GraphIt: Komponentni razvoj aplikacije za vizualizaciju podataka u formi grafa

Jovan Đorđić, *SW-46/2018,* Anastasija Đurić, *SW-48/2018,* Dina Petrov, *SW-52/2018,*

***Apstrakt*—Ubrzanim razvojem i popularizacijom mobilnih operativnih sistema i CLOUD tehnologija, sve više se javlja potreba za izgradnjom nezavisnih, integrabilnih i prilagodljivih aplikacija. Komponentnim razvojem kreirana aplikacija je robusna, skalabilna i fleksibilna. Razvoj, prilagođavanje, izmena i unapređivanje pojedinačnih komponenti potpuno je nezavisno, dok njihovo usklađivanje ostavlja utisak jedinstvene, homogene i koherentne celine. U prvom delu rada dekonstruisan je postupak projektovanja softvera baziranog na komponentama i analizirane su prednosti ovakvog pristupa, dok je u drugom delu predstavljena struktura i proces razvoja *GraphIt* aplikacije nastale praktičnom primenom ovih metoda. Aplikacija je namenjena za grafički prikaz skupova podataka i relacija između entiteta unutar njih. Jezgro *GraphIt* aplikacije integriše dva osnovna tipa komponenti - za učitavanje i parsiranje, i za prikaz podataka u formi grafa. Podaci proizvoljnog formata učitavaju se iz različitih izvora i prevode u međureprezentaciju grafičke strukture. Grafički prikazi podataka olakšavaju razumevanje i sagledavanje strukture podataka, kao i izvođenje zaključaka na osnovu generisanog prikaza.**

***Ključne reči*—**komponenta, plugin, CBD (Component Based Development), CORE, jezgro, platforma, servis, graf

I. Uvod

IDEJA razvoja baziranog na komponentama ogleda se u izgradnji kompleksnih softverskih rešenja od unapred definisanih, dizajniranih i konstruisanih delova, koji se ugrađuju u zajedničku vodeću platformu zarad obavljanja predviđene složene funkcionalnosti. Komponente koje sačinjavaju sistem su gotove softverske jedinice koje se odmah mogu uključiti u proces razvoja, bez potrebe za pisanjem celokupnog izvornog koda od samog početka [2].

Ovakav pristup znatno smanjuje cenu i trajanje proizvodnje. Iste komponente mogu se instalirati i koristiti u velikom broju različitih sistema [2]. Sve što je potrebno je definisanje servisnih ugovora koji će nametnuti pravila međukomponentne komunikacije. Zahvaljujući utvrđenim interfejsima preko kojih se odvija komunikacija, pomenute softverske jedinice mogu biti razvijene upotrebom različitih programskih jezika i tehnologija. Sistem koji se konstruiše je skalabilan, a proces proširivanja i uvođenja novih funkcionalnosti u velikoj meri olakšan, jer se obavlja na standardizovan način po tačno definisanom protoklu.

Cilj rada je da ukaže na prednosti komponentnog razvoja i publici predstavi proces primene ovih metoda kroz primer izgradnje *GraphIt* aplikacije za grafičku vizualizaciju podataka. *GraphIt* softverski alat objedinjuje komponente za učitavanje podataka iz nekoliko tipova izvora i njihovu grafičku predstavu na više različitih načina i stepena detaljnosti pruženih informacija. Zahvaljujući komponentnom pristupu, izrada novih softverskih celina i njihova integracija u sistem može se obaviti bez potrebe za redefinisanjem i izmenama na postojećoj aplikaciji koja funkcioniše nezavisno od specifičnih komponenti. Nova komponenta se može jednostavno instalirati i deinstalirati bez uticaja na rad i održivost sistema.

U poglavlju II dat je detaljan opis pojma, postupka, prednosti i ograničenja komponentizacije. Takođe se daje osvrt na česte greške koje nastaju pri pisanju novih programa, a koje se komponentnim razvojem mogu uspešno izbeći. U narednom poglavlju (III), ukratko je prikazan način programiranja *Python* aplikacija u skladu sa principima komponentnog razvoja. Objašnjeni su pojmovi poput *Python module*, *Python package*, *setuptools*, *Python wheel*. Poglavlje IV obrađuje strukturu, organizaciju i implementaciju *GraphIt* aplikacije, uz fokus na implementaciju i specijalizaciju instaliranih komponenti i način njihove međusobne sinhronizacije i komunikacije na CORE-u aplikacije. Opisan je način rada komponenti za učitavanje podataka - *XMLDataLoader* za učitavanje podataka u XML formatu, *FileSystemDataLoader* za učitavanje hijerarhijske strukture datoteka i direktorijuma iz fajl sistema i *DeezerDataLoader i SpotifyDataLoader* za učitavanje informacija o korisničkim plejlistama pesama. Od komponenti za vizualizaciju, obrađeni načini prikaza podataka uz *SimpleVisualisation* za pojednostavljen grafički prikaz, *ComplexVisualisation* za detaljniji prikaz, *TreeView* za prikaz podataka u formi stabla, i *Bird View* za sveobuhvatan umanjeni prikaz celog skupa podataka. Poglavlje V navodi uočene prednosti i izazove prilikom razvoja *GraphIt* aplikacije. U poglavlju VI objašnjena je svrha i cilj *GraphIt* aplikacije kao i način na koji se ona koristi.

