi primjer definiranja polja i svojstva.

```
1  public class Project
2  {
3      private string pravoIme;
4      public string PravoPrezime { get; set; }
5  }
```

Primjer dohvaćanja svih članova klase Student.

```
1  object item = new Student()
2  {
3      FirstName = "Ime"
4  };
5  MemberInfo[] members = item.GetType().GetMembers();
```



Slika 5: Watch panel za members (Izvor: autorski rad, 2023)

Obrada podataka s members instancom slična je s PropertyInfo, MethodInfo i FieldInfo klasom. FieldInfo je klasa u koju se spremaju informacije o objašnjenom polju.

3.2. Predlošci T4

T4, skraćenica od Text Template Transformation Toolkit, služi za metaprogramiranje, prvenstveno za generiranje koda pomoću predložaka koji su zapisani unutar tekstualne datoteke. Ujedno, može se koristi i za automatsko generiranje tekstualnih datoteka, primjerice XML s različitim parametrima. T4 koristi se u ASP.NET tehnologiji koja je zaslužna za izradu web stranica u .NET ekosustavu [6]. U ASP.NET-u možemo uklopiti C# jezik s HTML-om, CSS-om i JavaScriptom što uvelike daje prednost kod izrade web stranica, ukoliko je netko dobro upoznat s .NET okruženjem i raznim bibliotekama od treće strane koje su javno dostupne. Generatori koda za ASP.NET MVC, ADO.NET Entity Framework i drugi popularni okviri temelje se na T4, što ga čini jednim od najčešće korištenih okvira za izradu alata [6].

Kako bismo mogli koristiti generirati kod pomoću T4 unutar Visual Studija, potrebno je cijeli tekst napraviti unutar datoteke koja ima nastavak .tt. Nastavak označava prva slova od Text Template. Većina takvih predložaka počinje s <#@ i #>, a ostale linije teksta su tekstualni blokovi ili kontrolni blokovi. Najčešće direktive koje se koriste su:

- **Template Language** — Koristi se kako bi se specificirao programski jezik i postavke za kompajler, prevoditelj.
- **Output Extension** — Koristi se za upravljanje nastavkom generirane datoteke.
- **Assembly** — Koristi se za referenciranje .NET DLL-ova.
- **Import** — Koristi se za uključivanje ili korištenje C# i Visual Basic direktiva.

```
1 <#@ template language="C#" #>
2 <#@ output extension=".cs" #>
3 using System;
4
```

```
5  namespace T4Example
6  {
7      public class T4Class
8      {
9          <# for (int i = 1; i <= 5; i++) { #>
10         public string Property<#= i #> { get; set; }
11         <# } #>
12     }
13 }
```

U primjeru predloška iznad, koji se nalazi u datoteci s .tt nastavkom, generira se klasa koja ima pet svojstva tipa string. Pomoću tekstualnih i kontrolnih blokova, unutar for petlje generiramo svojstva. Prvo svojstvo je Property1, drugo Property2 i tako sve do pet. Kako bi se kod generirao, potrebno je izgraditi (engl. *Build*) projekt unutar Visual Studija. Temeljem toga, dobije se sljedeća klasa.

```
1  using System;
2
3  namespace T4Example
4  {
5      public class T4Class
6      {
7                  public string Property1 { get; set; }
8                  public string Property2 { get; set; }
9                  public string Property3 { get; set; }
10                 public string Property4 { get; set; }
11                 public string Property5 { get; set; }
12             }
13 }
```

3.3. CodeDOM

Također, metaprogramiranje može se izvoditi i pomoću CodeDOM-a za generiranje koda. CodeDOM možemo shvatiti na sličan kao i HTML. HTML ima svoj DOM (engl. *Document Object Model*) za generiranje prozora i za navigaciju unutar web stranica. Slično se može primijeniti i na CodeDOM za programski kod napisan u C#. Omogućuje stvaranje klasa, metoda i izjava koje se mogu kompajlirati i izvršiti prilikom izvođenja programa, a kako bi se mogle koristiti njegove metode i funkcije treba uključiti imenski prostor System.CodeDom unutar programa [6].

CodeDOM koristi se za generiranje koda tako što tretira logiku i strukturu koda kao informacije iliti podatke. Javio se odmah u originalnoj verziji .NET, 1.0 u veljači 2002. godine. U samo nekoliko novih izdanja radnog okvira (engl. *Framework*), dobio je veliko unapređenje time što je podržavao delegatni model i dodatne generičke oblike. [6].

CodeDOM se služi grakonima koda kako bi programski kod izrazio kao podatke. Graf koda je struktura podataka s hijerarhijskom organizacijom koja prikazuje logiku i strukturu algoritma. Ovaj graf može biti generiran iz izvornog koda ili se može postupno izgraditi [6].

CodeDOM pretežno je kompleksna kolekcija klasa. Sve potrebne klase nalaze se u imenskom prostoru System.CodeDom i System.CodeDom.Compiler, a glavna korijenska klasa je CodeObject koja sadrži samo rječnik listu (engl. *ListDictionary*) UserData. Evo nekoliko klasa koje nasljeđuju CodeObject bazičnu klasu [6]:

- **CodeNamespace** — Koristi se kako bi se definirao imenski prostor generiranog koda.
- **CodeTypeDeclaration** — Ovom klasom definiramo tip podatka. Najčešće je to IsClass, IsInterface i IsStruct.
- **CodeMemberMethod** — Koristi se za definiranje metode klase s parametrima, glavnim dijelom i atributima.
- **CodeCompileUnit** — Služi za stvaranje kompajlerske jedinice koja sadrži imenski prostor i tipove podataka.
- **CodeComment** - Klasa služi za dodavanje komentara.

```
1  using System;
2  using System.CodeDom;
3  using System.CodeDom.Compiler;
4  using Microsoft.CSharp;
5  using System.IO;
6
7  class Program
8  {
9      static void Main()
10     {
11         // Napravimo namespace
12         CodeNamespace codeNamespace = new CodeNamespace("
               CodeDOMImenskiProstor");
13
14         // Stvorimo klasu
15         CodeTypeDeclaration classDeclaration = new
               CodeTypeDeclaration("PrimjerKlase");
16         classDeclaration.IsClass = true;
17
18         // Radimo metodu
19         CodeMemberMethod method = new CodeMemberMethod();
20         method.Name = "PrintMessage";
21         method.Attributes = MemberAttributes.Public;
```

```
22          method.Statements.Add(new CodeSnippetStatement("Console.
                WriteLine(\"Hello, CodeDOM!\");"));
23
24          // Dodajemo metodu kao član klase
25          classDeclaration.Members.Add(method);
26
27          // Dodajemo deklaraciju u namespace
28          codeNamespace.Types.Add(classDeclaration);
29
30          // Stvaranje prevoditeljske jedinice
31          CodeCompileUnit compileUnit = new CodeCompileUnit();
32          compileUnit.Namespaces.Add(codeNamespace);
33
34          // Generiranje koda
35          CSharpCodeProvider codeProvider = new CSharpCodeProvider
                ();
36          using (StreamWriter writer = new StreamWriter("
                CodeDOMGeneriranaKlasa.cs"))
37          {
38              codeProvider.GenerateCodeFromCompileUnit(compileUnit,
                    writer, new CodeGeneratorOptions());
39          }
40      }
41  }
```

U primjeru iznad vidimo kako se uz pomoć CodeDOM-a može generirati kod. U trenutnom slučaju generira se klasa pod nazivom PrimjerKlase koja ima imenski prostor CodeDOMImenskiProstor i jednu metodu PrintMessage() koja je javno vidljiva i ima jednu liniju koda. Linija koda unutar konzole ispisuje poruku Hello, CodeDOM!. Slijedi sadržaj datoteke koja je generirana, CodeDOMGeneriranaKlasa.cs.

```
1  //
       ------------------------------------------------------------------------------
2  // <auto-generated>
3  //     This code was generated by a tool.
4  //     Runtime Version:4.0.30319.42000
5  //
6  //     Changes to this file may cause incorrect behavior and will
       be lost if
7  //     the code is regenerated.
8  // </auto-generated>
9  //
       ------------------------------------------------------------------------------
```

```
10
11 namespace CodeDOMImenskiProstor {
12
13
14     public class PrimjerKlase {
15
16         public virtual void PrintMessage() {
17 Console.WriteLine("Hello, CodeDOM!");
18         }
19     }
20 }
```

3.4. Roslyn

Roslyn predstavlja napredniju verziju CodeDOM-a, koja je zapravo okvir za dinamičko generiranje koda, i kompajlera, odnosno prevoditelja. Kompajlerski proces bio je često zatvoren, što je otežavalo metaprogramiranje sve dok Microsoft nije izdao Roslyn. Prevoditelj pruža skup API-ja koji omogućuje programsku analizu i manipuliranje kodom na tim jezicima. S Roslynom se dinamički analizira i generira kod te omogućuje napredne scenarije analize koda, alate za refaktoriranje i pomoćne programe za generiranje koda [6].

Glavni zadatak Microsofta, kada je radio na Roslynu, bio je da se nekadašnji kompajler unaprijedi omogućujući programerima bolji uvid u stvaranje DLL-ova (engl. *Dynamic-Link Library*). Nekadašnji prevoditelji bili su kao crna kutija (engl. *Black Box*). Crna kutija u ovom kontekstu znači da nije bilo moguće vidjeti kako prevoditelj radi na prevađanju koda. Vrlo je jednostavno dati kompajleru neku datoteku da prevede i vratit će DLL. Rad kompajlera je vrlo kompleksan, a Project Roslyn daje uvid u rad prevoditelja [6].

Roslyn je sada kompajler bijele kutije (engl. *White Box*). Unutar bijele kutije vidi se kako neki sustav radi. Kompajler sada omogućuje uvid u parsiranje, simbole, povezivanja (engl. *Binding*) i emitiranja (engl. *Emitting*) [6]. Kako bi mogli koristiti Roslyn, mora se instalirati unutar projekta u Visual Studiju kao NuGet paket Microsoft.CodeAnalysis. Slijedi primjer korištenja Roslyn API-ja za generiranje jednostavne klase slične kao i CodeDOM-a.

```
1 using System;
2 using Microsoft.CodeAnalysis;
3 using Microsoft.CodeAnalysis.CSharp;
4 using Microsoft.CodeAnalysis.CSharp.Syntax;
5 using System.IO;
6
7 class Program
8 {
9     static void Main()
```

```
10      {
11          // Kreiramo deklaraciju za metodu klase
12          MethodDeclarationSyntax methodDeclaration = SyntaxFactory
                .MethodDeclaration(
13              SyntaxFactory.PredefinedType(SyntaxFactory.Token(
                    SyntaxKind.VoidKeyword)),
14              "PrintMessage")
15              .WithModifiers(SyntaxFactory.TokenList(SyntaxFactory.
                    Token(SyntaxKind.PublicKeyword)))
16              .WithBody(SyntaxFactory.Block(
17                  SyntaxFactory.ExpressionStatement(
18                      SyntaxFactory.ParseExpression("Console.
                            WriteLine(\"Hello, Roslyn!\");"))));
19
20          // Kreiramo deklaraciju za klasu
21          ClassDeclarationSyntax classDeclaration = SyntaxFactory.
                ClassDeclaration("PrimjerKlase")
22              .WithModifiers(SyntaxFactory.TokenList(SyntaxFactory.
                    Token(SyntaxKind.PublicKeyword)))
23              .WithMembers(SyntaxFactory.SingletonList<
                    MemberDeclarationSyntax>(methodDeclaration));
24
25          // Napravimo deklaraciju za imenski prostor
26          NamespaceDeclarationSyntax namespaceDeclaration =
                SyntaxFactory.NamespaceDeclaration(SyntaxFactory.
                ParseName("RoslynImenskiProstor"))
27              .WithMembers(SyntaxFactory.SingletonList<
                    MemberDeclarationSyntax>(classDeclaration));
28
29          // Napravimo stablo sintakse
30          SyntaxTree syntaxTree = SyntaxFactory.SyntaxTree(
                namespaceDeclaration);
31
32          // Sve zapisemo u datoteku
33          using (StreamWriter writer = new StreamWriter("
                RoslynGeneriranaKlasa.cs"))
34          {
35              syntaxTree.GetRoot().NormalizeWhitespace().WriteTo(
                    writer);
36          }
37      }
38  }
```

Stablo sintakse je potpuni prikaz koda koji se analizira, uključujući razmake (poznate kao trivia). Važno je unutar stabla sintakse staviti takve razmake zbog izgleda datoteke u koju će se generirati kod. [6]. Stabla unutar Roslyna su nepromjenjiva (engl. *Immutable*) što znači da se ne može promijeniti kontekst stabla, ali možemo napraviti novo stablo koje se temelji na originalu s našim promjenama. Stabla imaju velike prednosti. Jedna od prednosti je što pomoću njih istovremeno programiranje možemo lakše razumjeti [6]. Slijedi generirana klasa iz prijašnjeg primjera.

```
1  namespace RoslynImenskiProstor
2  {
3      public class PrimjerKlase
4      {
5          public void PrintMessage()
6          {
7              Console.WriteLine("Hello, Roslyn!") ;;
8          }
9      }
10 }
```

4. Provedba znanstvenog oblikovanja

Softver će biti izrađen pomoću već spomenute metodologije u .NET tehnologiji pomoću C# programskog jezika u skladu s objektno-orijentiranim principima. Također, koristit će se i ostali programi za dizajn softvera. Slijedi popis korištenih programa.

