UDC: 519.77:519.85./87 INFO M: str. 44-52

GENERATORI PROGRAMSKOG KODA: RAZVOJ I PRAKTIČNA PRIMENA UPOTREBOM .NET PLATFORME
CODE GENERATORS: DEVELOPMENT AND PRACTICAL APPLICATION USING .NET PLATFORM

Saša D. Lazarević, Stefan Z. Mitić,
Univerzitet u Beogradu, Fakultet organizacionih nauka

REZIME: Generisanje programskog koda predstavlja proces gde se, pomoću definisane strukture ulaznih podataka i logike njihove obrade, kao rezultat rada dobija izvršni programski kod. Prikazani su principi na kojima počiva automatsko generisanje koda, najčešće i najefikasnije metode koje se koriste u procesu generisanja. Ukazano je na ključne činioce procesa generisanja programskog koda. Takođe, prikazano je nekoliko različitih vrsta generatora koda, njihove prednosti, mane i praktične primene. Kao konačni rezultat analize, dat je primer rešenja u vidu prototipa aplikacija koje, kao rezultat obrade ulaznih struktura podataka, daju programski kod na odabranom jeziku.Takav kod može biti tretiran kao deo buduće programske logike, ili kao samostalna logička celina.

KLJUČNE REČI: generator programskog koda, metaprogramiranje, XSLT, CodeDom, refleksija, XML

ABSTRACT: Code generation is a process which, with defined structure of input data and their processing logic, provides results in form of executable code. Our research has the task to present the principles underlying the automatic code generation, the most effective and common methods that are used as the basis of programming logic in the code generation process. We also present several different types of code generators, their advantages, disadvantages and practical application. As a final result of the analysis we provide solutions in the form of application prototypes that, as a result of processing the input data structure, can generate code in provided programming language. Such code can be treated as a part of future programming logic, or as an independent logic unit.

KEY WORDS: code generator, metaprograming, XSLT, CodeDom, reflection, XML

1. UVOD

Generisanje programskog koda predstavlja konstruisanje aplikacija koje su sposobne da, na osnovu ulaznih podataka, kao proizvod svog rada, daju programski kod koji predstavlja novu aplikaciju ili njen deo. Kao jedan od najnaprednijih aspekata softverskog inženjerstva, automatsko generisanje programskog koda nudi veoma značajne olakšice u procesu razvoja softvera, koje se ogledaju u značajnoj uštedi vremena u razvoju, primeni standardizovanih rešenja, jednostavnoj implementaciji modularnosti u razvoju aplikacija i minimizaciji grešaka.

Najčešći razlog za korišćenje (i pravljenje) sistema koji se bave generisanjem programskog koda predstavlja potencijalna ušteda u vremenu koja se njihovim korišćenjem ostvaruje. Ušteda se najviše ostvaruje u početnim fazama razvoja softvera, kada je neophodno napraviti osnovu za rad same aplikacije, na koju će se kasnije nadovezivati drugi elementi logike softvera.

Standardizovana softverska rešenja predstavljaju rešenja koja su se dobro pokazala u praksi, i kao takva poželjno ih je integrisati u svaki deo aplikacije, gde je to moguće. Uvek je dobro to učiniti na samom početku razvoja, ali njihova naknadna implementacija mora biti moguća. Generisanjem koda može se svako od ovih rešenja implementirati lako, bez bojazni da će neki segment aplikacije biti preskočen.

Najčešći tip generatora programskog koda je *Brute force* tip, odnosno generator koji na osnovu ulaznih podataka vrši prostu manipulaciju tekstualnim podacima. Drugi popularan tip generatora svoj rad zasniva na parsiranju ulaznih podataka pomoću odgovarajućih šablona. Takvi generatori za svoj rad najčešće koriste XSLT, T4 ili neke druge tehnologije koje omogućavaju razvoj i manipulaciju šablonima. Izbor programskog jezika za logiku generatora koda i generisani kod diktira vrstu šablona koja će se koristiti [1].

Istraživanje prikazuje tehnologije koje se najčešće koriste u procesu automatskog generisanja softverskih rešenja, kao i prikaz različitih tipova generatora programskog koda. Obuhvaćen je i proces razvoja prototipova generatora koda, korišćenjem prethodno opisanih tehnologija.

2. GENERATORI PROGRAMSKOG KODA

Generatori programskog koda mogu se definisati kao programi koji, na osnovu određenih ulaznih podataka i logike za njihovu obradu, pišu druge programe [2]. Ono što generatore programskog koda čini primamljivim jesu potencijalno jako velike vremenske uštede, koje se mogu ostvariti njihovom primenom, naročito u početnim fazama razvoja aplikacija [2]. U zavisnosti od načina implementacije jednog takvog sistema, moguće je u kasnijim fazama razvoja implementirati nova rešenja, bez potencijalnih teškoća i problema, koji mogu nastati prilikom masovnog prepravljanja softvera.

Tokom našeg istraživanja uočili smo jednu zajedničku osobinu svih generatora: neophodno je iz aplikacije koja se generiše razdvojiti ono što je generičko od onog što je varijabilno. Konstantni deo aplikacije se u generatoru predstavlja pomoću šablona, koji se upotpunjuje podacima dobijenim preko ulaznih parametara, koji predstavljaju varijabilni deo aplikacije [2].

Da bi se jedna takva aplikacija uspešno razvila, potrebno je u obzir uzeti veliki broj faktora: platformu za koju se generator koda piše, izabrani programski jezik za generator i

rezultujući kod, generisanje koda na nivou aplikacije ili aplikacionog sloja, mogućnost ručnog menjanja generisanih datoteka, odvajanje generisanog koda od koda kojim neposredno upravlja programer. Značajan faktor predstavlja definisanje standardnog ulaza, na osnovu kog će se, primenom izabrane metode obrade, vršiti definisanje varijabilnog dela koda. U našem istraživanju pokazalo se da je najpovoljini vid formata standardnog ulaza XML datoteka, zbog velikih mogućnosti za manipulaciju i jednostavne navigacije kroz datoteku pomoću Xpath, lambda ili regularnih izraza.

3. METODE RAZVOJA GENERATORA PROGRAMSKOG KODA

Osnovna razlika među generatorima programskog koda leži u pristupu koju koriste za obradu ulaznih parametara, koja kao rezultat daje programski kod. Postoji veliki broj mehanizama za generisanje programskog koda, s tim da se svi mogu svrstati u sledeće tri grupe:

1) Opšti pristup;
2) Pristup zasnovan na tehnologijama izabranog razvojnog okvira;
3) Pristup zasnovan na korišćenju šablona.