II. Razvoj baziran na komponentama

ČEST problem pri razvoju softvera je fokusiranje na pisanje koda od nule umesto istraživanja i integracije postojećih proverenih rešenja. Ovakav pristup za posledicu ima znatno sporiji razvoj i "izmišljanje tople vode" izradom programske celine koja već postoji i dostupna je za korišćenje. Razvoj baziran na komponentama nudi standardizovan način za rešavanje pomenutih problema. U tekstu koji sledi opisano je šta je to CBD, prednosti i izazovi koje on može doneti, izvršeno je poređenje standardnog i namenskog softvera i navedeni su preduslovi komponentizacije.

A. CBD

Razvoj baziran na komponentama (engl. *Component Based Development*, CBD) podrazumeva razvoj softvera upotrebom komponenti kao osnovne gradivne jedinice proizvoda. Autori knjige *Beyond Object-Oriented Programming* [1] dali su sledeću definiciju softverske komponente: "Softverske komponente predstavljaju binarne jedinice nezavisne proizvodnje koje međusobno sarađuju sa ciljem formiranja funkcionalnog koherentnog sistema. Omogućavaju ponovnu iskoristivost programskih celina i amortizaciju investicija prilikom razvoja više različitih aplikacija koje koriste slične funkcionalnosti. Binarni format i nezavisnost ključne su osobine koje omogućavaju robusnu integraciju i prodaju komponente na više prodajnih mesta (engl. *vendors*)".

Komponentizacija kao metoda razvoja nije nova ideja - već dugo se uspešno primenjuje u različitim inženjerskim oblastima. U poljima softverskog inženjerstva, većini programera i inženjera poznata je u vidu biblioteka, softverskih dizajn principa, arhitektura i sl. Izgradnja novih softverskih rešenja uz integraciju gotovih, eksterno dobavljenih komponenti dovodi do kvalitetnog, rapidnog razvoja, te je i ukupno vreme od početnih faza razvoja do isporuke finalnog proizvoda značajno skraćeno.

B. Prednosti CBD

Komponentizacijom se ostvaruje striktna podela nadležnosti. Komponente se bave jasno određenim poslom i komuniciraju sa svojom

okolinom preko jasno definisanih interfejsa i protokola, te je obezbeđen visok nivo kohezije i nizak nivo sprege. To znači da komponente međusobno sarađuju na rešavanju kompleksnih zadataka (visok nivo kohezije), a da pritom postojanje i način rada jedne komponente ne utiče i ne uslovljava drugu (nizak nivo sprege). Svaka komponenta je nezavisna tako da se može ponovno koristiti u više aplikacija (eng. *reusability*).

Iako su komponente celine čiji je postupak razvoja, održavanja i unapređivanja nezavistan, njihova međusobna komunikacija preko protokola i interfejsa obezbeđuje transparentnost komponentizacije. Ona podrazumeva da je modularan karakter aplikacije sakriven od krajnjeg korisnika, te se dobija utisak homogene, koherentne aplikacije.

Kupovinom komponenti od nezavisnih proizvođača specijalizovanih za izradu specifičnog tipa komponenti dovodi do brže izgradnje kvalitetnog softvera jer kupac ne mora da se upoznaje sa detaljima implementacije. Samim tim se i održavanje i unapređivanje dobavljene komponente prebacuje na eksternog proizvođača. Ovi proizvođači komponenti mogu istu komponentu prodavati velikom broju kupaca i time stvoriti preduslove za dalji razvoj i usavršavanje komponente.

C. Izazovi CBD

Kao što je napomenutno u radu [3], pored navedenih prednosti komponentizacije, važno je istaći i određene izazove koji se mogu javiti.

Upravo jedna od glavnih prednosti - integracija gotovih rešenja - takođe predstavlja jednu od većih prepreka prilikom početne organizacije projekta. Problem nastaje kada je neophodno pomiriti funkcionalne i nefunkcionalne zahteve našeg projekta sa mogućnostima gotovih rešenja koje integrišemo. Uvodi se novi element prilikom *requirement engineering*-a [1], a to je pravljenje kompromisa između zahteva aplikacije koja se razvija i sposobnosti dostupnih gotovih komponenti.

Priroda postojećih rešenja takođe predstavlja problem. S obzirom na to da su mnoge komponente dostupne isključivo u binarnom formatu (ne i izvorni kod), teže je verifikovati njihovu verodostojnost, pouzdanost i ispravnost. Samim tim, poseban problem je njihova integracija u okruženja pojačane bezbednosti gde su ove karakteristike ključne.