- **.NET** - "Besplatna razvojna platforma otvorenog koda za više platformi za izradu mnogo različitih vrsta aplikacija." [7].
- **WinForms** - "UI okvir koji stvara obogaćene desktop klijentske aplikacije za Windows." [9].
- **Visual Studio Community** - "Potpuno opremljen, proširiv, besplatan IDE za izradu modernih aplikacija za Android, iOS, Windows, kao i web aplikacija i usluga u oblaku." [10].
- **Visual Paradigm Community** - Softver pomoću kojeg se izrađuju UML dijagrami, ERA modeli i ostali artefakti.
- **Figma** - Web aplikacija za dizajniranje prototipa aplikacija i skica grafičkih sučelja različitih sustava.
- **draw.io** - Web aplikacija za dizajniranje jednostavnih različitih dijagrama.

Sljedeća potpoglavlja odnose se na razvoj softvera koji odgovara na problem otežane analize softvera. Potpoglavlja će biti istog naziva kao i aktivnosti u radnom okviru znanstvenog oblikovanja u programskom inženjerstvu (slika 2).

4.1. Objašnjavanje problema

Kod softverskih sustava s kompleksnom programskom logikom, analiza može biti izazovna, posebno kada korisnicima kroz grafička sučelja prikazujemo samo djelomične informacije. Ovo ograničava dublji uvid u stanje objekata tijekom izvođenja softvera te otežava identifikaciju problema i razumijevanje funkcioniranja sustava. Grafička sučelja često prikazuju agregirane podatke ili skrivaju detalje kako bi sučelje bilo jednostavnije za korištenje. No, ovo često znači da korisnici nemaju pristup međurezultatima, ne-agregiranim podacima ili drugim ključnim informacijama potrebnim za dublje razumijevanje sustava. Nedostatak detaljnih informacija o međurezultatima ili promjenama stanja objekata tijekom izvođenja otežava provjeru ispravnosti. Korisnicima je teško pratiti tijek izvođenja i identificirati greške ili nepravilnosti. Ručno implementiranje svih mogućih kombinacija prikaza podataka zahtijeva značajno vrijeme i resurse. Ovo dodatno povećava složenost softverskog sustava i otežava održavanje. Metaprogramiranje, tehnika koja omogućava programima da manipuliraju sami sobom kao podacima, može biti ključno za rješavanje ovih problema. Kroz dinamičko generiranje koda i pristup podacima, metaprogramiranje omogućava stvaranje generičkog mehanizma za prikazivanje podataka tijekom izvođenja.

4.2. Definiranje zahtjeva

Specifikacija softvera ili inženjering zahtjeva je proces razumijevanja i definiranja usluga koje su zahtijevane od strane sustava. Također, identificiraju se sva ograničenja upravljanja sustavom i njegovim razvojem. S time možemo zaključiti da je inženjering zahtjeva kritičan dio u softverskom razvoju, jer pogreške koje se naprave u toj fazi, mogu prouzročiti probleme u fazi dizajna i implementacije. Prije izrade specifikacije zahtjeva potrebno je istražiti postoji li već neki drugi sličan softver te je li moguće tehnički izraditi softver i je li financijski prikladno [2].

Proces specificiranja zahtjeva cilja na izradu dokumenta koji specificira budući softver ili sustav kojeg traži tržište i dionici. Zahtjevi su najčešće prikazani u dvije razine detalja. Na većoj razini, apstrahiranoj razini, zahtjevi su napravljeni za krajnje korisnike, a na nižoj razini, detaljnijoj razini, za programere koji trebaju detaljniju specifikaciju [2].

Tri su glavne aktivnosti u procesu specificiranja zahtjeva [2]:

- **Elicitacija zahtjeva** — Proces u kojem analiziramo sustav koji će se izraditi i uspoređujemo budući sustav s ostalim sustavima kako bi prikupili prikladne informacije.
- **Specifikacija zahtjeva** — Aktivnost u kojoj se transformiraju informacije dobivene u elicitaciji zahtjeva u prave zahtjeve.
- **Validacija zahtjeva** — Aktivnost u kojoj se provjerava konzistentnost, realnost i potpunost specifikacije zahtjeva. Nađene pogreške moraju se ispraviti.

Postoji mnogo predložaka za pisanje softverskih zahtjeva. Možemo sami osmisliti predložak ili možemo iskoristiti već gotove predloške. Jedan od takvih predložaka je IEEE 830-1998 [11]. Stoga u nastavku slijedi specifikacija softverskih zahtjeva prema predlošku za spomenuti problem otežane analize i budućeg softvera kao rješenje na problem.

4.2.1. IEEE 830-1998

1. Uvod

1.1. Svrha

Ovaj dokument predstavlja specifikaciju softverskih zahtjeva za softver koji pomaže u analiziranju podataka skrivenih od korisnika u kompleksnim sustavima. Ciljana skupina specifikacije zahtjeva su programski inženjeri i programeri koji trebaju osmisliti buduće rješenje te testeri čija je uloga testirati buduće rješenje i provjeriti ispunjava li softver zahtjeve.

1.2. Opseg

Analiza složenih softverskih sustava često je izazovna zbog kompleksne programske logike. Grafička sučelja često prikazuju djelomične informacije što otežava detaljnu analizu. Kod složenih sustava, korisnici (programeri, testeri, napredni korisnici) trebaju dublji uvid u stanje objekata tijekom izvođenja softvera. Često su suočeni s nedostatkom informacija, poput nevidljivih međurezultata ili neagregiranih podataka. To komplicira provjeru ispravnosti i razumijevanje funkcioniranja sustava. Jedan od izazova je već spomenuta djelomična informacija oko objekata i metoda vezanih uz njih. Time se otežava provjera ispravnosti te je korisnicima teže identificirati problem tijekom izvođenja. Još jedan od izazova je složenost implementacije. Vrlo je nezgodno i otežano nadzirati i analizirati kompleksne sustave zbog njihove programske implementacije. Metaprogramiranje ovdje uskače kao rješenje, zato što daje odgovore na spomenute izazove. Prednost metaprogramiranja je u dinamičkom prikazivanju podataka, brzoj identifikaciji problema i fleksibilnosti kao generički mehanizam koji podržava različite scenarije analize. Buduće softversko rješenje, RED MetaAnalyzer, za navedeni problem bit će samostalno rješenje. RED MetaAnalyzer će analizirati i prikazati bilo koji složeniji podatak unutar .NET tehnologije i C# programskog jezika. Softversko rješenje neće prikazivati primitivne podatke, primjerice integer ili string.

1.3. Definicije, akronimi, skraćenice

- **RED** - Reflection Enabled Declassification. Pomoću metaprogramiranja, tj. refleksije moći će se pregledati podaci vezani uz složenije podatke. RED predstavlja samo naziv koji upućuje na to.
- **MetaAnalyzer** - Naziv koji naglašava metaprogramiranje i analiziranje složenih podataka i struktura.
- **Metaprogramiranje** - Programiranje gdje pomoću programskog koda generiramo programski kod i analiziramo ga na nižoj razini.
- **.NET** - Microsoftova tehnologija otvorenog koda pomoću koje se izrađuju softverske aplikacije.

1.4. Reference

- IEEE 830-1998

1.5. Struktura dokumenta

U poglavlju 2, opisujemo perspektive i funkcije budućeg softverskog rješenja, RED MetaAnalyzer. Također, opisujemo karakteristike korisnika koji će upotrebljavati softver te ograničenja koja će možda utjecati na razvoj softverskog rješenja.

U poglavlju 3 definiramo funkcionalne zahtjeve za RED MetaAnalyzer na razini gdje dizajneri i programeri mogu početi s osmišljavanjem i implementacijom budućeg rješenja. Testeri tada mogu osmišljavati testne slučajeve.

U poglavlju 4 definiramo nefunkcionalne zahtjeve za buduće softversko rješenje. Programeri i dizajneri trebaju uzeti te zahtjeve u obzir kako bi osmislili prikladnu arhitekturu za rješenje te koje tehnologije bi se iskoristile.

U poglavlju 5 daje se uvid u grafičko sučelje RED MetaAnalyzera. Bit će priložene skice grafičkog sučelja.

2. Općeniti opis

2.1. Perspektiva proizvoda

RED MetaAnalyzer zamišljen je kao samostalno softversko rješenje koje je generičko za problem otežane analize sustava. Rješenje bi trebalo sadržavati klijentu aplikaciju koja bi se mogla ukomponirati u drugi sustav te kako bi RED MetaAnalyzer mogao analizirati podatke. RED MetaAnalyzer ne sadrži bazu niti zahtjeva bilo kakav pristup internetu što upućuje da nema potrebe za izravno korištenje hardverskih ili komunikacijskih tehnologija. Bilo koja uporaba takvih resursa, predviđena je za operacijski sustav ili okruženje za vrijeme izvođenja programa.

2.2. Funkcije proizvoda

Budući korisnici očekuju sljedeće mogućnosti koje pruža RED MetaAnalyzer:

- Pregledavanje složenih tipova podataka koje sadrži jedan objekt.
- Pregledavanje metoda i svojstava promatranog objekta.
- Mogućnost ulaska u dubinu podataka trenutno analiziranog objekta.
- Praćenje dubinskog ulaska u objekte.
- Mogućnost vraćanja iz dubinskog ulaska.
- Mogućnost filtriranja analiziranih podataka.
- Ispis analiziranih podataka u JSON, XML ili CSV obliku.

2.3. Karakteristike korisnika

Korisnici koji će koristiti softversko rješenje RED MetaAnalyzer su programski inženjeri, programeri i testeri. Svi korisnici bi trebali posjedovati naprednu razinu računalne i tehničke pismenosti.

2.4. Ograničenja

Nema dodatnih ograničenja, zato što se ne radi o sigurnosno kritičnoj domeni i hardverska konfiguracije u ovo doba je više nego dovoljna za rad sa softverom. Očekuje se da će RED MetaAnalyzer biti izrađen u skladu s dobrim praksama struke.

2.5. Pretpostavke i ovisnosti

Korištenje softverskog rješenja ovisno je o .NET tehnologiji. Za vrijeme izrade softverskog rješenja, ne očekuju se izmjene u odabranoj tehnologiji koji bi izazvale promjene u softverskim zahtjevima.

2.6. Ostalo

Nema potrebe za navođenjem dodatnih informacija.