Za svaki od ova tri pristupa, specifično je da se u okviru njih čuva obrazac, koji nakon što se popuni metapodacima, na svom izlazu daje datoteku sa programskim kodom. Metapodaci su informacije o aplikaciji, koju programer želi da konstruiše, a koje se mogu dobiti iz strukture baze podataka, korisničkog unosa ili pripadaju nekoj drugoj vrsti izvora informacija (WSDL, alata baziranih na UML-u, refleksija) [3].

3.2 *Metoda opšteg pristupa*

Opšti pristup generisanju koda predstavlja princip gde se generički deo koda upisuje u tekstualni tip podatka, koji se kasnije popunjava podacima koji se dobijaju iz ulaznih datoteka (varijabilni deo koda). Najčešće je korišćen način za generisanje programskog koda u svetu programiranja, jer ne zahteva napredno poznavanje drugih tehnologija, već jedino zahteva od programera poznavanje osnovnih operacija sa objektima klasa *String*, *StringWritter*, *StreamReader* i *StreamWritter*. Opšti pristup ne nameće nikakve stroge standarde koji se tiču organizacije koda unutar njega, već se sve ostavlja na savesti programera koji ga piše. To ponekad može predstavljati problem, posebno prilikom dodavanja novih funkcionalnosti u generator [3].

Glavna mana ovakvog pristupa generisanju programskog koda je to što za svaku liniju koda koju želimo da generišemo potrebno je napisati liniju koda koja će taj rezultat da ispiše kao rezultat, a na sve to se mora dodati i programski kod, koji je zadužen za proces obrade ulaznih parametara. U prostijim primerima, to možda ne deluje kao veliki problem, ali u situaciji kada je potrebno izgenerisati program sa nekoliko stotina ili hiljada linija koda, mnogo je veći trud potreban za pisanje generatora nego samog programa. Drugi bitan problem se odnosi na sam dizajn generatora. Naime, teško je razdvojiti programsku logiku od onoga što je potrebno da se upisuje kao rezultat, što otežava proces čitanja i razumevanja programskog koda kao i njegovog održavanja. Samim tim, otežan je i proces pronalaženja i otklanjanja grešaka, što se posebno odnosi na moguće sintaksne greške u kodu koji se generiše [2].

3.3 *Pristup zasnovan na tehnologijama izabranog razvojnog okvira*

U objektno orijentisanom svetu, ovaj pristup se najčešće zasniva na primeni refleksije, odnosno programskog interfejsa koji vraća podatke o kompajliranim klasama i njihovim članovima. Pored toga, u okviru Microsoft-ovog .NET razvojnog okvira moguće je koristiti i druge, podjednako moćne objektne modele kao što su CodeDom i Roslyn.

3.3.1 CodeDom model

CodeDom objektni model omogućava kreiranje apstraktnog modela programskog koda, koristeći objekte koji predstavljaju imenske prostore klasa, klase, metode, izraze, naredbe, promenljive. Njegova glavna prednost je što kreiranjem ovakvog apstraktnog modela programskog koda, otvaramo mogućnost za generisanje koda na više različitih programskih jezika u okviru .NET razvojnog okvira. Moguće je generistati kod za potpuno novi jezik, ali je prethodno potrebno da se za njega napiše odgovarajuća provajder klasa [6].

Velika mana ovog modela je to što je moguće generisati samo elemente programskog jezika koji su opšteg karaktera - CodeDom model može da opiše samo one jezičke strukture koje su zajedničke za sve programske jezike .NET okruženja [7].

CodeDom predstavlja veoma koristan programski interfejs za metaprogramiranje koji programeru omogućava da:

- Opiše strukturu programskog koda, nezavisno od programskog jezika
- Generiše programski kod za više programskih jezika
- Kompajlira kod i strukturu koda u programske sklopove.

3.3.2 Roslyn programski okvir

Roslyn programski okvir pruža uvid u C# i VB kompajler i njegovu logiku za analizu programskog koda, putem objektnog modela koji odslikava proces rada kompajlera. Svaka faza rada kompajlera predstavljena je odgovarajućim objektnim modelom, koji pruža uvid u informacije koje se dobijaju u određenoj fazi. Tako da je faza parsiranja predstavljena kao sintaksno stablo, deklaraciona faza je predstavljena kao hijerarhijska tabela simbola, faza mapiranja je predstavljena kao objektni model koji pruža uvid u podatke dobijene kompajlerovom semantičkom analizom, i na kraju faza emisije je predstavljena kao objektni model koji generiše CIL kod [5].

Ovakav pristup otvara mnogo mogućnosti za napredak u oblastima kao što su metaprogramiranje, transformacija i generisanje programskog koda, dinamičko korišćenje C# i VB programskih jezika, kao i ubacivanje C# i VB elemenata u domenski specifične jezike [4].

3.4 *Pristup zasnovan na korišćenju šablona*

Ovaj pristup polazi od činjenice da se željeni generisani kod, dobija kao rezultat obrade ulaznih podataka u definisanom šablonu. Šablon poseduje programsku logiku kojom podatke dobijene iz ulazne datoteke, pretvara u programski kod. Šabloni se najčešće pišu pomoću sledećih tehnologija: T4, i XSLT jezika. XSLT predstavlja jezik koji je specijalno definisan da generiše bilo koji tip teksta, na osnovu ulaznih podataka koji moraju biti u obliku XML datoteke.

4. OSNOVNI TIPOVI GENERATORA PROGRAMSKOG KODA

Generatori koda se mogu podeliti u dve osnovne kategorije: aktivne i pasivne. U pasivnu kategoriju spadaju generatori koji izgenerišu određen programski kod, koji programer može da menja po svom nahođenju. Pasivni kod generatori ne snose nikakvu odgovornost za kod koji izgenerišu bilo kratkoročno ili dugoročno. Primer pasivnih generatora predstavljaju "vizardi" u integrisanim razvojnim okruženjima [3].

Aktivnu kategoriju čine oni generatori koji pružanju dugoročnu podršku programskom kodu tako što omogućavaju pokretanje u više iteracija preko istog izlaznog koda.Sa protokom vremena promene u programskom kodu bivaju neophodne, pa članovi razvojnog tima mogu ubaciti dodatne parametre u generator koda, koji nakon ponovnog pokretanja daje ažuriran programski kod. U našem istraživanju pažnju smo posvetili aktivnim generatorima koda, jer kao takvi imaju znatno veće područje praktične primene.

Generatori koda se mogu podeliti po njihovoj kompleksnosti, nameni ili po vrsti izlaza koji daju. Najjednostavniji pristup je podela prema izlaznom kodu, jer arhitekture generatora programskog koda kao i njihove namene variraju.Gledano po tipu izlaza koje generatori daju postoji pet različitih tipova generatora koda.