Na duže staze, zavisnost aplikacije od eksternih rešenja može da "koči" budući razvoj aplikacije ili da dovede do nepredviđenih grešaka na koje nije moguće uticati. Imajući u vidu da ovakav softver sadrži različite eksterne komponente - nameću se pitanja: Da li se zamenom komponente može adekvatno ažurirati sistem? Koje komponente je u datom trenutku moguće, koje poželjno, a koje neophodno ažurirati? Ako neki deo sistema ne funkcioniše ispravno ili otkaže, ko je zaduženo lice koje preuzima odgovornost za rešavanje tog problema i nastalih posledica?

D. Standardni i namenski softver

Prema načinu dobavljanja, razlikuju se namenski i standardni softver. Namenski softver je potpuno prilagođen korisniku i razvoj takvog softvera odvija se od nule oslanjajući se na poslovnu logiku, znanje i iskustvo firme za koju se proizvodi. Ovakav pristup može doneti značajnu prednost na tržistu. Mana ovakvog pristupa ogleda se u skupom razvoju i održavanju. Takođe, interoperabilnost sa drugim sistemima može biti problematična.

Nasuprot namenskom softveru, standardni softver je univerzalan i opslužuje više kupaca koji ga parametrizuju svojim potrebama tako da bude "dovoljno dobar" za posao za koji se koristi. Integracija ovakvog softvera je brza - svodi se samo na konfiguraciju i prilagođavanje. Unapređenje, održavanje i implementacija interoperabilnosti sa drugim sistemima se prebacuje na proizvođača standardnog softvera. Za razliku od namenskog softvera, standardni softver teško može biti osnova sticanja prednosti na tržištu jer i konkurencija može kupiti isti.

E. Preduslovi komponentizacije

Kako bi se komponentizacija uspešno sprovela, neophodno je postojanje standarda koji su praktični i ostvarivi [1]:

- OSGi - *Open Services Gateway initiative* - Java radni okvir za razvoj modularnih softverskih programa i biblioteka. Svaki paket (engl. *bundle*) je čvrsto povezana kolekcija klasa, jar fajlova (jar - *Java ARchive*) i konfiguracionih fajlova koji eksplicitno navode spoljne zavisnosti.
- EJB - *Jakarta Enterprise Beans*. Java API (engl. *Access Point Interface*) za modularnu konstrukciju softvera. Koristi se za enkapsulaciju poslovne logike aplikacije na serverskoj strani.
- DCOM - *Distributed Component Object Model*. DCOM je tehnologija kompanije *Microsoft* koja omogućava komunikaciju komponenti na umreženim računarima.
- ActiveX - ActiveX je stariji radni okvir kompanije *Microsoft* koji je omogućavao razmenu funkcionalnosti i podataka među aplikacijama preko internet pretraživača bez obzira na programske jezike kojima su pisane. ActiveX priključci (engl. *add-ons*) omogućavale su prvim pretraživačima prikaz multimedijalnih sadržaja.

Pored propisanih standarda, neophodno je i postojanje tržišta komponenti:

- *Eclipse Marketplace*
- *Firefox/Chrome* priključci
- *Google Play, Apple App Store* i dr.

III. Komponentni razvoj u *Python*-u - Osnove *setuptools* alata

PODRŠKA za komponentni razvoj aplikacija u *Python* programskom jeziku ogleda se u vidu *Python package* načina organizacije koda i upotrebe biblioteke za *packaging* poput *setuptools* [10].

Minimalna „jedinica građe" svakog *Python* programa je modul. Jedan *Python* fajl sa nastavkom *.py* predstavlja jedan modul [9].

Spajanje više modula vrši se dodavanjem specijalno definisanih *__init__.py* fajlova. Ovako kreirana struktura naziva se *Python package*. Svaki paket može u sebi sadržati jedan ili više modula čija je povezanost definisana *__init__.py* fajlovima. Ovakav način organizacije koda takođe omogućava pregledniju organizaciju funkcionalnosti aplikacije kao i veću enkapsuliranost pojedinačnih komponenti.

Kako bi krajnji korisnik mogao da koristi neki paket, neophodno je kreirati određene dodatne fajlove. U fajlu *pyproject.toml* definisane su neophodne alatke za ispravan *build* nekog paketa. Svaki *pyproject.toml* sadrži:

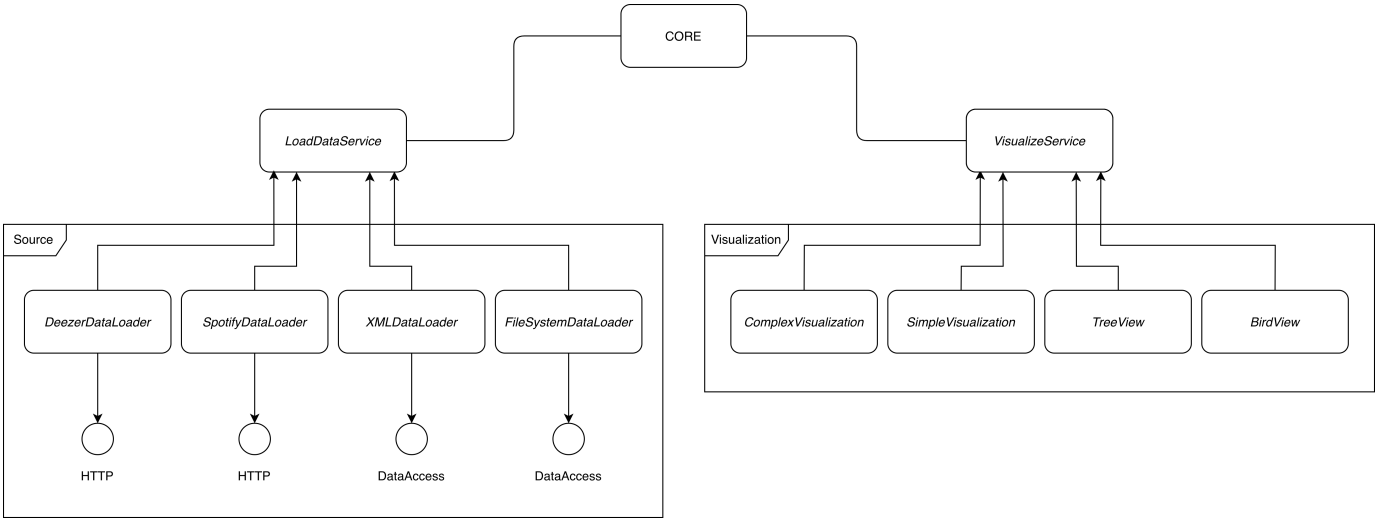
- *Build system* poput *setuptools*
- *Distribution system* poput *wheel* ili *egg* [9]

Ispravan rad pojedinačne komponente u *Python*-u može direktno zavisiti od drugih komponenti. Ovu uvezanost neophodno je definisati u *setup.cfg* fajlu. Pored toga, taj fajl takođe sadrži i metapodatke o paketu poput verzije, naziva komponente, njenog autora,...

S obzirom na to da se očekuje da svaki paket sadrži minimum informaciju o nazivu i verziji, poželjno je biti precizan prilikom definisanja zavisnosti (navesti sa kojom verzijom neophodne komponente naša komponenta funkcioniše).

IV. Implementacija aplikacije za vizualizaciju podataka - *GraphIt*

VODEĆA komponenta *GraphIt* aplikacije je takozvana CORE komponenta koja preko definisanih ulaznih jedinica koristi komponente iz kategorija za prikaz i vizualizaciju i na taj način ostvaruje predviđenu zajedničku funkcionalnost. Na slici 1 prikazane su konkretne komponente kojima sistem raspolaže, smer njihove međusobne interakcije, kao i način na koji su grupisane.



Slika 1. Dijagram komponenti *GraphIt* aplikacije

A. CORE komponenta

Jezgro aplikacije (CORE komponenta, platforma) obezbeđuje infrastrukturu u koju se ugrađuju različite kategorije komponenti u vidu *plugin*-a s ciljem formiranja jedinstvene funkcionalne celine. Glavni zadatak jezgra je definisanje servisnih ugovora (interfejsa) i modela podataka preko kojih se uspostavljaju pravila komunikacije sa ulaznim komponentama koje implementiraju neke od formulisanih ugovora. Implementacijom odgovarajućeg interfejsa, komponenta pruža skup funkcionalnosti koji je specijalizuje za adekvatnu namenu.

Platforma osnovne *GraphIt* aplikacije koncipira model, odnosno međureprezentaciju podataka u formi grafa, i dva interfejsa, jedan za pružanje usluga učitavanja i parsiranja, a drugi za obezbeđivanje grafičke vizualizacije podataka. Svaki od tri navedena elementa pojedinačno je analiziran u tekstu koji sledi.

1) Model podataka: Model podataka predstavljen je kroz klasu *Graph* koja u sebi sadrži ugnježdene klase *Vertex* i *Edge*. *Vertex* modeluje čvor grafa koji poseduje jedinstveni identifikator, naziv, tip i atribute u vidu parova ključ-vrednost. *Edge* predstavlja granu koja spaja dva čvora grafa. Svojstva grane su jedinstveni identifikator, vrednost, polazni i odredišni čvor.

Klasa *Graph* pruža servise za dobijanje informacija o usmerenosti i povezanosti grafa, stepenu i broju čvorova, broju grana, metode za dodavanje i uklanjanje čvora ili grane, kao i dobavljanje korena ukoliko je učitani graf ima strukturu stabla. Moguće je kreirati i podgraf kao rezultat pretrage ili filtriranja pozivanjem funkcija *create_search_graph* i *create_filter_graph*. Na slici 2 ilustrovane su gore navedene klase i nabrojane njihove metode.