3. Specifični funkcionalni zahtjevi

Tablica 1: Funkcionalni zahtjev 1

Identifikator	FZ-1
Zahtjev	Sustav će omogućiti dohvaćanje članova objekta.
Obrazloženje	RED MetaAnalyzer mora prikazati sve članove objekta. Članovi su metode i svojstva klase.
Način provjere	Članovi objekta trebaju biti vidljivi na grafičkom sučelju.
Prioritet[1-5]	1
Izvor/Porijeklo	Programeri i testeri

Tablica 2: Funkcionalni zahtjev 2

Identifikator	FZ-2
Zahtjev	Sustav će omogućiti dohvaćanje vrijednosti svojstava objekta.
Obrazloženje	RED MetaAnalyzer mora dohvatiti moguće vrijednosti članova objekta. Taj dio najviše se odnosi na svojstva promatranog objekta.
Način provjere	Vrijednosti članova objekta trebaju biti vidljivi na grafičkom sučelju.
Prioritet[1-5]	1
Izvor/Porijeklo	Programeri i testeri

Tablica 3: Funkcionalni zahtjev 3

Identifikator	FZ-3
Zahtjev	Sustav će omogućiti prikaz detalja složenijih članova objekta.
Obrazloženje	Ukoliko objekt sadrži složeniji tip podatka, RED MetaAnalyzer treba omogućiti jedinstveni prikaz za taj složeniji tip.
Način provjere	Složeniji tip podatka prikazan je u drugačijem obliku nego obična instanca klase.
Prioritet[1-5]	1
Izvor/Porijeklo	Programeri i testeri

Tablica 4: Funkcionalni zahtjev 4

Identifikator	FZ-4
Zahtjev	Sustav će omogućiti praćenje podobjekata trenutno promatranog objekta.
Obrazloženje	RED MetaAnalyzer ima mogućnost ulaska u podobjekte promatranog objekta i izlaska iz promatranog podobjekata.
Način provjere	Prikaz članova i vrijednosti u sustavu se mijenja kada ulazimo u podobjekt ili izlazimo iz podobjekta.
Prioritet[1-5]	2
Izvor/Porijeklo	Programeri i testeri

Tablica 5: Funkcionalni zahtjev 5

Identifikator	FZ-5
Zahtjev	Sustav će omogućiti izvoz prikazanih podataka u bilo kojem trenutku.
Obrazloženje	RED MetaAnalyzer ima mogućnost izvoza prikazanih podataka u formatu koji su prikladni za daljnju analizu.
Način provjere	Prikazani podaci zapisani su u odgovarajućoj datoteci u odgovarajućem formatu.
Prioritet[1-5]	1
Izvor/Porijeklo	Programeri i testeri

Tablica 6: Funkcionalni zahtjev 6

Identifikator	FZ-6
Zahtjev	Sustav će omogućiti filtriranje prikazanih podataka.
Obrazloženje	Prikazani podaci moći će se filtrirati prema vidljivosti, metaatributima i članovima te prikazu stupaca.
Način provjere	Postaviti postavke za filtriranje te filtrirati. Prikazni podaci filtrirani su prema postavkama.
Prioritet[1-5]	1
Izvor/Porijeklo	Programeri i testeri

3.1. Dinamika realizacije zahtjeva

U inicijalnoj verziji softvera bit će realizirani sljedeći zahtjevi:

- **FZ-1** - Sustav će omogućiti dohvaćanje članova objekta.
- **FZ-2** - Sustav će omogućiti dohvaćanje vrijednosti svojstava objekta.
- **FZ-3** - Sustav će omogućiti drugačiji prikaz složenijih članova objekta.
- **FZ-4** - Sustav će omogućiti praćenje podobjekta trenutno promatranog objekta.

- **FZ-5** - Sustav će omogućiti izvoz prikazanih podataka u bilo kojem trenutku.
- **FZ-6** - Sustav će omogućiti filtriranje prikazanih podataka.

4. Nefunkcionalni zahtjevi

4.1. Izgled softvera

NFZ-1 - Sustav će interakciju s korisnikom provoditi preko grafičkog sučelja.

NFZ-2 - Sustav će pratiti formalan stil grafičkog sučelja.

4.2. Upotrebljivost softvera

NFZ-3 - Sustav ne smije ovisiti o konkretnim korisničkim aplikacijama čije će podatke analizirati.

NFZ-4 – Sustav će moći koristiti bilo koja osoba s dobrim poznavanjem .NET tehnologije.

NFZ-5 - Sustav će ponuditi mehanizme koji će smanjiti broj grešaka prilikom ulaska u podobjekte promatranog objekta.

NFZ-6 - Sustav će se moći naknadno proširiti za određene funkcije proizvoda.

4.3. Performanse sustava

NFZ-7 - Sustav će biti sposoban učinkovito izvoditi radnje ovisno o veličini promatranog objekta.

4.4. Izvođenje softvera i okruženje

NFZ-8 – Sustav treba raditi na računalima s instaliranim Windows 10 ili novijim Windows operacijskim sustavom.

4.5. Sigurnost i privatnost

Nema dodatnih nefunkcionalnih zahtjeva uz sigurnost i privatnost.

4.6. Ostalo

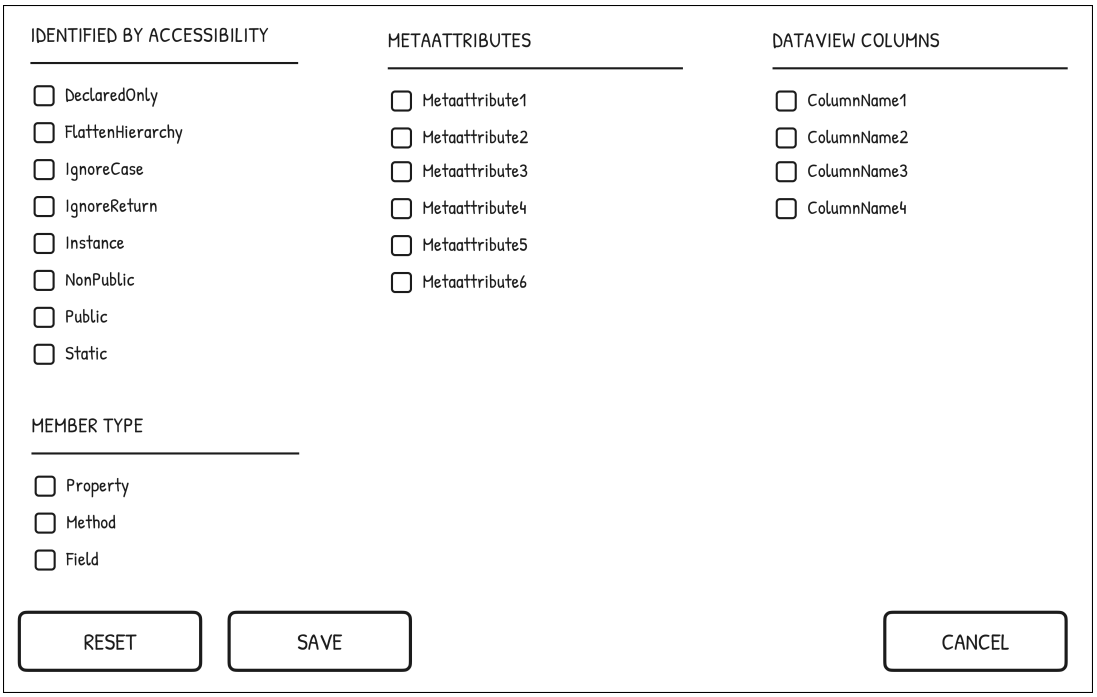
Nema dodatnih nefunkcionalnih zahtjeva.

5. Skice zaslona

5.1. Skica početne forme

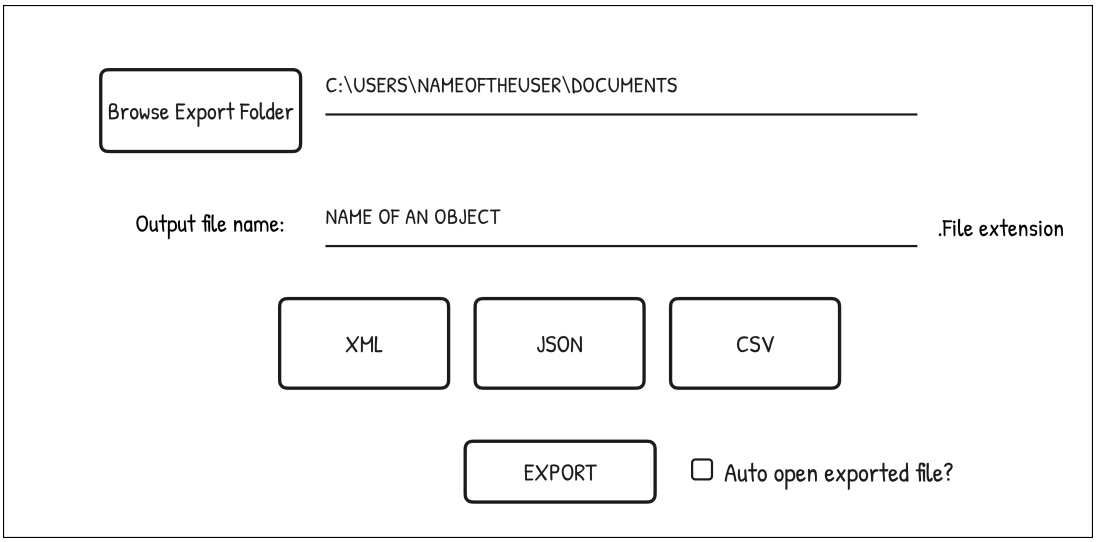
Slika 6: Prva skica (Izvor: autorski rad, 2023)

5.2. Skica forme za filtriranje



Slika 7: Druga skica (Izvor: autorski rad, 2023)

5.3. Skica forme za izvoz podataka



Slika 8: Treća skica (Izvor: autorski rad, 2023)

4.2.2. Dijagram korištenja

Vrlo je važno uz specifikaciju zahtjeva priložiti i dijagram koji opisuje slučajeve korištenja (engl. *Use Case*). Dijagram korištenja sastoji se od tri najvažnija dijela: slučaj korištenja, onaj tko ga koristi (engl. *Actor*) i sustav kojeg promatramo [12]. U nastavku slijedi slika dijagrama slučaja korištenja prema zahtjevima. Slučajevi korištenja najčešće su zahtjevi.

Slika 9: Dijagram slučaja korištenja (Izvor: autorski rad, 2023)

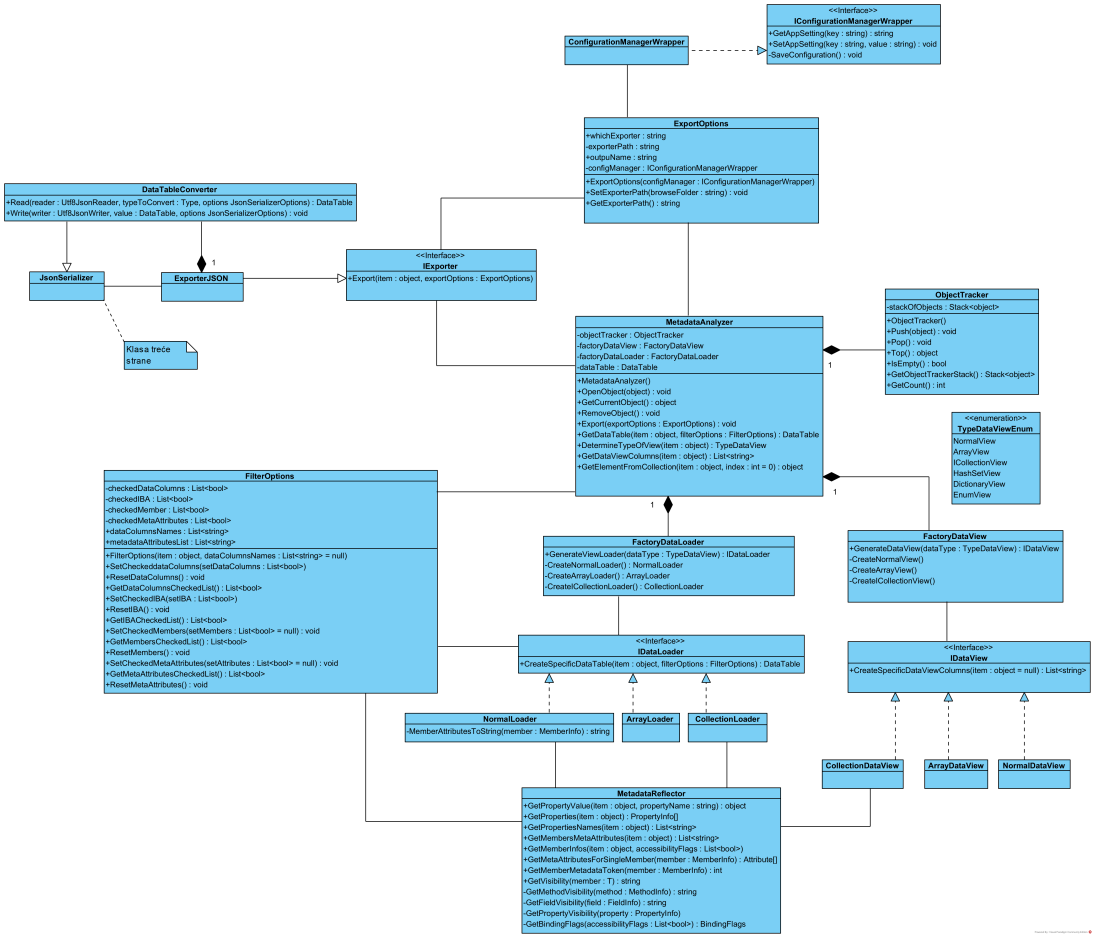
4.3. Dizajniranje i izrada artefakta

Dizajniranje softvera je proces kreativnog planiranja i oblikovanja programske aplikacije. To uključuje definiranje arhitekture, organizaciju komponenti i definiranje njihovih međusobnih odnosa kako bi se osigurala funkcionalnost, efikasnost i lakoća održavanja. Dobar dizajn softvera omogućava programerima da razvijaju visokokvalitetne aplikacije koje su skalabilne, prilagodljive i jednostavne za razumijevanje i proširenje tijekom vremena. Dizajniranje je vrlo teška aktivnost, ako se želi napraviti dobar, održiv i skalabilan softver. U ovom slučaju, potrebno je izraditi softver za očitavanje vrijednosti tijekom izvršavanja. Rješenje mora biti modularno, kako bi se softver kasnije mogao nadograditi. Ovdje nastupaju dizajnerski principi i uzorci dizajna te čisti kod kako bi se realiziralo buduće rješenje.