4.1. *Code munging*

Code munging iz ulazne datoteke izvlači bitne ili naglašene karakteristike i koristi ih kako bi napravio izlaznu datoteku. Prilikom obrade ulaznih datoteka, korišćenjem regularnih izraza ili parsiranjem koda, primenjuju se ugrađeni ili eksterni šabloni, kako bi se kao rezultat dobila izlazna datoteka. Code munging generator može se koristiti prilikom kreiranja dokumentacije ili za iščitavanje konstanti ili funkcionalnih prototipova iz dokumenta [3].

4.2. *Inline-code expander*

Inline-code expander kao ulaz dobija programski kod koji on prevodi u izvršni programski kod. Ulazna datoteka sadrži poseban pseudo kod, koji će expander zameniti izvršnim kodom prilikom kreiranja konačne datoteke. Vrlo često se koriste prilikom dodavanja SQL upita u programski kod. Programeri postavljaju anotacije čija je namena da nagoveste inline-code expanderu gde treba da ubaci SQL upit ili komandu. Samim tim se programski kod "čisti" od infrastrukture koja se koristi za upravljanje SQL upitima [3].

4.3. *Mixed-code generator*

Mixed-code generator čita ulaznu datoteku i nakon toga je menja. Razlikuje se od prethodnog tipa generatora po tome što, umesto da izgeneriše potpuno novu datoteku na izlazu, on novi sadržaj upisuje u ulaznu datoteku.Ovaj tip generatora traži specijalne komentare, na čije mesto stavlja novi programski kod [3].

4.4. *Partial-class generator*

Partial-class generator čita apstraktnu definiciju iz ulazne datoteke, koja omogućava da se izgeneriše skup klasa. Nakon toga, koristi šablone kako bi izgenerisao biblioteke izlaznih klasa. Tako dobijene klase se kombinuju sa klasama koje su napravili sami programeri, kako bi dobili konačni produkcioni skup klasa. Ovaj tip generatora predstavlja polaznu tačku prilikom konstruisanja generatora aplikacionog sloja. U početku, generator će praviti samo osnovne klase, dok sa porastom broja posebnih slučajeva koje generator obrađuje, doći će do prelaza na generator koji generiše kod za čitav aplikacioni sloj [3].

4.5.*Generator sloja aplikacije (generatori aplikacionog sklopa)*

U okviru ove grupe, generatori preuzimaju na sebe odgovornost kreiranja programskog koda za čitav aplikacioni sloj. Jedan od principa rada ovih generatora je modelom vođeno generisanje, gde se kombinuju aplikacija za pravljenje konceptualnog modela (UML ili PMOV model) i generator programskog koda, koji na ulazu najčešće dobija XML datoteku, a kao izlaz dobije se jedan ili više slojeva aplikacije. Princip rada se zasniva na tome da generator obradi ulaznu datoteku i da pomoću šablona napravi izlazne klase, koje predstavljaju implementaciju onoga što je navedeno u ulaznoj datoteci.

Glavna razlika između parcijalnog generatora i generatora aplikacionog sloja je u tome što parcijalni generišu samo osnovne klase, dok generatori aplikacionog sloja generišu logičku celinu. Prednost parcijalnih generatora je u brzini implementacije. Najveći problem prilikom rada sa generatorima aplikaciong sloja jeste dodavanje novih specifičnih slučajeva, dok u slučaju parcijalnih generatora, korisnik može da napravi jednostavan generator, a specifične slučajeve rešiti kasnije u okviru programskog koda, koji korisnik sam implementira. Kada su korisnički zahtevi i elementi dizajna nejasni, najbolje je razvijati parcijalni generator, a kasnije u toku razvoja, kada se ti zahtevi ustale i kada se izdvoji deo koda koji postaje generički od onog koji se menja, poželjno je preći na generator aplikacionog sloja [3].

5. STUDIJSKI PRIMERI

Za potrebe našeg istraživanja napravili smo dve aplikacije koje generišu programski kod. Prva predstavlja generator koda koji je razvijan primenom metode opšteg pristupa, i predstavlja parcijalni generator koda. Druga predstavlja generator koda čiji se rad zasniva na obradi šablona, i generiše kod za jedan ili više aplikacionih slojeva.

5.1. *Modeliranje realnog sistema*

Izrađivani su modeli realnih situacija pomoću proširenog modela objekti i veze (PMOV). Naredna sekcija prikazuje podržane elemente modela i njihovo preslikavanje u tabele baze:

5.1.1. Veza između dva jaka objekta

(1,1) – (1,1): Pravi se jedna relacija, oba entiteta ulaze u nju i kandidat za ključ je jedan od identifikatora. Imajući u vidu da oba kandidata imaju jednaku verovatnoću da postanu primarni ključ relacije, u našim primerima smo odlučili da, u slučaju ove veze, uzmemo identifikator prvog entiteta u vezi za primarni ključ relacije.

(0,1) – (1,1): Svaki entitet je jedna relacija, ključ entiteta koji ima donju granicu kardinalnosti 0 ubacujemo kao spoljni ključ u drugu relaciju.

(0,1) – (0,1): Svaki entitet postaje šema relacije, i za vezu se pravi jedna. U relaciju veze ubacujemo ključeve obe entitetske relacije, i oba su kandidati za ključ. Predefinisano se uzima ključ prve relacije kao primarni ključ relacije veze.

(1,1) – (0,M): Veza ne postaje posebna relacija, dok entitetska relacija čija gornja granica kardinalnosti ima vrednost 1, daje svoj ključ kao spoljni ključ drugoj relaciji.

(1,1) – (1,M): Veza ne postaje posebna relacija, dok entitetska relacija, čija gornja granica kardinalnosti ima vrednost 1, daje svoj ključ kao spoljni ključ drugoj relaciji.

(0,1) – (0,M): Veza postaje posebna relacija, elementi su joj primarni ključevi entitetskih relacija, a ključ joj je primarni ključ entitetske relacije sa gornjom granicom kardinalnosti 1. *Drugi način da se ovo reši je da se u entitetsku relaciju sa gornjom granicom kardinalnosti 1 spusti ključ druge entitetske relacije. Time se izbegava pravljenje nove tabele za relaciju veze.*

(0,1) – (1,M): Veza postaje posebna relacija, elementi su joj primarni ključevi entitetskih relacija, a ključ joj je primarni ključ relacije sa gornjom granicom kardinalnosti 1. *Drugi način da se ovo reši je da se relaciji sa gornjom granicom kardinalnosti 1 dodeli, kao vrednost spoljnog ključa, ključ druge entitetske relacije. Time se izbegava pravljenje nove tabele za relaciju.*

(0,M) – (0,M): Veza postaje posebna relacija, njen primarni ključ je složen i sastoji se od primarnih ključeva entitetskih relacija.

(0,M) – (1,M): Veza postaje posebna relacija, njen primarni ključ je složen i sastoji se od primarnih ključeva entitetskih relacija.