2) Interfejsi: Servisni ugovori za komunikaciju platforme sa ulaznim komponentama definisani su u modulu *services* i realizovani korišćenjem apstraktnih klasa i metoda. *Service* je bazna klasa koja deklariše apstraktne metode za dobavljanje osnovnih informacija o *plugin*-u - identifikator i naziv. Ovu klasu nasleđuju specijalizovani servisni ugovori.

LoadDataService svojom apstraktnom metodom *load_data* pruža interfejs za učitavanje i parsiranje podataka u strukturu međureprezentacije. Ulazni parametar je izvorište podataka, a povratna vrednost graf definisan modelom u koji su parsirani podaci iz prosleđenog izvora. Komponente iz kategorije za učitavanje i parsiranje podataka implementiraju ovaj servisni ugovor.

VisualizeService putem apstraktne metode *visualize* pruža interfejs za grafički prikaz podataka parsiranih u međureprezentaciju formulisanu modelom. Ulazni parametar je struktura grafa, a povratna vrednost *string* reprezentacija *JavaScript* koda koji se ugrađuje u odgovarajući deo jezgra namenjen vizuelnom prikazu podataka.

B. Komponente za učitavanje i parsiranje podataka

Kategorija komponenti koje sprovode servisni ugovor za dobavljanje podataka nasleđivanjem *LoadDataService* klase jezgra, i implementiranjem njenih apstraktnih metoda. Na ovaj način realizovana komponenta specijalizuje se za učitavanje skupa podataka iz željenog tipa izvora i njihovo parsiranje u međureprezentaciju strukture grafa.

1) XMLDataLoader: Komponenta iz grupe za učitavanje i parsiranje, kod koje se kao izvor podataka koriste XML dokumenti proizvoljnog ili specificiranog formata prepoznatog na osnovu postojanja atributa *ref* i *id* unutar XML elemenata. Definiše klasu *LoadXMLData* koja nasleđuje *LoadDataService*. Implementacija *load_data* metode od prosleđenog XML dokumenta formira *ElementTree* stablo upotrebom *Python ElementTree* modula [11] i proverava format u zavisnosti od kog se konstruiše cikličan graf ili klasična hijerarhijska struktura stabla. Čvorovi *ElementTree* stabla parsiraju se u čvorove grafa koji se povezuju granama na odgovarajući način. Naziv i atributi XML elemenata mapiraju se na korespodentne atribute objekata klase *Vertex*. Povratna vrednost metode *load_data* je kreirani graf.

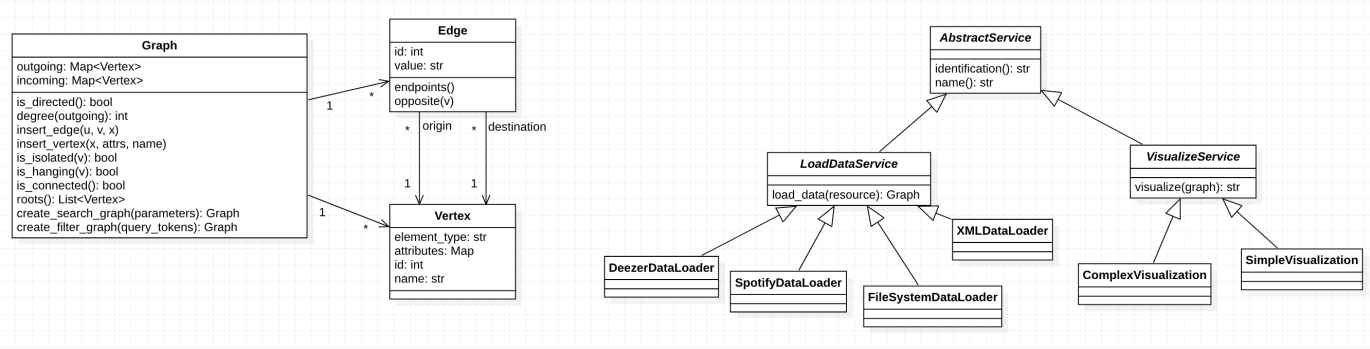
XML dokumenti po svojoj prirodi imaju hijerarhijsku organizaciju elemenata stoga je stablo najpogodnija struktura za njihovu reprezentaciju [11]. *XMLDataLoader* komponenta dodatno je osposobljena da kreira cikličan graf od XML dokumenta bez obzira na ograničenje njegove nativne strukture. Ovo se postiže uvođenjem atributa identifikatora - *id* i reference - *ref* unutar XML elemenata. U *ref* svojstvo upisuje se *id* vrednost elementa koji se referencira čime se formiraju dodatne veze.

2) FileSystemDataLoader: Komponenta koja omogućava učitavanje podataka o fajlovima nekog lokalnog direktorijuma. Samim tim, nasleđuje apstraktnu klasu *LoadDataService* i implementira njenu metodu *load_data(resource)*.