Što se tiče arhitekture softvera, ona nije toliko velika, zato što nema previše vanjskih

ovisnosti, osim jedne konfiguracijske datoteke. RED MetaAnalyzeru nije potrebna niti baza podatka ili mogućnost softvera za spajanje na internet, zato što ne treba dohvaćati ili slati podatke. Trenutnoj verziji softvera, takve funkcionalnosti još nisu potrebne. U nastavku slijedi dijagram klasa i isječci dijagrama klasa, pojašnjenja klasa, dijagrami slijeda i dijagram aktivnosti. Istaknuti su oni dijagrami koji se smatraju najvažnijim u korištenju i radu RED MetaAnalyzera.

4.3.1. Dijagram klasa



Slika 10: Sveukupni dijagram klasa (Izvor: autorski rad, 2023)

Dijagrami klasa jedna su od najosnovnijih vrsta dijagrama u UML-u (engl. *Unified Modeling Language*). Koriste se za prikazivanje statičkih odnosa softvera, drugim riječima, kako su stvari povezane. Prilokom izrade softvera, konstantno se donose dizajnerske odluke: koje klase sadrže reference na druge klase, koja klasa "posjeduje" neku drugu klasu i tako dalje. Dijagrami klasa pružaju način za prikazivanje "fizičke" strukture sustava [12]. Slijedi tablica s nazivima klasa i njihovom odgovornošću.

Tablica 7: Glavne klase i njihove odgovornosti

MetadataAnalzyer	Glavna klasa koja predstavlja ulaz u ostale funkcionalnosti RED MetaAnalyzera. Primjerice izvoz podataka.
ObjectTracker	Klasa služi za praćenje podobjekata trenutno promatranog objekta.
MetadataReflector	Pomoćna klasa za rad s refleksijom.
TypeDataViewEnum	Nije klasa, ali je enumeracija pomoću koje raspoznajemo koji objekt treba prikazati i kako.
FactoryDataLoader	Klasa koja nam na temelju TypeDataViewEnum daje objekt koji definira način učitavanja objekta.
NormalLoader	Daje odgovor za implementaciju učitavanja objekta trenutnog objekta u stogu za FactoryDataLoader klasu. Vraća se DataTable tip podatka. NormalLoader se koristi za bilo koju prilagođenu klasu. Prilagođena klasa je normalna klasa. Primjerice klasa Student koja ima svoja svojstva i metode.
ArrayLoader	Slična klasa kao i NormalLoader, ali se vraća DataTable za polja.
CollectionLoader	Klasa koja vraća DataTable, ali za objekte koji nasljeđuju sučelje ICollection.
FactoryDataView	Klasa koja također uz pomoć TypeDataViewEnum-a, vraća implementaciju metode za dobivanje imena stupaca za DataTable objekt.
NormalDataView	Klasa koja vraća listu tipa string s imenima stupaca za prilagođenu klasu.
ArrayDataView	Klasa koja vraća listu tipa string s imenima stupaca za polje.
CollectionDataView	Klasa koja vraća listu tipa string s imenima stupaca za objekte koji nasljeđuju sučelje ICollection. Primjerice lista, rječnik i slično.
FilterOptions	Klasa čija je odgovornost pamtiti koje postavke su složene za pretraživanje prikazanog objekta. Primjerice, koje stupce treba prikazati, koje objekte s kojim metaatributima.

Tablica 8: Glavne klase i njihove odgovornosti, nastavak

ConfigurationManagerWrapper	Klasa koja komunicira s vanjskom datotekom, konkretno s konfiguracijskom datotekom. Pomoću te klase u konfiguracijsku datoteku zapisuje se određena putanja za izvoz podataka. Klasa i čita zapisanu vrijednost.
ExportOptions	Klasa u koju se spremaju podaci vezani uz izvoz prikazanih podataka. Najčešće se izvozi DataTable.
ExporterJSON	Klasa koja sadrži implementaciju sučelja IExporter i izvozi podatke u JSON obliku.
DataTableConverter	Klasa koja definira kako bi se DataTable tip podatka trebao izvesti u JSON obliku.
JsonSerializer	Klasa treće strane koja služi za pretvaranje objekta u JSON.

Iz cijelog dijagrama klasa jasno se može uočiti kako većina klasa ne ukazuje na duboku međusobnu ovisnost. Izuzetak je MetadataAnalyzer klasa. Takvo stanje daje uvid u koncept labavog povezivanja (engl. *Loose Coupling*), ključnog za postizanje modularnosti i održivosti u dizajnu softverskih sustava.

Labavo povezivanje predstavlja strategiju dizajna koja teži smanjenju čvrste ovisnosti između različitih komponenti sustava. Ovo se postiže smanjivanjem direktnih veza i interakcija između klasa. Svrha ovog pristupa je stvaranje komponenata koje djeluju neovisno, gdje svaka komponenta obavlja svoje zadatke bez potrebe za dubokim poznavanjem drugih dijelova sustava. Ovime se postiže fleksibilnost, jer se promjene u jednoj komponenti rijetko šire na ostatak sustava.

Također, iz dijagrama klasa vidimo primjenjivanje nazivlja za imena klase, imena metoda i svojstva. Iz naziva možemo pretpostaviti koji tip podatka se dohvaća, a samo ime metode ukazuje najviše na funkcionalnost metode. Takvo pisanje programskog koda, unaprjeđuje održavljivost softvera i razumijevanje programske logike. Sve ovo što je opisano pripada pojmu, čisti kod (engl. *Clean Code*). Čisti kod promiče jasnoću, čitljivost i održavljivost softvera.

4.3.2. Primijenjeni uzorci dizajna

Uzorak dizajna sustavno definira, potiče i objašnjava sveobuhvatan dizajn koji rješava ponavljajući dizajnerski izazov u okviru objektno orijentiranih sustava. Ovaj uzorak opisuje problem, nudi rješenje, daje smjernice o situacijama u kojima se primjenjuje i istražuje njegove posljedice. Također pruža smjernice za implementaciju i donosi primjere. Rješenje koje proizlazi iz uzorka dizajna predstavlja općeniti skup objekata i klasa koji se primjenjuju za rješavanje spomenutog problema te se prilagođava i implementira u specifičnom kontekstu kako bi učinkovito rješavalo taj problem [13]. Postoje tri vrste uzoraka dizajna:

- **Kreatorski uzorci dizajna** - Uzorci dizajna koji se bave procesom stvaranja objekata na način koji omogućava fleksibilnost i optimizaciju. Uključuju: Singleton, Factory i ostali.

- **Strukturni uzorci dizajna** - Uzorci dizajna koji se bave kompozicijom objekata kako bi stvorili veće strukture ili cijele podatke. Uključuju: Adapter, Facade i ostali.
- **Uzorci dizajna ponašanja** - Uzorci dizajna koji se bave interakcijama između objekata i odgovornošću njihovih uloga. Uključuju: Observer, Strategy i ostali.

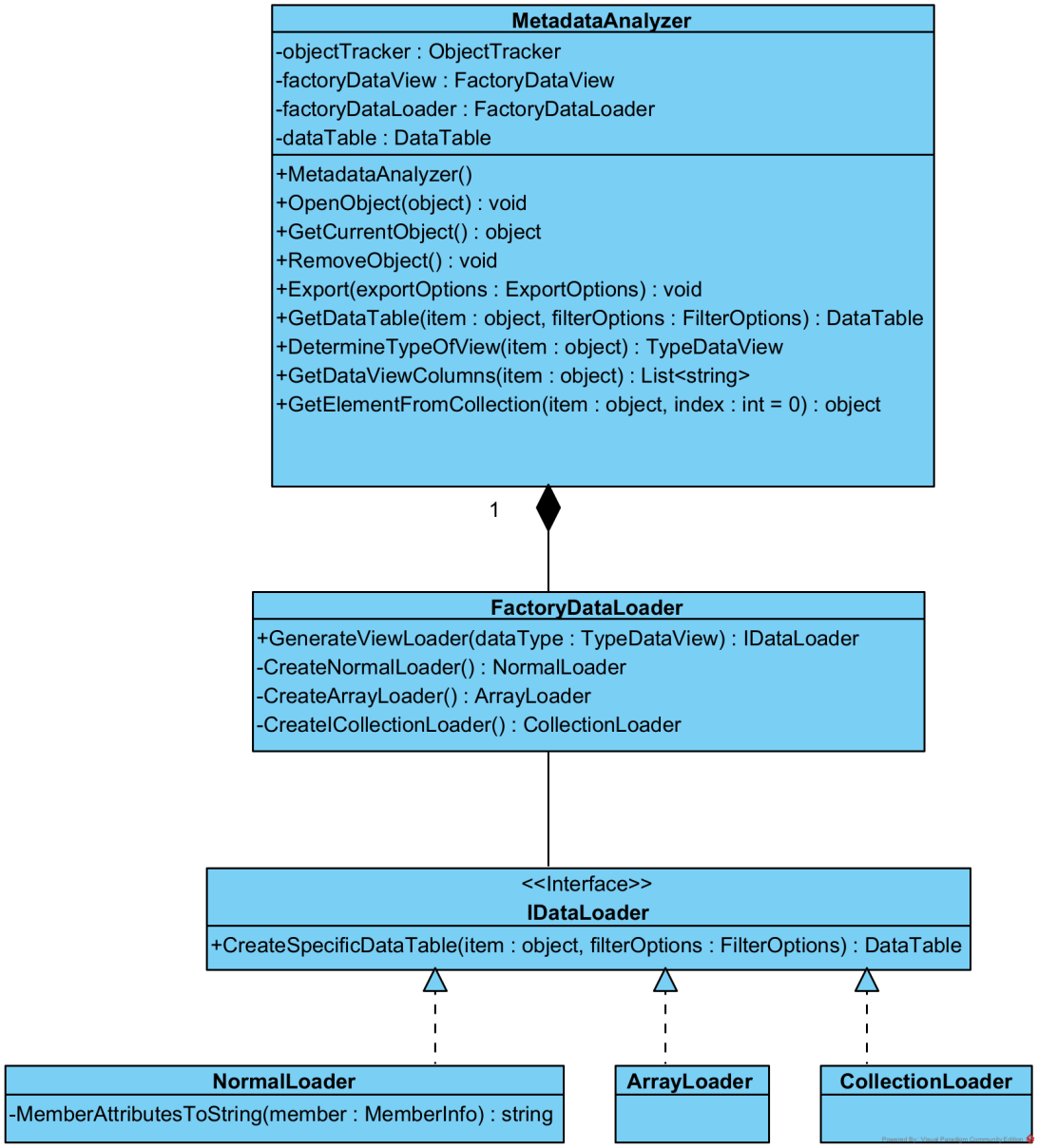
4.3.2.1. Facade

Uzorak dizajna Facade pruža jednostavno višestruko sučelje za podsustav, čime olakšava korisnicima interakciju s kompleksnim podsustavom. Ova fasada definira apstraktan sloj više razine koji pojednostavljuje upotrebu podsustava, prikrivajući njegovu detaljnu unutarnju implementaciju i kompleksnost. Koristeći Facade, klijentima se omogućuje lakša i manje ovisna interakcija s podsustavom, čime se povećava razina apstrakcije i pojednostavljuje korištenje kompleksnog sustava [13]. Jako dobar primjer fasade je prevoditelj. Isto možemo primijeniti i na MetadataAnalzyer klasu koja služi kao ulaz u kompleksnije sustave, primjerice izvoz prikazanih podataka u željeni oblik.

Poželjna situacija kada bi taj uzorak dizajna dobro došao je kada želimo pružiti jednostavno sučelje složenom podsustavu. Podsustavi često postaju složeniji kada se razvijaju . Većina uzoraka dizajna, kada se primijeni, rezultira većim brojem i manjim klasama. To čini podsustav više iskoristivim i lakšim za prilagodbu, ali postaje težak za korištenje klijentima koji ne trebaju prilagođavati softver. Fasada može pružiti jednostavan zadani prikaz podsustava koji je dovoljno dobar za većinu klijenata [13].