5.1.2. Veza jakog i slabog objekta

Svaki entitet u vezi postaje posebna relacija. Sve veze jakog i slabog objekta imaju jednu zajedničku karakteristiku – ključ nadređenog objekta postaje deo primarnog ključa slabog objekta.

5.1.3. Generalizacija-Specijalizacija

Prave se posebne relacije za nadtip i podtip, s tim da je primarni ključ nadtipa ujedno i primarni ključ podtipa. Veze specijalizacije sa gornjom kardinalnošću M (kao i sa donjom granicom kardinalnosti koja je veća od 1) ne mogu se implementirati u objektno orijentisanim jezicima Java i C#, jer ne postoji mehanizam koji bi omogućio da jedna instanca predstavlja više klasa. Veze između podtipa i nadtipa tretiraju se kao veze dva jaka objekta.

Prilikom modeliranja rekurzivnih veza uočili smo da treba izbegavati veze sa donjom granicom kardinalnosti 1, jer se onda dolazi u situaciju da nadtip ne može da postoji bez podtipa, kao i da je neophodno prvo definisati element nadtipa, kako bi se definisao podtip.

5.1.4. Rekurzivne veze

Posmatraju se kao veze između dva jaka objekta. Prave se relacije za entitet, i u zavisnosti od kardinalnosti veze, i za samu vezu. Spoljni ključ se prosleđuje sa izmenjenim imenom. U slučaju postojanja višestrukih veza, i ukoliko je kardinalnost takva da je potrebno napraviti posebnu relaciju za nju, treba napraviti onoliko relacija koliko ima odgovarajućih veza.

5.2. *Studijski primer parcijalnog generatora koda (razvijan metodom opšteg pristupa)*

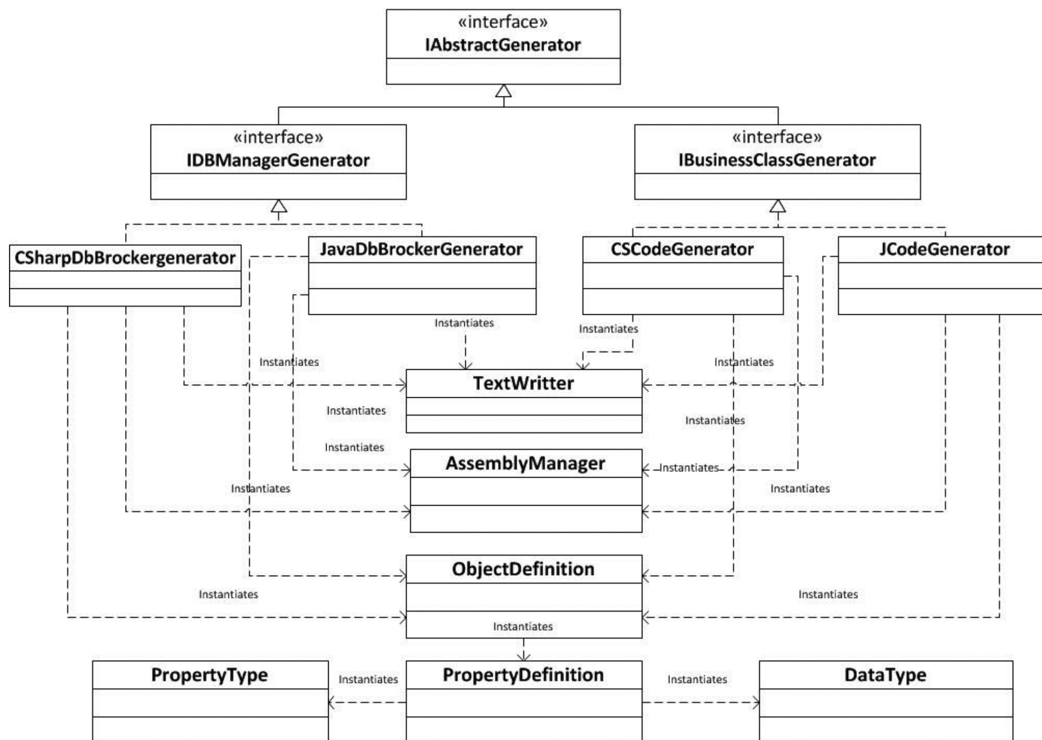
Generator koda se sastoji iz nekoliko segmenata: korisnički interfejs, sekcija za generisanje šablona, sekcija koja je zadužena za parsiranje i deserializaciju ulazne datoteke, generatore datoteka sa programskim kodom, koji na osnovu vrednosti iz ulazne datoteke popunjavaju datoteke sa odgovarajućim kodom. *Slika 2* predstavlja konceptualni model ovog generatora.

Ovim generatorom korisniku se pruža mogućnost da na osnovu ulazne XML datoteke, koja se obrađuje u logici generatora kao ulazni parametar, izgeneriše kod za domenske klase i klase za upravljanje bazom podataka, za jezike C# ili Java. Korisniku je na raspolaganju mogućnost da iskoristi ili da promeni postojeći šablon, kao i da napravi novi. Pored toga, za .NET platformu je prisutna mogućnost da se izgenerisane datoteke grupišu u programske sklopove (assemby), i da se kao biblioteke klasa distribuiraju kroz sistem. *Slika 1* predstavlja primer ulazne XML datoteke:

```
<?xml version="1.0" encoding="UTF-8"?>
<ObjectDefinition>
  <ClassName>Osoba</ClassName>
  <ClassNamespace>Range</ClassNamespace>
  <Property>
    <ClassName>Osoba</ClassName>
    <PropertyName>ID</PropertyName>
    <CompositeTypeName></CompositeTypeName>
    <PropType>Simple</PropType>
    <DataType>Int32</DataType>
    <IsID>true</IsID>
  </Property>
  <Property>
    <ClassName>Osoba</ClassName>
    <PropertyName>Ime</PropertyName>
    <CompositeTypeName></CompositeTypeName>
    <PropType>Simple</PropType>
    <DataType>String</DataType>
    <IsID>false</IsID>
  </Property>
  <Property>
    <ClassName>Osoba</ClassName>
    <PropertyName>Prezime</PropertyName>
    <CompositeTypeName></CompositeTypeName>
    <PropType>Simple</PropType>
    <DataType>String</DataType>
    <IsID>false</IsID>
  </Property>
</ObjectDefinition>
```

Slika 1: *Primer ulazne XML datoteke*

Generisanje programskog koda obavlja se metodom opšteg pristupa (Brute Force metoda). Ova, relativno jednostavna, metoda za pravljenje generatora koda podrazumeva da se



Slika 2: *Konceptualni model parcijalnog generatora*

kompletan programski kod upisuje u objekat tipa *StringWriter*, čiji se sadržaj čuva u datoteci programskog koda (sa ekstenzijom .cs ili .class).