Resource u ovoj implementaciji predstavlja putanju ka željenom direktorijumu. Nakon prosleđivanja putanje, *FileSystemDataLoader* iterira kroz fajlove i njihove atribute i popunjava *Graph* strukturu ovim podacima. Čvorove dobijene strukture tipa graf predstavljaju folderi i fajlovi, a grane pripadnost fajlova folderima. Zbog prirode podataka, dobijeni graf je ujedno i stablo. Atributi koje sadrže čvorovi grafa su:

- *'id'* - jedinstveni identifikator svakog čvora (*hash*-irana putanja)
- *'path'* - puna putanja do fajla/foldera
- *'name'* - ime fajla/foldera
- *'type'* - tip fajla (prazan atribut kod foldera)
- *'size'* - veličina fajla/foldera
- *'date'* - datum kreiranja fajla/foldera

3) DeezerDataLoader i SpotifyDataLoader: Ove dve komponente funkcionišu po istom principu i nasleđuju istu apstraktnu



Slika 2. Dijagram klasa *GraphIt* aplikacije

klasu (*LoadDataService*), ali učitavaju podatke sa dva različita izvora. Cilj obe komponente je da podatke sa eksternog (veb sajta *Deezer* odnosno *Spotify*) izvora učitaju podatke o određenoj muzičkoj plejlisti u *Graph* strukturu kompatibilnu sa CORE komponentom aplikacije.

Za učitavanje podataka, neophodno je uneti URL adresu željene plejliste — to predstavlja *resource* argument funkcije *load_data(resource)*. Komponenta zatim vrši poziv ka eksternom izvoru podataka putem API biblioteke omogućenom od strane samog izvora. Dobijeni podaci u formi rečnika se zatim filtriraju i preslikavaju na strukturu tipa grafa. Dobijeni graf ima tri tipa čvora — *artist* (umetnik/izvođač pesme), *song* (numera) i *playlist* (plejlista). U sredini grafa se nalazi čvor tipa *playlist* i svi drugi čvorovi su povezani direktno ili indirektno sa njim.

- *'id'* - jedinstveni identifikator svakog čvora
- *'type'* - tip čvora
- *'name'* - naziv numere/ime izvođača/naziv plejliste
- *'description'* - opis
- *'url'* - URL putanja ka elementu
- *'followers_num'* - broj obožavalaca koji ima element
- *'tracks_num'* - broj numera (samo kod plejliste)
- *'duration'* - trajanje (samo za numere i plejlistu)

C. Komponente za prikaz učitanih podataka

Kategorija komponenti koje sprovode servisni ugovor za grafički prikaz podataka nasleđivanjem *VisualizeService* klase jezgra, i implementiranjem njenih apstraktnih metoda.

Na ovaj način realizovana komponenta specijalizuje se za određeni oblik grafičkog prikaza podataka međureprezentacije sa odgovarajućim stepenom detaljnosti pruženih informacija.

1) SimpleVisualization: Komponenta iz grupe za grafičku vizuelizaciju podataka, koja prikazuje osnovne informacije o čvorovima i granama. Definiše klasu *SimpleVisualization* koja nasleđuje *VisualizeService*. Metoda *visualize* kao ulazni parametar prima objekat klase modela *Graph* na osnovu čijih čvorova i grana kreira odgovarajuće vizuelne elemente upotrebom D3 biblioteke [4].

Čvorovi grafa su predstavljeni kao krugovi - D3 *circle* oblik [5], koji u sebi sadrže naziv, dok se prelaskom miša preko elementa mogu videti vrednosti atributa. Veze između elemenata modelovane su linijama - D3 *line* [5]. Pomeranje i zumiranje prikaza omogućeno je upotrebom i parametrizacijom metoda D3.*zoom* [6] i D3.*drag* [7]. Za modelovanje odbojnosti između čvorova korišćene su *force* metode [8]. Povratna vrednost *visualize* metode je string reprezentacija *JavaScript* koda koji se ugrađuje u odgovarajući deo jezgra namenjen jednostavnom grafičkom prikazu podataka.

2) ComplexVisualization: Pripada kategoriji komponenti za grafičku vizuelizaciju podataka, koja prikazuje detaljne informacije o čvorovima i granama. Definiše klasu *ComplexVisualization* koja nasleđuje *VisualizeService*. Metoda *visualize* kao ulazni parametar prima objekat klase modela *Graph* na osnovu čijih čvorova i grana kreira odgovarajuće vizuelne elemente upotrebom D3 biblioteke [4].

Čvorovi su predstavljeni kao pravougaonici - D3 *rect* oblik [5], koji u sebi sadrže naziv i vrednosti svih atributa, a veze idmeđu njih linijama - D3 *line* [5]. Pomeranje i zumiranje prikaza omogućeno je upotrebom i parametrizacijom metoda D3.*zoom* [6] i D3.*drag* [7]. Za modelovanje odbojnosti između čvorova upotrebljene su *force* metode [8]. Povratna vrednost *visualize* metode je string reprezentacija *JavaScript* koda koji se ugrađuje u odgovarajući deo jezgra namenjen grafičkom prikazu podataka sa visokim stepenom detaljnosti.