4.3.2.2. Factory

Factory uzorak dizajna osigurava sučelje za stvaranje objekata koji su međusobno povezani ili pripadaju istoj obitelji objekata, bez potrebe za preciznim specificiranjem njihovih konkretnih klasa. Taj uzorak dizajna idealno se primjenjuje kada je potrebno osigurati da sustav bude neovisan o načinu izrade ili prezentacije proizvoda. Ovaj uzorak dolazi do izražaja kada sustav podržava različite obitelji proizvoda, dopuštajući fleksibilnost u konfiguraciji sustava [13]. Konkretno u našem slučaju imamo dva Factoryja. Jedan je vezan za definiranje, bolje rečeno, dobivanje određenog prikaza pojedinog objekta. To je FactoryDataView klasa. Drugi Factory vezan je uz klasu FactoryDataLoader. Ta klasa dobiva od sučelja implementaciju o tome što se treba prikazati u DataGridViewu, a treba se prikazati DataTable kojeg radi NormalLoader, ArrayLoader ili CollectionLoader ovisno o tome kojeg je tipa trenutni objekt u stogu. Oni realiziraju sučelje IDataLoader. U nastavku slijedi slika strukture ovog uzorka dizajna i pune implementacije za Array tip podatka.



Slika 11: Uzorak dizajna: Factory (Izvor: autorski rad, 2023)

Implementacija sučelja IDataLoader:

```
1 public interface IDataLoader
2 {
3     DataTable CreateSpecificDataTable(object item, FilterOptions
          filterOptions);
4 }
```

Metoda GenerateViewLoader u FactoryDataLoader klasi:

```
1 public class FactoryDataLoader
2 {
```

```
3       public IDataLoader GenerateViewLoader(TypeDataViewEnum.
            TypeDataView dataType)
4       {
5           switch (dataType)
6           {
7               case 0:
8                    return CreateNormalLoader();
9
10              case (TypeDataViewEnum.TypeDataView)1:
11                  return CreateArrayLoader();
12
13              case (TypeDataViewEnum.TypeDataView)2:
14                   return CreateICollectionLoader();
15
16              default: throw new InvalidOperationException("There
                    is no such TypeDataView enumeration.");
17          }
18      }
19
20      private NormalLoader CreateNormalLoader()
21      {
22          return new NormalLoader();
23      }
24
25      private ArrayLoader CreateArrayLoader()
26      {
27          return new ArrayLoader();
28      }
29
30      private CollectionLoader CreateICollectionLoader()
31      {
32          return new CollectionLoader();
33      }
34  }
```

U GenerateViewLoader metodi vidimo kako na temelju TypeDataViewEnum određujemo koji Loader će se napraviti. Ako je dataType u "switchu" 1, a broj jedan označava ArrayView unutar TypeDataViewEnum enumeracije, stvorit će se ArrayLoader. Isti princip vrijedi i za FactoryDataView. Iz enumeracije se određuje koje stupce će DataGridView imati na glavnoj formi. U nastavku slijedi implementacija ArrayLoader klase koja ima samo jednu metodu CreateSpecificDataTable od sučelja.

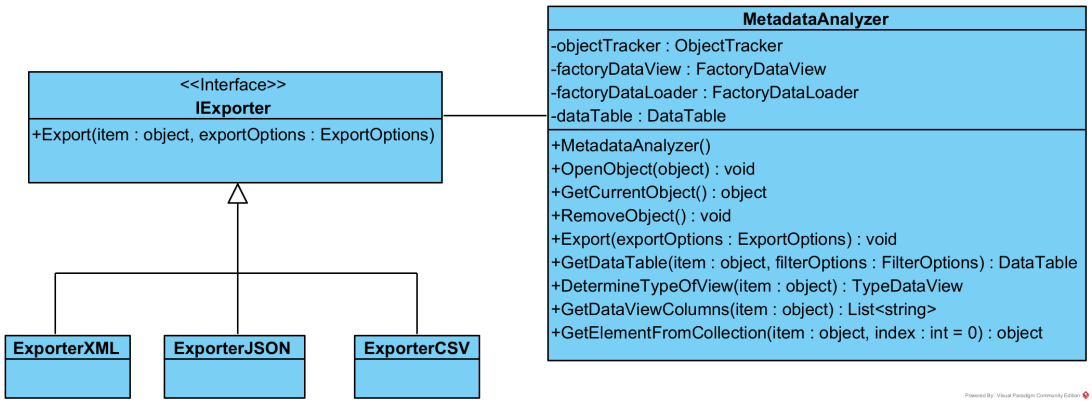
Metoda CreateSpecificDataTable u ArrayLoader klasi:

```
1  public class ArrayLoader : IDataLoader
2  {
3      public DataTable CreateSpecificDataTable(object item,
           FilterOptions filterOptions)
4      {
5          DataTable dataTable = new DataTable();
6          List<string> dataColumnsNames = filterOptions.
               dataColumnsNames;
7          List<bool> checkedDataColumns = filterOptions.
               GetDataColumnsCheckedList();
8
9          for (int i = 0; i < dataColumnsNames.Count; i++)
10         {
11             string columnName = dataColumnsNames[i];
12             dataTable.Columns.Add(columnName);
13         }
14
15         Array array = (Array)item;
16         int z = 0;
17         foreach (object element in array)
18         {
19             DataRow elementRow = dataTable.NewRow();
20             elementRow["Type"] = element.GetType();
21             elementRow["Index"] = z;
22             elementRow["Value"] = element;
23             z++;
24            dataTable.Rows.Add(elementRow);
25         }
26
27         for (int i = 0; i < checkedDataColumns.Count; i++)
28         {
29             if (!checkedDataColumns[i])
30             {
31                 dataTable.Columns.Remove(dataColumnsNames[i]);
32             }
33         }
34         return dataTable;
35     }
36 }
```

4.3.2.3. Strategy

Sljedeći uzorak dizajna koji je primijenjen u dizajnu rješenja je Strategy. Taj uzorak definira obitelj algoritama koji ih čini međusobno zamjenjivima. Uzorak Strategy omogućava da se algoritam mijenja neovisno o klijentima koji ga koriste. Često se naziva i Policy. Strategy uzorak često se koristi kada se mnoge povezane klase razlikuju samo u svom ponašanju. Strategije pružaju način za konfiguriranje klase s jednim od mnogih ponašanja. Na primjer, možemo definirati algoritme koji odražavaju različite kompromise između prostora i vremena. Strategije se mogu koristiti kada su ove varijante implementirane kao hijerarhija klasa algoritama [13].

U RED MetaAnalyzer to je realizirano prilikom izvoza prikazanih podataka. Možemo birati koji format izvoza želimo, na primjer, JSON, XML ili CSV. U trenutnoj verziji RED MetaAnalyzera implementiran je samo jedan algoritam za izvoz, a to je JSON. Upravo zbog same strukture Strategyja, vrlo brzo se dodaju i osposobe ostali algoritmi za drugi format izvoza kao što je XML ili CSV, jer pripadaju istoj obitelji algoritama i realiziraju isto sučelje IExporter. U nastavku slijedi slika strukture uzorka i cijela implementacija. Implementacija je vrlo kratka.



Slika 12: Uzorak dizajna: Strategy (Izvor: autorski rad, 2023)

Implementacija sučelja IExporter:

```
1 public interface IExporter
2 {
3     void Export(object item, ExportOptions exportOptions);
4 }
```

Metoda Export u MetadataAnalyzer klasi:

```
1 public void Export(ExportOptions exportOptions)
2 {
3     IExporter exporter = null;
4     switch (exportOptions.whichExporter)
5     {
6         case "JSON": exporter = new ExporterJSON(); break;
```

```
7           default: throw new ExporterException(exportOptions.
                whichExporter + " that exporter does not exist.");
8       }
9       exporter.Export(dataTable, exportOptions);
10  }
```

Ako bismo željeli implementirati novi format izvoza, recimo XML, sve što trebamo napraviti je stvoriti novu klasu ExporterXML koja nasljeđuje sučelje. Unutar metode Export definiramo kako će se DataTable izvesti. Vjerojatno će trebati napraviti novi konverter za XML format. Ujedno treba dodati novi slučaj unutar "switcha" i postaviti exporter na novi ExporterXML. Sva slova moraju biti velika, zato što se u formi za izvoz podataka postavlja varijabla whichExporter unutar exportOptions instance. U ovom primjeru možemo vidjeti kako u samo nekoliko koraka, možemo implementirati novi format za izvoz podataka.

Metoda Export u ExporterJSON:

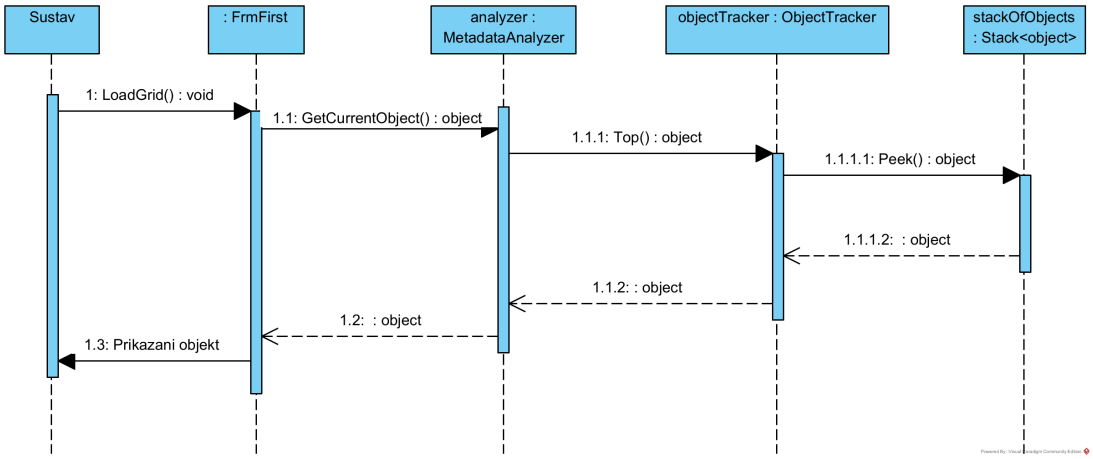
```
1  public void Export(object item, ExportOptions exportOptions)
2  {
3      var options = new JsonSerializerOptions
4      {
5          WriteIndented = true,
6          PropertyNamingPolicy = JsonNamingPolicy.CamelCase,
7          Converters = { new DataTableConverter() }
8      };
9
10     string exporterPath = exportOptions.GetExporterPath();
11
12     if (exportOptions.outputName == null) { throw new
           ExporterException("Output name of exporting object is null
           ."); }
13     if (exporterPath == null) { throw new ExporterException("
           Exporter path is null. Please set the folder where to
           export."); }
14
15     var jsonString = JsonSerializer.Serialize(item, options);
16     string filePath = Path.Combine(exporterPath, exportOptions.
           outputName);
17     File.WriteAllText(filePath, jsonString);
18 }
```

4.3.3. Opis interakcije elemenata rješenja

Interakcijski dijagrami definirani su od strane UML-a kako bi naglasili komunikaciju između objekata, a ne manipulaciju podacima povezanu s tom komunikacijom. Interakcijski dija-

grami usredotočuju se na određene poruke između objekata i kako te poruke surađuju kako bi ostvarile funkcionalnost. Dok sastavljeni dijagrami pokazuju koji se objekti slažu zajedno kako bi ispunili određenu zahtjevu, interakcijski dijagrami prikazuju točno kako će ti objekti to ostvariti [12].

Interakcijski dijagrami obično su povezani s elementima u sustavu. Na primjer, možemo imati interakcijski dijagram povezan s podsustavom koji pokazuje kako podsustav ostvaruje uslugu koju nudi na svojem javnom sučelju. Najčešći način povezivanja interakcijskog dijagrama s elementom je referenciranje interakcijskog dijagrama u napomeni pričvršćenoj za taj element. Detalje interakcije možemo prikazati koristeći nekoliko različitih notacija, međutim, dijagrami slijeda su daleko najčešći. Ostale notacije uključuju pregled interakcije, dijagrame komunikacije, vremenske dijagrame i tablice interakcija [12]. U nastavku su prikazane slike tri dijagrama i programski kod vezan uz važne funkcije koje objekti pozivaju.