Kompletan proces generisanja programskog koda se odvija u okviru klase *CodeGenerator* (odnosno *DBBrokerGenerator*, ukoliko se radi o generisanju klase za rad sa bazom), u okviru koje se vrši upisivanje specifičnih informacija koje dolaze iz korisnički definisane datoteke u statički programski kod. Specifični podaci koji dolaze iz XML datoteke se deserijalizuju u objekat tipa *ObjectDefinition*. Sama klasa *ObjectDefinition* je parcijalna klasa, izdeljena u dve datoteke *ObjectDefinitionAbstraction* i *ObjectDefinitionCustom*. U prvoj klasi se čuvaju polja koja se odnose na mapiranje (deserijalizaciju) XML datoteke u podatke o samoj klasi, poljima i njihovim pristupnim metodama, kao i podaci o identifikatoru generisane klase. Druga klasa sadrži metode koje su zadužene za upisivanje ili čitanje iz XML datoteke.Klasa zadužena za upisivanje programskog koda u *StringWriter* je klasa *TextWritter*, koja pravi instancu StringWriter objekta i pomoću seta metoda vrši upisivanje tekstualnih vrednosti. Pored toga, ona programeru omogućava da kontroliše horizontalni razmak (space), korišćenjem metoda Indent i Outdent.

Objekat koji se dobija deserijalizacijom XML datoteke, čuva se u *TypeManager* klasi. Ona omogućava pozivanje metoda koje deserijalizuju XML u *ObjectDefinition* objekat, ali i njegovo ponovno serijalizovanje u XML, kada se vrši kreiranje templejta iz same klase.

U okviru *ObjectDefinition* klase, postoje posebne instance u okviru kojih se čuvaju podaci o poljima i tipovima podataka tih polja. Te podatke čuva klasa *PropertyDefinition*. U njenim poljima se čuvaju podaci o imenu, tipu podatka polja, kao i podaci o kardinalnosti (da li je u pitanju prost tip podatka, objekat, lista objekata ili lista prostih tipova.)

Tip podatka i vrednost kardinalnosti se čuvaju u posebni enumeracijama: *DataType* i *PropertyType* .

5.2.1. Pregled podržanih operacija

Parcijalni generator poseduje sledeće operacije:

1. Kreiranje ulazne XML datoteke.
2. Ažuriranje postojeće ulazne XML datoteke.
3. Brisanje ulazne XML datoteke.
4. Generisanje programskog koda za domenske klase.
5. Generisanje ulazne datoteke na osnovu postojeće klase.
6. Generisanje koda za rad nad bazom.
7. Pravljenje biblioteke klasa (samo za C#).

5.2.1.1. Kreiranje ulazne XML datoteke

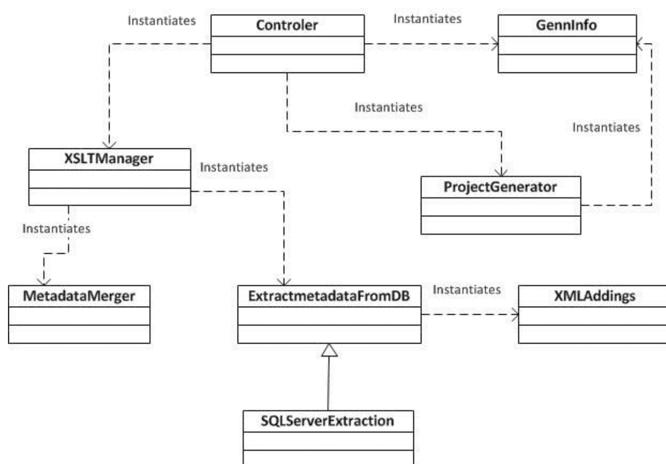
Ulazna datoteka je neophodna za rad generatora jer sadrži podatke o varijabilnom delu koda koji se generiše. Ti podaci se odnose na klasu koja se generiše: njeno ime, naziv paketa ili imenskog prostora u kom se nalazi, kao i podatke o njenim poljima. Pomenuti podaci se unose preko korisničke forme, gde korisnik unosi ime klase koju želi da generiše, ime njenog imenskog prostora (ili paketa za Java jezik). Za svako njeno polje može da unese ime polja, tip podatka koji se čuva u njemu. Podržali smo tipove polja koji čuvaju osnovne tipove podataka, kao i polja koja predstavljaju referencu na neki drugi entitet. Moguće je definsati neko polje i kao kolekciju. Nakon završene kofiguracije, korisniku se prezentuje ulazna datoteka, koja se čuva u XML formatu.

5.2.1.2. Ažuriranje postojeće ulazne XML datoteke

Omogućili smo korisniku da sa sistema datoteka učita već postojeću datoteku, i da je izmeni po svom nahođenju. Ulaznu datoteku je moguće menjati u slobodnoj tekstualnoj formi, ili preko korisničke forme.

5.2.1.3. Brisanje ulazne XML datoteke

Svaku kreiranu ulaznu datoteku je moguće trajno ukloniti sa sistema. Do željene datoteke dolazi se navigacijom kroz sistem datoteka. Jednom obrisana ulazna datoteka može se ponovo dobiti iz generisane klase.



Slika 3: *Konceptualni model generatora aplikacionog sloja*

5.2.1.4. Generisanje programskog koda za domenske klase

Nakon kreiranja (ili odabira postojeće) ulazne datoteke, generator nudi mogućnost da se odabere programski jezik na kom će se generisati izlazni kod. Omogućili smo da se generišu klase na programskim jezicima Java i C#. Sledeći korak je odabir tipa klase koji se generiše: POJO (POCO za C#) ili klasa za komunikaciju sa bazom. Nakon odabira jezika i tipa klase, neophodno je odabrati lokaciju gde će se čuvati konačne datoteke sa kodom.

5.2.1.5. Generisanje ulazne datoteke na osnovu postojeće klase

Izgenerisana klasa (ili bilo koja druga C# klasa) može da posluži kao šablon za kreiranje ulazne datoteke. Ovo smo rešili pokretanjem XSD.EXE aplikacije, koja prima klasu, a kao izlaz izgeneriše XSD shemu. Na osnovu XSD sheme moguće je napraviti ulaznu XML datoteku.

5.2.1.6. Generisanje koda za rad nad bazom

Navedena operacija omogućava da se, na osnovu ulazne datoteke, generiše klasa koja će korisniku omogućiti da ostvari konekciju ka bazi, i da obavlja osnovne (CRUD) operacije nad bazom.

5.2.1.7. Pravljenje biblioteke klasa

Naš studijski primer omogućava da se kreirane klase sačuvaju u biblioteku klasa. Neophodno je odabrati skup generisanih klasa (ili klasa koje su se već nalazile na sistemu datoteka) i proslediti ih kao ulazni parametar modulu koji, pomoću CodeDom modela generiše biblioteku klasa.