3) TreeView: Komponenta za prikaz podataka definiše klasu *TreeView* koja nasleđuje *VisualizeService* iz modela vodeće platforme. Metoda *visualize* prima grafičku međureprezentaciju podataka koje mapira na odgovarajuće elemente stabla iz biblioteke *jqtree* [12].

Povratna vrednost *visualize* metode je *JavaScript* kod koji se ugrađuje u deo jezgra zadužen za pregled hijerarhijske povezanosti podataka. Za svaki nivo hijerarhije podaci se dinamički renderuju tako da, iako je u pitanju prikaz u vidu stabla, podaci ne moraju isključivo obrazovati strukturu stabla. Potrebno je naći čvor najvećeg stepena počevši od koga se može doći do svih ostalih čvorova i eventualne viseće čvorove ukoliko graf nije povezan. Informacije o navedenom tipu čvorova komponenti *TreeView* obezbeđene su od strane odgovarajućih metoda modela.

4) BirdView: Vizualizaciona komponenta definiše klasu *BirdView* koja nasleđuje *VisualizeService* iz modela. Metoda *visualize* kao ulazni parametar prima objekat klase modela *Graph* na osnovu čijih čvorova i grana kreira odgovarajuće vizuelne elemente upotrebom D3 biblioteke [4].

Čvorovi su predstavljeni kao krugovi - D3 *circle* [5] koji u sebi sadrže naziv. Namena ove komponente je umanjen pregled celokupnog skupa podataka iz tzv. ptičije perspektive. Povratna vrednost *visualize* metode je *JavaScript* kod koji se ugrađuje odgovarajući deo jezgra zadužen za uopšten pregled podataka i veza.

V. Prednosti i izazovi komponentnog razvoja *GraphIt* aplikacije

ZAHVALJUJUĆI komponentnom pristupu razvoja *GraphIt* aplikacije, omogućeno nam je lako proširivanje aplikacije novim funkcionalnostima u vidu *plugin*-a koje se brzo i jednostavno instaliraju i time ugrađuju u jezgro aplikacije. U slučaju pojavljivanja novog izvora podataka ili načina vizuelne predstave podataka, razvoj novih nezavisnih komponenti i komunikacija sa njima odvija se po jasno propisanim intefejsima i protokolima. Takođe, jezgro aplikacije je nezavisno i radi bez obzira na broj i vrstu trenutno instaliranih komponenti. Dakle, instalirane komponente nisu preduslov za ispravan rad aplikacije.

Iako je komunikacija glavnog jezgra aplikacije sa svim instaliranim komponentama preko interfejsa uniformna i jednostavna, otežavajuća okolnost bila je upravo definisanje tih univerzalnih servisnih ugovora. Sve specijalizacije nekog tipa komponenti moraju da se prilagode i implementiraju jedan te isti interfejs koji pripada tom tipu. Na primer, komponente za učitavanje moraju da popune istu međureprezentaciju podataka definisanu u jezgru aplikacije bez obzira na raznolikost

samih izvorišnih podataka i njihovih struktura. Analogno važi i za komponente za vizualizaciju, koje moraju da obave grafički prikaz na osnovu međureprezentacije podataka i vrate uniforman *JavaScript* kod koji se može ugraditi u deo platforme namenjen prikazu.

VI. Upotreba *GraphIt* aplikacije

Upotreba *GraphIt* aplikacije svodi se na izbor *plugin*-a za učitavanje odabranog skupa podataka, a zatim vizualizaciju kreirane međureprezentacije sa odgovarajućim stepenom detaljnosti, primenom adekvatnog *plugin*-a za prikaz. Predstavljeni sadržaj korisniku daje temeljan uvid u povezanost entiteta od interesa koji se može iskoristiti za izvođenje brojnih zaključaka i izveštaja. Rasute informacije u vidu vrednosti atributa skoncentrisane su na jedinstven način što znatno olakšava analizu i interpretaciju. Izgled *GraphIt* aplikacije prikazan je na slikama 3 i 4.

VII. Zaključak

U Ovom radu analizirane su strategije razvoja baziranog na komponentama. Publici su ukazane prednosti koje takav pristup donosi poput fleksibilnosti, robusnosti, skalabilnosti, efikasnosti i isplativosti. Kroz konkretan primer implementacije i primene *GraphIt* aplikacije za vizualizaciju podataka, prikazana je praktična primena navedenih koncepata. Opisane su dve kategorije specijalizovanih komponenti (za učitavanje podataka i grafički prikaz) i njihovo postavljanje na zajedničku centralizovanu platformu. Takođe su spomenuti slučajevi upotrebe *GraphIt* softvera. Za kraj, skrenuta je pažnja i na izazove sa kojima se programeri mogu susresti prilikom projektovanja sistema oslanjajući se na navedene mehanizme.