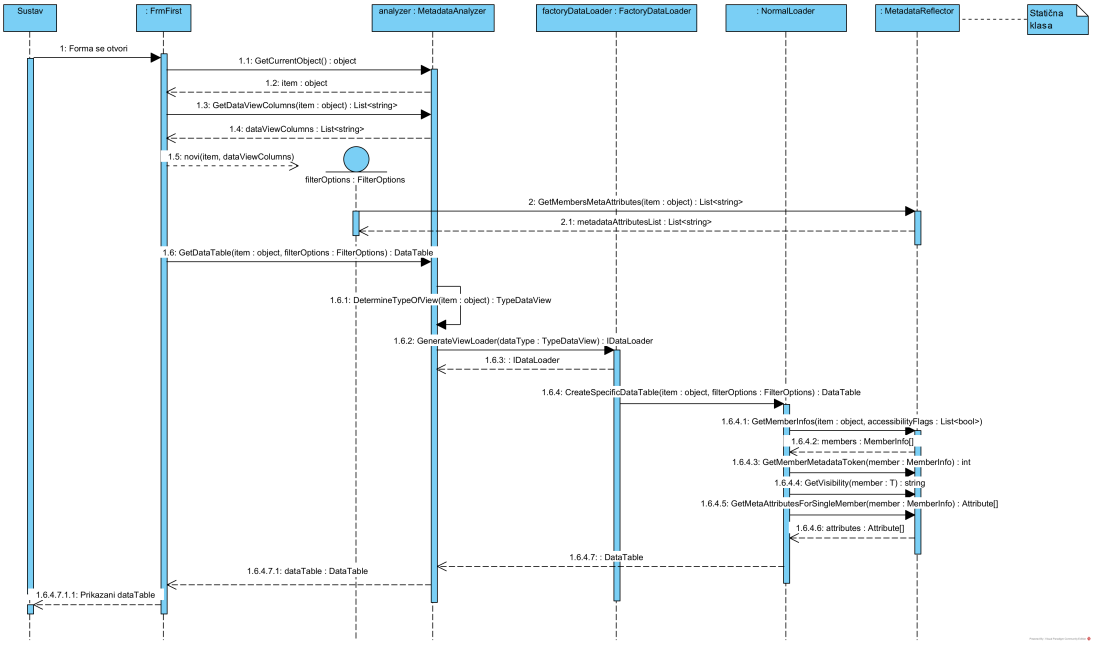
Slika 13: Dijagram slijeda za dohvaćanje objekta sa stoga (Izvor: autorski rad, 2023)

Metoda GetCurrentObject u MetadataAnalyzer klasi:

```
1 public object GetCurrentObject()
2 {
3     return objectTracker.Top();
4 }
```

Metoda Top u ObjectTracker klasi:

```
1 public object Top()
2 {
3     if (!IsEmpty())
4     {
5         return stackOfObjects.Peek();
6     }
7     else throw new StackException("Stack is empty!");
8 }
```



Slika 14: Dijagram slijeda za vraćanje DataTable objekta (Izvor: autorski rad, 2023)

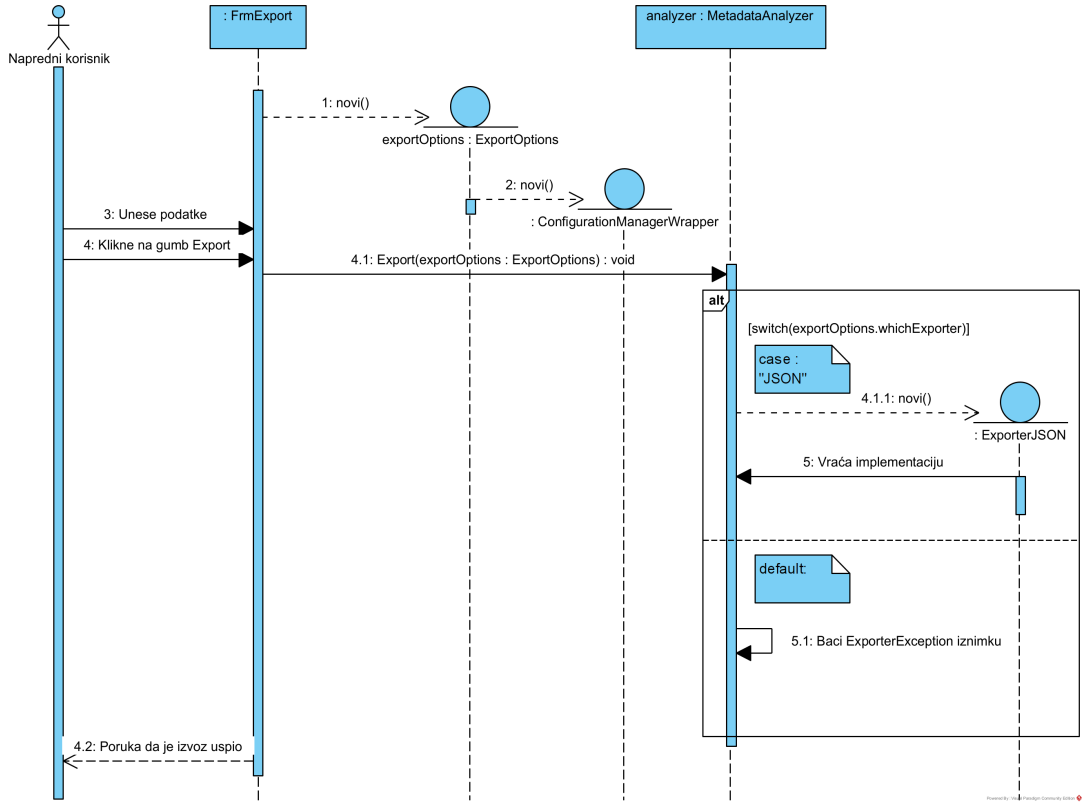
Metoda DetermineTypeOfView u MetadataAnalyzer klasi:

```
1  public TypeDataViewEnum.TypeDataView DetermineTypeOfView(object
       item)
2  {
3      TypeDataViewEnum.TypeDataView typeDataViewEnum =
           TypeDataViewEnum.TypeDataView.NormalView;
4
5      if (item.GetType().IsArray)
6      {
7              typeDataViewEnum = TypeDataViewEnum.TypeDataView.
                   ArrayView;
8      }
9      else if (item is ICollection)
10     {
11         typeDataViewEnum = TypeDataViewEnum.TypeDataView.
               ICollectionView;
12     }
13
14     return typeDataViewEnum;
15 }
```

Metoda GetMemberInfos u MetadataReflector klasi:

```
1  public static MemberInfo[] GetMemberInfos(object item, List<bool>
       accessibilityFlags)
```

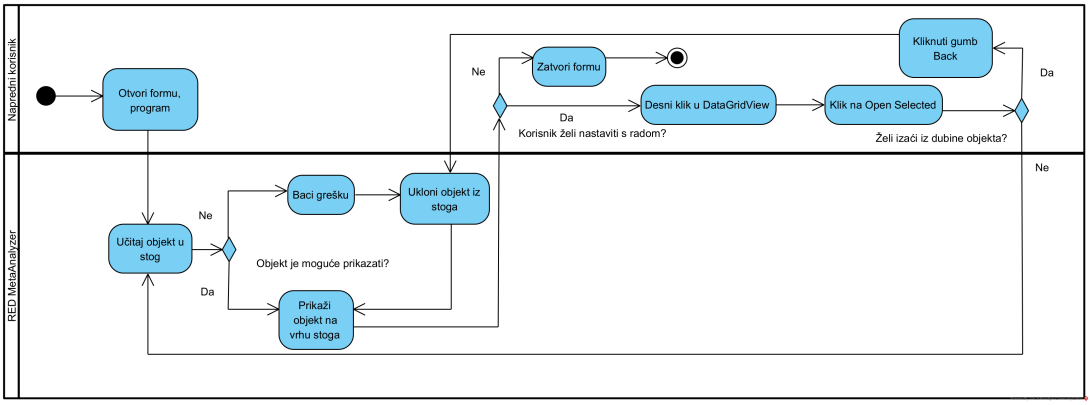
```
2  {
3      Type itemType = item.GetType();
4      var bindingFlags = GetBindingFlags(accessibilityFlags);
5      MemberInfo[] memberInfos = itemType.GetMembers(bindingFlags);
6      return memberInfos;
7  }
```



Slika 15: Dijagram slijeda za izvoz prikazanih podataka (Izvor: autorski rad, 2023)

Uz navedene dijagrame slijeda, važni su i dijagrami aktivnosti koji su usmjereni na izvršenje i tijek ponašanja sustava, a ne kako je sustav sastavljen. Vjerojatno više od bilo kojeg drugog UML-a dijagrama, dijagrami aktivnosti primjenjuju se na i na ostalim područjima, a ne samo kod softverskog modeliranja. Primjenjivi su skoro na sve vrste modeliranja ponašanja. Na primjer, poslovne procese, softverske procese ili tijekove rada [12].

Kada se koriste za softversko modeliranje, aktivnosti obično predstavljaju ponašanje kao rezultat poziva metode. Kada se koristi za poslovno modeliranje, aktivnosti mogu pokrenuti vanjski događaji, kao što je izdavanje narudžbe, ili interni događaj, kao što je mjerač vremena za pokretanje nekog procesa unutar određenog odjela [12]. Slijedi slika jednostavnog dijagrama aktivnosti koji pokazuje rad korisnika i ulaska u dubinu objekata u RED MetaAnalyzeru.



Slika 16: Dijagram aktivnosti za osnovni rad s RED MetaAnalyzerom (Izvor: autorski rad, 2023)

5. Demonstracija i evaluacija artefakta

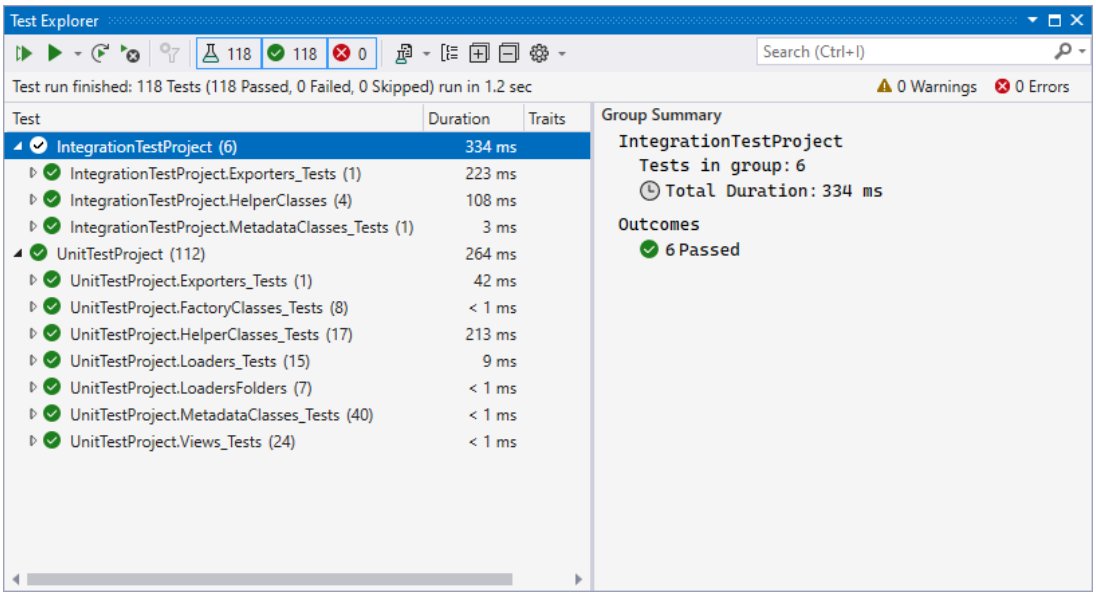
U znanstvenom oblikovanju u programskom inženjerstvu, demonstracija i evaluacija artefakta predstavljaju ključni korak u procesu razvoja i provjere softverskih rješenja. Ovaj proces uključuje prezentaciju i praktično testiranje stvorenog artefakta, poput softverskog alata ili sustava, kako bi se utvrdila njegova funkcionalnost, korisničko iskustvo i učinkovitost. Evaluacija artefakta uključuje analizu njegovih prednosti, nedostataka i potencijalnih unaprjeđenja. Ova faza igra ključnu ulogu u osiguravanju da artefakt odgovara specifičnim zahtjevima i ciljevima projekta te da pruža vrijednost korisnicima.