5.3. Studijski primer generatora aplikacionog sloja (razvijan metodom obrade šablona)

U ovom delu dat je primer generatora koji kao ulaz uzima podatke iz MS SQL baze podataka, i na izlazu daje klase i datoteke, koji se mogu ukombinovati u gotov projekat. Datoteke koje se dobijaju na izlazu postaju ulaz u poseban modul, koji je zadužen za generisanje Visual Studio projekta, s tim da to može biti konzolna , Windows form aplikacija, ili ASP.NET projekat. *Slika 3* predstavlja konceptualni model generatora koji smo razvili za potrebe istraživanja.

Princip rada se zasniva na XML datoteci, koja se dobija kao rezultat obrade podataka dobijenih iz baze. Na osnovu tih podataka XML datoteka se puni odgovarajućim informacijama, na osnovu kojih se definišu podaci potrebni u procesu generisanja različitih datoteka. Uz pomoć XML datoteka, kao i odgovarajućih parametara koji se dobijaju preko korisničkih formi, moguće je izgenerisati programski kod za čitavu .Net aplikaciju, ili samo pojedinačne datoteke. Generator koda koji sledi ovaj princip, predstavlja mešavinu generatora aplikacionog sloja i parcijalnog generatora, što proizilazi iz njegove prirode da, u zavisnosti od korisničkih zahteva, može da generiše aplikaciju sa funkcionalnim korisničkim interfejsom i vezom ka bazi, a može i da generiše samo jedan njen deo ili čak pojedinačne datoteke. Na osnovu šeme baze pravi se ulazna datoteka u formi XML-a koji u sebi sadrži podatke o tabelama, kao i o njihovim relacijama. Kasnije se takva datoteka proširuje dodatnim podacima. *Slika 4* prikazuje primer jedne ulazne XML datoteke:

```
<dbs:Table Name="Proizvod" OriginalName="Proizvod">
   <dbs:TableColumns>
     <dbs:TableColumn Name="SifraProizvoda"
         OriginalName="SifraProizvoda"
         SQLType="Int" NETType="System.Int32"
         MaxLength="" IsPrimaryKey="true"/>
     <dbs:TableColumn Name="Naziv" OriginalName="Naziv"
         SQLType="NVarChar" NETType="System.String"
         MaxLength="50" IsPrimaryKey="false"/>
     <dbs:TableColumn Name="Kolicina"
         OriginalName="Kolicina"
         SQLType="Int" NETType="System.Int32" MaxLength=""
         IsPrimaryKey="false"/>
   </dbs:TableColumns>
   <dbs:TableConstraints>
     <dbs:PrimaryKey>
       <dbs:PKField Name="SifraProizvoda"/>
     </dbs:PrimaryKey>
     <dbs:TableRelations>
       <dbs:ChildTables>
         <dbs:ChildTable Name="Korpa_Proizvod">
           <dbs:ChildKeyField Name="SifraProizvoda"/>
         </dbs:ChildTable>
       </dbs:ChildTables>
     </dbs:TableRelations>
   </dbs:TableConstraints>
 </dbs:Table>
```

Slika 4: *Primer ulazne XML datoteke*

Na osnovu pomenute XML datoteke, vrši se generisanje skladišnih procedura, kao i domenskih klasa. Proširivanjem ove datoteke dobija se ulazna datoteka za generisanje klasa, koje će se kasnije koristiti za logiku aplikacije. Za svaki element koji predstavlja tabelu, pravi se jedan element koji predstavlja klasu i njene atribute. Takođe se prenose podaci o vezama između tabela, i to postaju relacije između klasa. *Slika 5* prikazuje deo XML datoteke koja je zadužena za čuvanje informacija o klasama.

```
<orm:Object Root="true" Collection="true" Child="true" CollectionRoot="true" CollectionChild="false"
ObjectChild="false" Name="Proizvod" TableName="Proizvod" CollectionName="Proizvod">
          <orm:ChildCollection Name="Korpa_Proizvod" ObjectName="Korpa_Proizvod" ChildTableName="Korpa_Proizvod">
            <orm:ChildKeyField Name="SifraProizvoda" />
          </orm:ChildCollection>
          <orm:Properties>
            <orm:Property Name="SifraProizvoda" Column="SifraProizvoda" SQLType="Int" NETType="System.Int32"
                 Empty="0" IsPrimaryKey="true"
                ControlType="System.Windows.Forms.TextBox" ControlPrefix="txt" ControlName="txtSifraProizvoda"
                BindProperty="Text" ReadOnly="true" Display="false">
            </orm:Property>
            <orm:Property Name="Naziv" Column="Naziv" SQLType="NVarChar" NETType="System.String"
                Empty="&quot;&quot;" IsPrimaryKey="false" MaxLength="50"
                ControlType="System.Windows.Forms.TextBox" ControlPrefix="txt" ControlName="txtNaziv"
                BindProperty="Text" ReadOnly="false" Display="true">
            </orm:Property>
            <orm:Property Name="Kolicina" Column="Kolicina" SQLType="Int" NETType="System.Int32"
                Empty="0" IsPrimaryKey="false" ControlType="System.Windows.Forms.TextBox"
                ControlPrefix="txt" ControlName="txtKolicina" BindProperty="Text" ReadOnly="false" Display="true">
            </orm:Property>
          </orm:Properties>
          <orm:AllProperties />
        </orm:Object>
```

Slika 5: *Primer ulazne XML datoteke za generisanje klasa*

```
<xsl:template match="orm:Objects" mode="BuildClasses">
      <xsl:for-each select="orm:Object">
          <xsl:choose>
              <xsl:when test ="@Name = 'sysdiagram'">
              </xsl:when>
              <xsl:otherwise>
          <xsl:variable name="dirname" select ="'..\..\..\XSLTResourceCreator\FinalResultObjectClasses\'"/>
          <xsl:variable name="filename" select="concat($dirname,@Name,'.cs')"/>
          <xsl:result-document method="text" href="{$filename}">
                  <xsl:call-template name="Header"/>
                  using Middletier;
                  <xsl:variable name="PrimaryKeyPr" select="orm:Properties/orm:Property[@IsPrimaryKey='true']/@Name"/>
                  namespace ObjectClasses
                  {
                  [CodeAttribute("<xsl:value-of select="@TableName"/>")]
                  public class <xsl:value-of select="@Name"/><xsl:call-template name="ParentClass">
                      <xsl:with-param name="PrimaryKeyProp" select="$PrimaryKeyPr"/>
                  </xsl:call-template>
                  {
                  <xsl:call-template name="ClassConstructors"/>
                  <xsl:call-template name="ClassFields"/>
                  <xsl:call-template name="FieldProperties">
                      <xsl:with-param name="tableName" select="@TableName"/>
                  </xsl:call-template>
                  <xsl:call-template name ="CreteChildCollection"/>
                  <xsl:call-template name="FillTheChildList"/>
                  }
                  }
          </xsl:result-document>
              </xsl:otherwise>
          </xsl:choose>
      </xsl:for-each>
  </xsl:template>
```