5.1. Testiranje rješenja

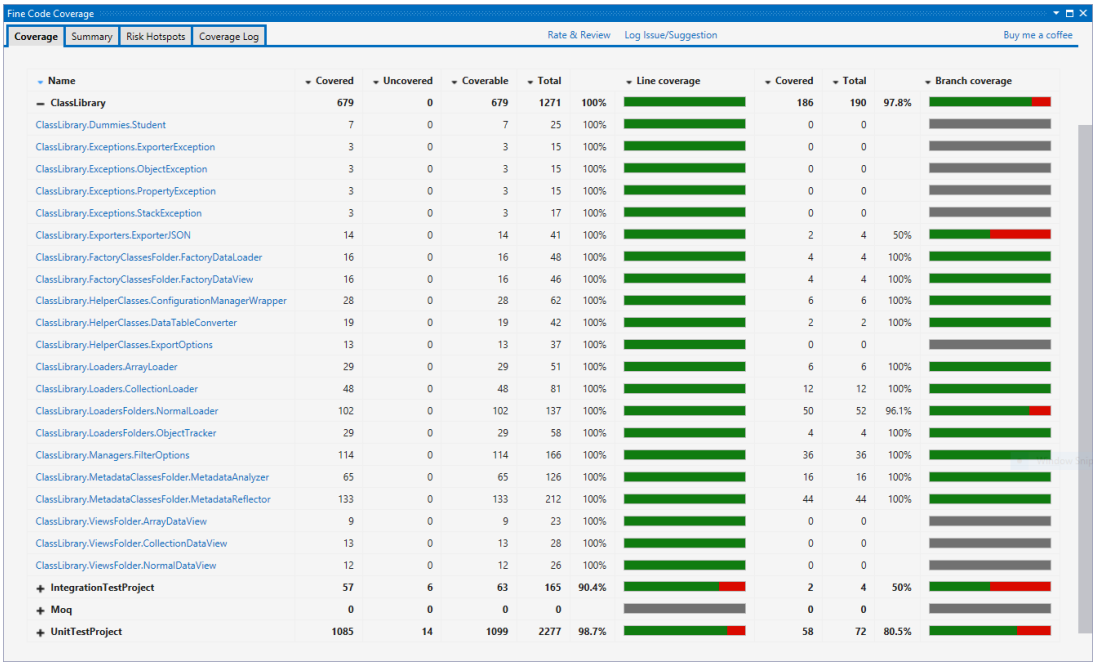
Testiranje programskog koda izuzetno je važno, zato što omogućava otkrivanje pogrešaka, propusta i nepravilnosti u softveru prije nego što ga pustimo u stvarno okruženje ili korisnicima. To pomaže osigurati da softver radi kako je zamišljen i da se stabilno ponaša pod različitim uvjetima. Testiranje istovremeno doprinosi poboljšanju kvalitete softvera, smanjuje rizik od neočekivanih problema i olakšava održavanje i buduće izmjene koda. Kroz taj proces, programeri mogu biti sigurni da njihova rješenja ispravno i pouzdano funkcioniraju, što pridonosi zadovoljstvu korisnika i razvojnog tima.

Testiranje RED MetaAnalyzera odvijalo se tek kada je implementacija bila završena. Za testiranje softvera koristio se MSTest i biblioteka Moq koja omogućuje korištenje klase Mock. Klasa Mock pomaže u stvaranju lažnih ovisnosti kako bi se programski kod izvršio onako kako je zamišljen, ali ne s pravim, realnim ovisnostima, jer takvi testovi mogu narušiti podatke u vanjskim ovisnostima. Primjerice, u konfiguracijskoj datoteci promijeni se vrijednost nekog ključa zbog testa. To uvelike može oštetiti produkcijski dio koda. Ujedno, vrlo je važno testove učiniti čitljivima. Čitljivost omogućuje testerima koje se metode koriste, što je postavljeno za test i koja je očekivana vrijednost. Kroz testiranje RED MetaAnalyzera, koristit će se AAA (engl. *Triple A*) pristup, Arrange, Act i Assert. U prvom dijelu, Arrange, postavljamo okolinu za izvođenje testa. U drugom, Act, dijelu izvršavamo metodu koja se u tijeku testiranja, a u zadnjem dijelu, Assert, provjeravamo je li vraćeni rezultat onakav kako je zamišljen. Ako nije, test je pao, a ako je, test je prošao.

Sveukupno je napravljen veliki broj testnih slučajeva, sveukupno 118 testova. Konkretno je napravljeno 112 jediničnih testova i 6 integracijskih testova. Većina integracijskih testova odnosi se na izvoz podataka i upravljanje konfiguracijskim datotekama. Ovim testnim slučajevima pokriva se cijela programska logika unutar projekta ClassLibrary. Grafičko sučelje nije testirano. U nastavku slijede slike Test Expolorera, pokrivenosti koda s Fine Code Coverage ekstenzijom unutar Visual Studija i dva primjera testa. Jedan primjer je jedinični test, a drugi, integracijski test.



Slika 17: Test Explorer pregled (Izvor: autorski rad, 2023)



Slika 18: Fine Code Coverage pregled (Izvor: autorski rad, 2023)

Primjer jediničnog testa CreateSpecificDataTable metode ArrayLoader klase:

```
1  [TestMethod]
2  public void
       CreateSpecificDataTable_TypeColumnDisabled_TypeDoesntExist()
3  {
4      //arrange
5      ArrayLoader loader = new ArrayLoader();
6
```

```
7       int[] item = new int[2];
8       item[0] = 0;
9       item[1] = 1;
10
11      MetadataAnalyzer metadataAnalyzer = new MetadataAnalyzer();
12      FilterOptions filterOptions = new FilterOptions(item,
            metadataAnalyzer.GetDataViewColumns(item));
13      filterOptions.SetCheckedDataColumns(new List<bool> { false,
            true, true });
14
15      //act
16      var expected = loader.CreateSpecificDataTable(item,
            filterOptions);
17
18      //assert
19      Assert.IsFalse(expected.Columns.Contains("Type"));
20  }
```

Primjer integracijskog testa Export metode MetadataAnalzyer klase:

```
1   [TestMethod]
2   public void Export_ExporterExportedJsonFile_JsonFileExists()
3   {
4       //arrange
5       object item = new Student()
6       {
7           FirstName = "Test"
8       };
9       string projectDirectory = Directory.GetParent(Directory.
            GetParent(AppDomain.CurrentDomain.BaseDirectory).FullName)
            .FullName;
10      string exportPath = Path.Combine(projectDirectory, "
            TestExportPath");
11      var configManagerMock = new Mock<IConfigurationManagerWrapper
            >();
12      configManagerMock.Setup(cm => cm.GetAppSetting("SelectedPath"
            )).Returns(exportPath);
13
14      if (!Directory.Exists(exportPath))
15      {
16          Directory.CreateDirectory(exportPath);
17      }
18
```

```
19      ExportOptions exportOptions = new ExportOptions(
            configManagerMock.Object) { outputName = "
            testOutputAnalyzer.json" , whichExporter = "JSON"};
20      MetadataAnalyzer analyzer = new MetadataAnalyzer();
21      analyzer.OpenObject(item);
22
23      //act
24      analyzer.Export(exportOptions);
25
26      //assert
27      string expectedFilePath = Path.Combine(exportPath,
            exportOptions.outputName);
28      Assert.IsTrue(File.Exists(expectedFilePath));
29  }
```

5.2. Demonstracija artefakta

5.2.1. ZMG Desktop

ZMG Desktop je aplikacija koja olakšava rad i praćenje poslovanja privatnog obrta. Aplikacija je vrlo interaktivna i ovisi o radnjama korisnika. Korisniku omogućuje stvaranje računa na kojem se nalaze stavke utrošenog materijala i usluge. Osim kreiranja računa, korisnik ima uvid u popis klijenata gdje za svakog klijenta može prikazati popis radnih naloga i izdane račune. ZMG Desktop omogućuje poslodavcu izdavanje izvještaja za radne naloge i popis klijenata.

Za korištenje RED MetaAnalyzera, potrebno je uključiti DLL ClassLibrary i izvršnu datoteku RED MetaAnalyzera u projekt koji je postavljen kao početni te dodati sljedeće postavke u konfiguracijsku datoteku App.config.

```
1  <appSettings>
2    <add key="SelectedPath" value="" />
3  </appSettings>
```

Ujedno, potrebno je omogućiti otvaranje početne forme RED MetaAnalyzera unutar ZMG Desktopa s proslijeđenim objektom koji se želi analizirati. U trenutnom slučaju, pregledava se lista računa. Za demonstraciju, RED MetaAnalyzer otvara se pomoću pritiska tipke F11 na formi za račune. Slijedi isječak programskog koda za inicijalizaciju početne forme RED MetaAnalyzera na pritisak tipke.

```
1  // Za zeljenu formu potrebno je ukljuciti dogadaj za pritisak
       tipke
2  private void Form1_KeyDown(object sender, KeyEventArgs e)
3  {
4      if (e.KeyCode == Keys.F11)
5      {
```

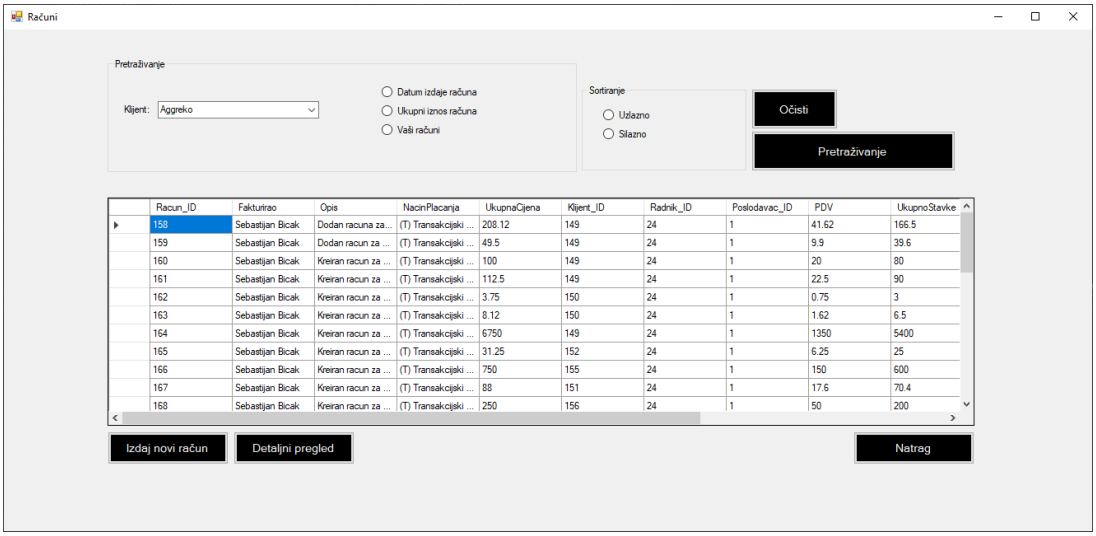
```
6           REDMetaAnalyzerHomeForm metaAnalyzerHomeForm = new
                REDMetaAnalyzerHomeForm(racunServis.DohvatiSveRacune()
                );
7           metaAnalyzerHomeForm.Show();
8       }
9   }
```

Demonstracija artefakta za aplikaciju ZMG Desktop, uklopit će analiziranje liste računa te ulazak u dubinu objekta. U ovom slučaju, to je lista pomoću koje će se pregledavati neki račun iz te liste računa. Također, bit će prikazan isječak JSON izvoza. Postupak demonstracije je sljedeći:

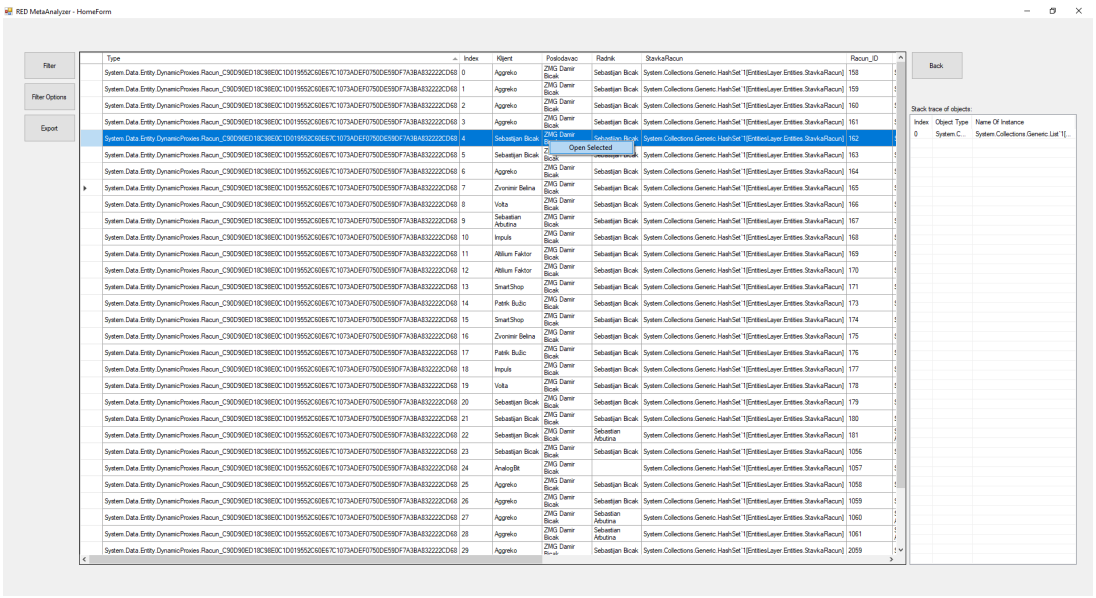
- **Korak 1** - Otvoriti ZMG Desktop i ulogirati se.
- **Korak 2** - Otvoriti formu gdje su računi.
- **Korak 3** - Otvoriti RED MetaAnalyzer s definiranim postavkama za njegovo otvaranje početne forme.
- **Korak 4** - Odabrati željeni račun.
- **Korak 5** - Kliknuti na gumb Export i popuniti formu.
- **Korak 6** - Kliknuti ponovo na gum Export, ali na formi za izvoz.