Slika 6: *Primer XSLT transformacije za generisanje klasa*

Da bi se na osnovu prikazanih XML datoteka dobile željene datoteke sa programskim kodom, neophodno je da se XML datoteke obrade pomoću XSLT transformacija. Parsiranje transformacija se vrši pomoću SAXON modela, a transformacije su pisane da podrže XSLT verziju 2.0. Svaka transformacija podržava generisanje višestrukih datoteka, gde se za svaki element pravi posebna datoteka. Primer jedne transformacije, koja kao izlaz daje C# klasu, može se videti na *Slici 6.*

Sledeća mogućnost koja se nudi korisniku je da se izgenerisane klase uklope u programski sklop, ili da se na osnovu ulaznih podataka napravi kompletna aplikacija. Da bi se to postiglo, potrebno je da se generisane klase ubace kao ulazni parametar u sekciju koja kontroliše generisanje .NET softverske solucije. Ona vodi računa o generisanju projekata, učitavanju generisanih datoteka u odgovarajući projekat, njihovo kompajliranje i integraciju u konačnu programsku soluciju. Sledi primer koda (*Slika 7*) koji kontroliše generisanje softverske solucije:

```
private void GenerateObjectProjects(GennInfo gen, Solution4 sol, string cl)
        {
            ClearDirectory(@"e:\tempSolutionCLOB", @"e:\tempSolutionCLOB\classLibproject");
            sol.Create(@"e:\tempSolutionCLOB", "ObjectProject");
            MessageFilter.Register();
            sol.AddFromTemplate(cl, @"e:\tempSolutionCLOB\classLibproject", "ObjectProject", true);
            string[] ObjectFiles = Directory.GetFiles(gen.locationObj);

            for (int i = 0; i < ObjectFiles.Length; i++)
            {
                sol.Projects.Item(1).ProjectItems.AddFromFileCopy(ObjectFiles[i]);
            }

            VSProject vp1 = (sol.Projects.Item(1)).Object as VSProject;

            vp1.References.Add(gen.locationMid + @"\Middletier.dll");

            sol.SaveAs("ObjectProject.sln");

            SolutionBuild2 solSb = (SolutionBuild2)sol.SolutionBuild;
            solSb.SolutionConfigurations.Item("Debug").Activate();
            solSb.Build(true);
            sol.Close(false);
            MessageFilter.Revoke();

            MessageFilter.Revoke();
        }
```

Slika 7: *Programski kod kojim se generiše Visual Studio programska solucija*

5.3.1.Pregled podržanih operacija

Studijski primer generatora aplikacionog sloja podržava sledeći skup operacija:

1. Generisanje koda domenskih klasa.
2. Generisanje koda skladišnih procedura.
3. Generisanje koda korisničkog interfejsa.
4. Generisanje softverske solucije za razvojno okruženje Visual Studio.

Prvi korak prilikom realizacije ovih operacija je odabir baze podataka, na osnovu koje se generiše ulazna XML datoteka.

5.3.1.1. Generisanje koda klasa

Nakon generisanja odgovarajuće ulazne datoteke, omogućili smo da se ista prilagodi procesu generisanja domenskih (POCO) klasa. Ovim korakom je moguće definisati koje tabele iz baze želimo da prikažemo kao klase u sistemu, kao i koje kolone tebele će biti mapirane u budućim klasama. Sledeći korak je odabir lokacije gde će se čuvati konačne datoteke.

5.3.1.2. Generisanje koda skladišnih procedura

Skladišne procedure uzimaju u obzir polja koja su definisana (u slučaju da je zahtev da se generišu i klase) u klasama. Ukoliko se kreiraju samo skladišne procedure, u obzir prilikom njihovog kreiranja se uzimaju podaci o tabelama iz odabrane baze.

5.3.1.3. Generisanje koda korisničkog intefejsa

Na osnovu prethodno kreirane ulazne datoteke za generisanje klasa, vrši se generisanje ulazne datoteke za kreiranje klasa korisničkog interfejsa. Tom prilikom svakom polju se pridružuje odgovarajući tip kontrole, kojom će ono biti prikazano korisniku na konačnoj formi. Tip kontrole zavisi od odabrane vrste korisničkog interfejsa, kao i od tipa samog polja.

5.3.1.4 Generisanje softverske solucije za razvojno okruženje Visual Studio

Generisane .Net klase, postojeće C# klase i programski sklopovi, kao i elemente korisničkog interfejsa, moguće je uklopiti u jednu softversku celinu – Visual Studio programska solucija. Prilikom pokretanja generatora, pružena je mogućnost da se generisane datoteke uklope u Visual Studio programsku soluciju. Omogućili smo da se odabere nekoliko različitih (trenutno aktuelnih) verzija samog razvojnog okruženja. Tipovi softverske solucije, koje je moguće generisati su:

1. Konzolna aplikacija (Console application).
2. ASP.NET aplikacija (ASP.NET Web application).
3. Windows form aplikacija (Windows Form application).

6. PRAVCI DALJEG RAZVOJA

Prilikom razvoja prototipova generatora koda prednost je data razvoju osnovnih funkcionalnosti. Naredni pravci razvoja obuhvatiće poboljšanja postojećih, kao i razvoj novih funkcionalnosti, koje za cilj imaju poboljšanje kvaliteta koda koji se dobija korišćenjem generatora. Neke funkcionalnosti nisu uzete u razmatranje, jer se za njima nije ukazala potreba, ili tip generatora po svojoj definiciji ne može da ih podrži.

6.1 *Pravci razvoja prototipa generatora razvijanog metodom obrade šablona*

U toku je rad na poboljšanjima koja će imati uticaja na sledeće funkcionalnosti:

1. Podržati rad sa drugim tipovima baza podataka. Trenutno prototip omogućava rad samo sa SQL Server bazama.
2. Implementiranje dinamičke logike u generisanoj aplikaciji. Poboljšanje omogućava primenu Roslyn modela.
3. Mapiranje novih polja generisanih klasa u tabele. Logika, koja će ovo omogućiti, zasnivaće se na T4 šablonima.
4. Stilizovanje korisničkog interfejsa. Poboljšanje se ogleda u generisanju korisničkog interfejsa koji može da prihvati stilske datoteke, koje definišu izgled korisničkog interfejsa (css, master page, apstraktne klase, specijalizovane kontrole).

Funkcionalnosti čiji se razvoj ne razmatra:

1. Implementacija specifične logike u generisanim klasama. Nju je neophodno napisati u klasi koja nasleđuje generisanu klasu, ili generisati parcijalne klase.
2. Promena upita nad bazom. Tipizirani upiti su odvojeni od same aplikacije, dok će svaka promena upita skladišnih procedura biti pregažana sledećim pozivom generatora.
3. Kreiranje apstraktnih klasa i interfejsa.

6.2 *Pravci razvoja prototipa generatora razvijanog metodom opšteg pristupa*

U toku je rad na poboljšanjima koja će imati uticaja na sledeće funkcionalnosti:

1. Kreiranje tabela baze pomoću podataka o klasama. Poboljšanje će pružiti mogućnost da se napravi tabela u izabranoj bazi ili, ukoliko baza ne postoji, da se prvo ona kreira.
2. Implementiranje dinamičke logike u generisanoj aplikaciji. Poboljšanje omogućava primenu Roslyn modela.
3. Trenutno broker klasa ne uzima u obzir da li baza postoji. Imena kolona i tabela dobijaju se na osnovu podataka o klasi. Poboljšanjem aplikacione logike, podaci o tabelama i kolonama će se upoređivati sa onima u bazi, ukoliko baza i/ili tabela postoje. Ukoliko baza ili tabela ne postoje, prototip kreira novu bazu/tabelu, podatke o tabelama popunjava na osnovu podataka o klasi.
4. Kreiranje apstraktnih klasa, interfejsa, kao i njihove implementacije.
5. Generisanje više klasa u jednoj iteraciji. Ovakvu funkcionalnost je moguće postići promenom logike obrade, kao i ulazne XML datoteke.

Funkcionalnosti čiji se razvoj ne razmatra:

1. Nije podržano kreiranje skladišnih procedura. Prototip kreira broker sa tipiziranim upitima.
2. Nije podržano generisanje korisničkog interfejsa.

Jedan od budućih modula prototipova generatora omogućiće generisanje koda za NUnit testove. Ovim modulom generisanje aplikacionog koda biće praćeno generisanjem testnog koda, kako za aplikacionu logiku, tako i za korisnički interfejs. Kao dominantni pristup razvoja softvera razmatra se Test Driven Development (TDD) metodologija [8].

Kao jedan od pristupa razvoju logike budućih aplikacija razmatra se DCI arhitektura. DCI arhitektura uzima mentalni model krajnjeg korisnika računarskog programa kao osnovni putokaz u kreiranju poslovnih aplikacija. Domenski objekti u DCI arhitekturi nemaju funkcionalnosti, već im se one dodaju dinamički, u vreme izvršenja aplikacije [9].

Kako bi se bolje prikazala generisana struktura, u okviru oba prototipa postojaće mogućnost predstavljanja strukture generisanih klasa preko pseudo UML dijagrama. Nijedan od prototipova generatora ne podržava generisanje enumeracija i struktura, jer se za navedenim tipovima podataka do sada nije ukazala realna potreba.

7. ZAKLJUČAK

U pomenutim principima razvoja generatora programskog koda, najviše se ističu razlike po pitanju održavanja samog generatora, kao i rešavanja potencijalnih problema. Generatori koji primenjuju opšti model razvoja ne zahtevaju znanje naprednih tehnologija, i veoma su jednostavni za implementaciju, dok sa druge strane zadaju velike probleme prilikom potrage za greškama u procesu generisanja koda. Ujedno, količina napisanog koda za generator je identična količini koda koja se generiše: programer mora da napiše aplikaciju u opštem (generičkom) obliku, i da praznine u modelu popunjava pomoću vrednosti iz ulaznih XML datoteka. Prilikom razvoja generatora koji rade na principu obrade šablona, nije bilo većih poteškoća u procesu nadogradnje ili potrage za greškama, ali je težina prebačena na deo aplikacije koji kontroliše generisanje XML datoteke koja sadrži ulazne informacije, na osnovu kojih se generišu datoteke sa programskim kodom. Neophodno je napredno poznavanje rada sa XPath i regularnim izrazima, kako bi se pokrili svi zahtevani scenariji u aplikaciji. U zavisnosti od izbora tehnologije za pravljenje i parsiranje šablona (XSLT ili T4) dodavanje eksternih logičkih elemenata može predstavljati manji ili veći problem. Nakon uspešno završenog generisanja datoteka sa programskim kodom, kao najpraktičnije metode integracije generisanog koda i koda kojim programer lično upravlja, pokazale su se nasleđivanje generisanih klasa i generisanje parcijalnih klasa. U oba slučaja generisani deo je odvojen u posebnu datoteku, te je samim tim kod koji programer piše potpuno bezbedan sa stanovišta mogućeg prepisivanja od strane generatora koda.

Daljim razvojem prototipova dobiće se nove funkcionalnosti koje će znatno doprineti skraćenju vremena razvoja novih aplikacija, kao i poboljšanju kvaliteta i bezbednosti koda koji se dobija primenom generatora.

8. LITERATURA

[1] Practical Code Generation in .NET: Covering Visual Studio 2005, 2008, and 2010 (Addison-Wesley Microsoft Technology Series), *Peter Vogel* (2010)

[2] Code Generation in Microsoft .NET, *Kathleen Dollard* (2004)

[3] Code Generation in Action, *Jack Herrington*(2003)

[4] Metaprogramming in .NET, *Kevin Hazzard i Jason Bock* (2013)

[5] Roslyn projekat (http://msdn.microsoft.com/en-us/hh500769#Toc306015663), *Karen Ng, Matt Warren, Peter Golde, Anders Hejlsberg*

[6] CodeDom model (http://www.codeguru.com/vb/gen/article.php/c19573/Microsoft-NET-CodeDom-Technology.htm), *Quin Street*

[7] CodeDom API (http://msdn.microsoft.com/en-us/library/y2k85ax6.aspx), Microsoft forum

[8] Test driven development uzori irefaktorisanje testnog koda (InfoM časopis http://www.infom.org.rs/brojeviNovi/2013-45.html), Slobodan Mirković , Saša D. Lazarević

[9] Implementacija DCI arhitekture primenom .NET platforme (InfoM časopis http://www.infom.org.rs/brojeviNovi/2013-46.html), Branko Stevanović, Saša D. Lazarević

dr. Saša D. Lazarević, Fakultet organizacionih nauka, Univerzitet u Beogradu
Kontakt: slazar@fon.rs
Oblast interesovanja: softversko inženjerstvo, informacioni sistemi, baze podataka, sistemi za upravljanje dokumentacijom, .NET platform

Stefan Mitić, master inženjer, Fakultet organizacionih nauka
Kontakt: mitic.stefan@orion.rs
Oblast interesovanja: softversko inženjerstvo, generatori programskog koda, obrnuti inženjering, integrisana razvojna okruženja, .NET platforma, Web tehnologije