Nakon svih koraka, u željenoj mapi, trebao bi biti izvoz podataka u JSON formatu.

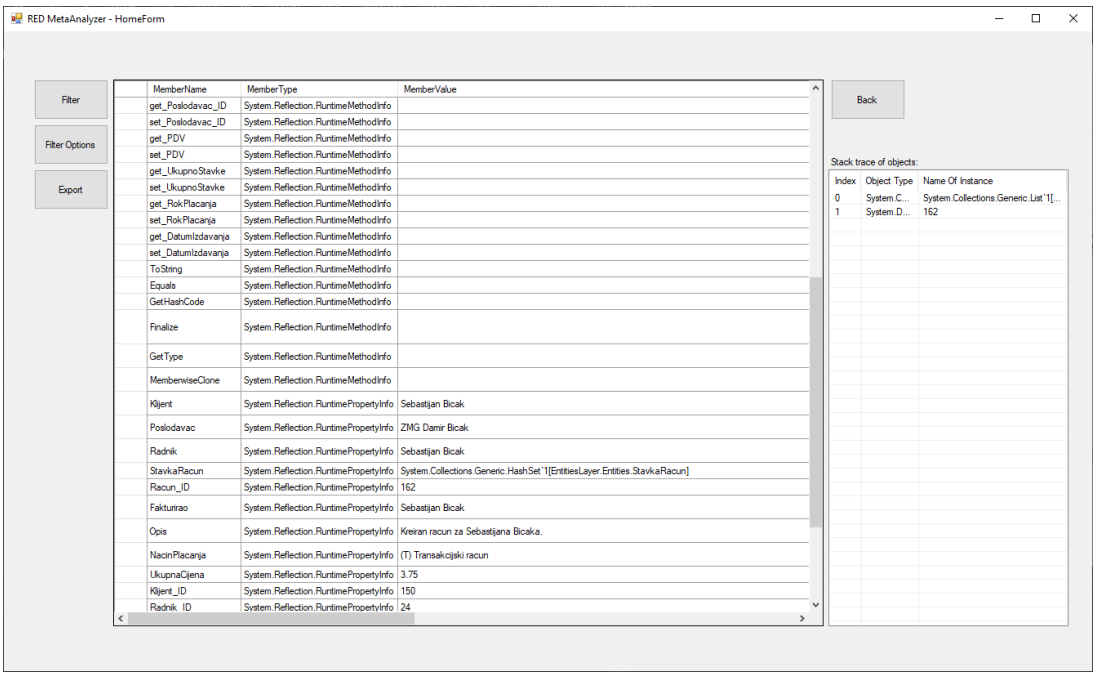
Slika 19: Glavni izbornik ZMG Desktop aplikacije (Izvor: autorski rad, 2023)



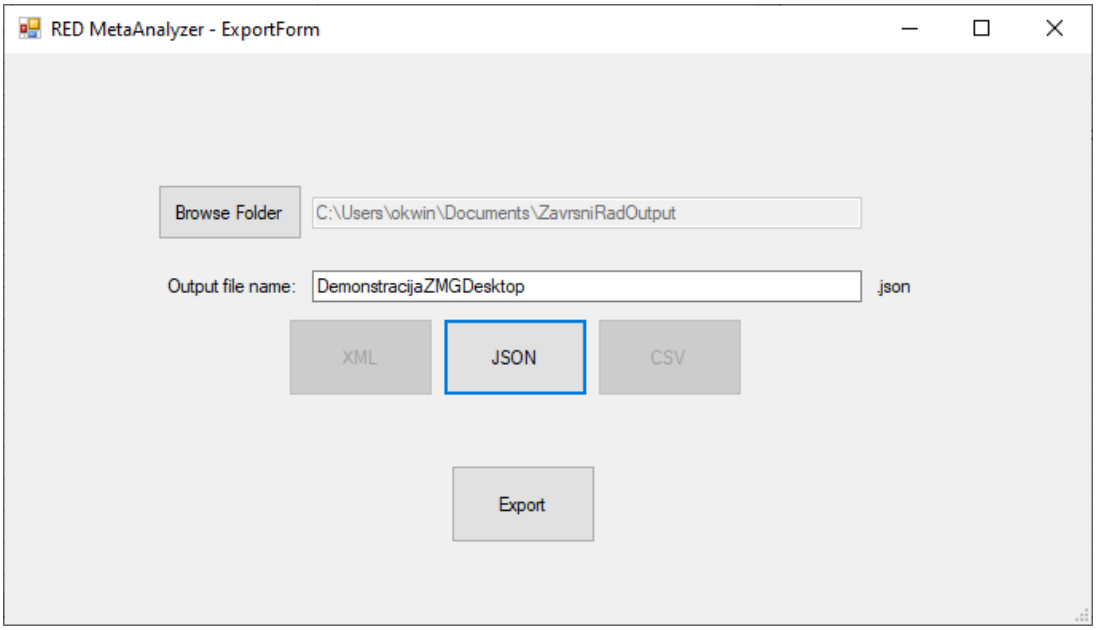
Slika 20: Prikaz računa u ZMG Desktopu (Izvor: autorski rad, 2023)



Slika 21: Prikaz početne forme RED MetaAnalyzera (Izvor: autorski rad, 2023)



Slika 22: Prikaz odabranog računa (Izvor: autorski rad, 2023)



Slika 23: Prikaz ispunjene forme za izvoz (Izvor: autorski rad, 2023)

```
[
  ...
  {
    "MemberName": "Poslodavac",
    "MemberType": "System.Reflection.RuntimePropertyInfo",
    "MemberValue": "ZMG Damir Bicak",
    "MemberMetaDataAttributes": "",
    "MemberVisibility": "public",
    "MemberMetaDataToken": "385875973"
  },
  {
    "MemberName": "Radnik",
    "MemberType": "System.Reflection.RuntimePropertyInfo",
    "MemberValue": "Sebastijan Bicak",
    "MemberMetaDataAttributes": "",
    "MemberVisibility": "public",
    "MemberMetaDataToken": "385875974"
  },
  {
    "MemberName": "StavkaRacun",
    "MemberType": "System.Reflection.RuntimePropertyInfo",
    "MemberValue": "System.Collections.Generic.HashSet\u00601[
       EntitiesLayer.Entities.StavkaRacun]",
    "MemberMetaDataAttributes": "",
    "MemberVisibility": "public",
    "MemberMetaDataToken": "385875975"
  }
  ...
]
```

6. Zaključak

U ovom radu pokazali smo što je metaprogramiranje, tehniku programiranja pomoću koje možemo dobiti uvid u metapodatke objekata. Primjerice, kako iz klase Student prikazati njezina svojstva i metode. Metaprogramiranje nije samo vezano uz analizu objekata i njegovih članova, nego se ono može iskoristiti i za generiranje koda. Pošto je naglasak u radu na .NET tehnologiji, generiranje koda u toj tehnologiji odvija se pomoću T4, CodeDOM-a i novog Microsoftovog prevoditelja, Roslyn. Svaki od njih na jedinstven način može generirati dinamički programski kod i omogućiti programskim inženjerima i programerima bolji osvrt na izvršavanje koda.

Jedna od većih neugoda u svijetu programiranja je otežano pregledavanje i očitavanje vrijednosti softvera tijekom softvera. To rezultira otežanoj analizi podataka softverskog sustava, zato što je većina programa apstrahirano od grafičkih sučelja. Iza takvih sučelja nalaze se veoma kompleksni sustavi koji svaki od njih ima svoje funkcionalnosti. Jedan od najvećih problema takvih sustava je održavanje i već spomenuta otežana analiza. Stoga je zadatak bio izgraditi generičko softversko rješenje koje daje uvide u vrijednosti svojstava i imena svojstava te metoda.

Generičko softversko rješenje izrađeno je pomoću metodologije koja se sve češće i češće upotrebljava u informacijskim sustavima i programskom inženjerstvu. Naziv takve metodologije je znanstveno oblikovanje. Bit takve metodologije je izrada artefakata u promatranom kontekstu te kako bi interakcija između aktivnosti dizajniranja i aktivnosti istraživanja dala odgovor na određene probleme u odgovarajućem kontekstu. Znanstveno oblikovanje možemo primijeniti u softverskom inženjerstvu. Radni okvir znanstvenog oblikovanja prati procese izrade softvera uz veće ili manje promjene, ovisno o kontekstu gledanja problema.

RED MetaAnalyzer je artefakt, softverski proizvod koji je rezultat praćenja metodologije znanstvenog oblikovanja. Dizajn softverskog rješenja poštuje principe čistog koda i labavog povezivanja, što ga čini održavljivim i skalabilnim. Generički dizajn omogućuje jednostavno dodavanje novih algoritama i modularne prikaze podataka. Softver u potpunosti ispunjava sve definirane funkcionalne zahtjeve, omogućuje pregled članova objekta, dohvaćanje vrijednosti, te podržava filtriranje i izvoz podataka. Unatoč svojoj funkcionalnosti, korisničko sučelje je intuitivno, ali zastarjelo. U budućim verzijama softvera preporučuje se razmotriti korištenje modernijih tehnologija za korisničko sučelje, kao što je WPF. Osim toga, planira se proširenje algoritama za izvoz podataka i dodavanje pravila za prikaz drugih složenih tipova podataka. Također, razmatra se uvođenje novih funkcionalnosti koje će biti korisne programerima i testerima. Na temelju ove evaluacije, artefakt je pokazao uspješno izrađeno softversko rješenje.

Popis literature

[1] R. J. Wieringa, *Design science methodology for information systems and software engineering*. Heidelberg, Njemačka: Springer, 2014.

[2] I. Sommerville, *Software Engineering, Global Edition*. Harlow, Essex, UK: Pearson Education, 2016.

[3] M. Mijač, A. García-Cabot i V. Strahonja, „Reactor Design Pattern," *TEM Journal*, sv. 10, br. 1, str. 18–30, veljača 2021., [Na internetu]. Dostupno: TEM Journal, https://www.temjournal.com/ [pristupano 23.7.2023.].

[4] Arvind Padmanabhan, "Metaprogramming.", 2021. [Na internetu]. Dostupno: https://devopedia.org/metaprogramming [pristupano 13.8.2023.].

[5] "Metaprogramming", (bez dat.) [Na internetu]. Dostupno: https://cs.lmu.edu/~ray/notes/metaprogramming/ [pristupano 13.8.2023.].

[6] K. Hazzard i J. Bock, *Metaprogramming in .NET*. Shelter Island, Suffolk County, SAD: Manning, 2013.

[7] Margaret Rouse, "What is Declarative Programming?", 2020. [Na internetu]. Dostupno: https://www.techopedia.com/definition/18763/declarative-programming [pristupano 13.8.2023.].

[8] Microsoft, "System.Reflection Namespace", (bez dat.) [Na internetu]. Dostupno: https://devopedia.org/metaprogramming [pristupano 13.8.2023.].

[9] Microsoft, "Desktop Guide (Windows Forms .NET)", 2023. [Na internetu]. Dostupno: https://learn.microsoft.com/en-us/dotnet/desktop/winforms/overview/?view=netdesktop-7.0 [pristupano 13.8.2023.].

[10] Microsoft, "Visual Studio Community", (bez dat.) [Na internetu]. Dostupno: https://visualstudio.microsoft.com/vs/community/ [pristupano 13.8.2023.].

[11] „IEEE Recommended Practice for Software Requirements Specifications," *IEEE Std 830-1998*, str. 1–40, 1998., [Na internetu]. Dostupno: IEEE Xplore, https://ieeexplore.ieee.org/document/720574 [pristupano 13.9.2023.].

[12] D. Pilone i N. Pitman, *UML 2.0 in a Nutshell: A Desktop Quick Reference*. Sebastopol, California, SAD: O'Reilly Media, 2005.

[13] E. Gamma, R. Helm, R. Johnson i J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Professional, 1995.

Popis slika

Popis tablica