Dalja unapređenja osnovne *GraphIt* aplikacije usmerena su ka definisanju novih funkcionalnih celina koje bi proširile postojeće mogućnosti upotrebe. Analizirano softversko rešenje u budućnosti bi se moglo koristiti za čuvanje, dodavanje i izmenu podataka u realnom vremenu (povezivanjem sa eksternom bazom podataka/lokalnim fajlovima sa potpunim pravima pristupa i izmena), donošenje zaključaka i predikcija na osnovu izložene grafičke strukture i kolaborativni rad grupe korisnika (uređivanje podataka od strane više korisnika u realnom vremenu).

Literatura

[1] C. Szyperski, Component Software: Beyond Object-Oriented Programming, Addison-Wesley Longman Publishing Co., Inc., 2002 - Osnovi principi komponentnog razvoja i standardi

[2] Lau, Kung-Kiu, and Simone Di Cola. An Introduction to Component-Based Software Development. Vol. 3. # N/A, 2017. - Uvod u komponentni razvoj

[3] Crnkovic, I. (2001). Component-based software engineering - new challenges in software development. Software Focus - Izazovi pri komponentnom razvoju

[4] https://d3js.org/ - Dokumentacija D3.js biblioteke za grafički prikaz podataka

[5] https://github.com/d3/d3/blob/master/API.md#shapes-d3-shape - Dokumentacija za D3 *shape* oblike

[6] https://github.com/d3/d3/blob/master/API.md#zooming-d3-zoom - Dokumentacija za D3 *zoom* metode za zumiranje prikaza

[7] https://github.com/d3/d3/blob/master/API.md#dragging-d3-drag - Dokumentacija za D3 *drag* metode za pomeranje prikaza

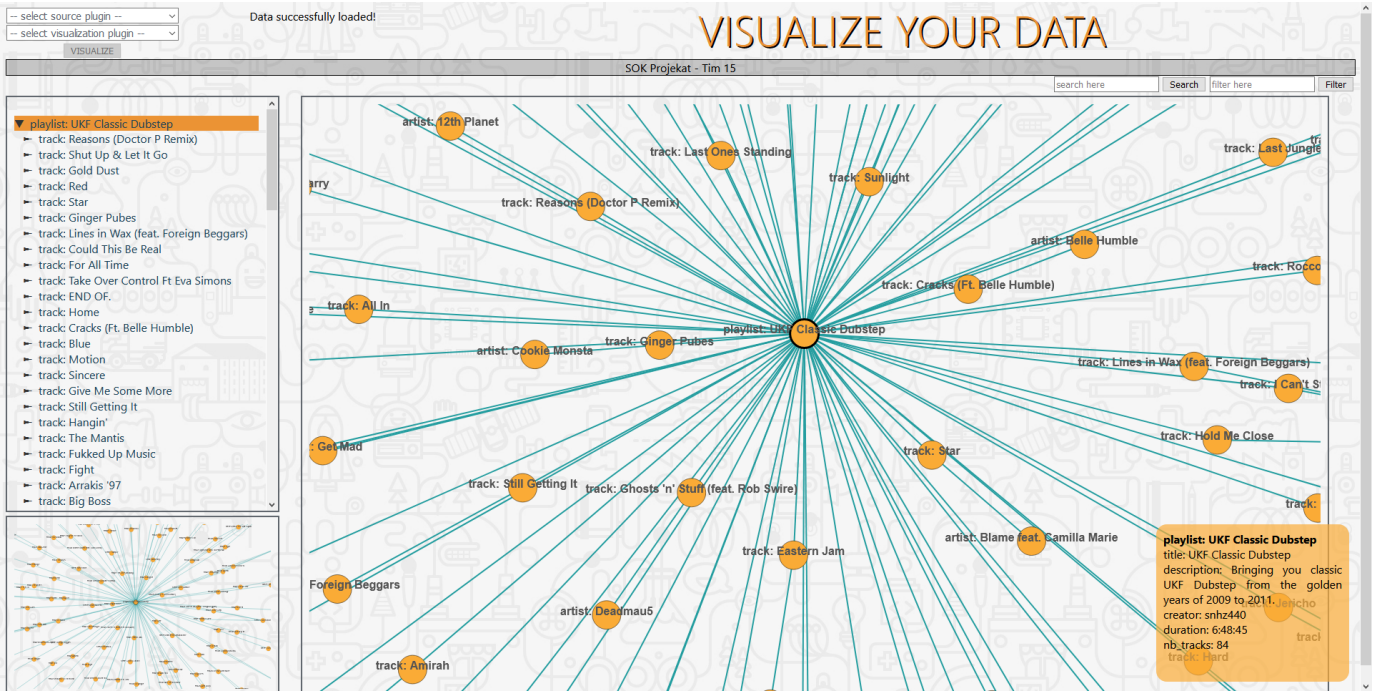
[8] https://github.com/d3/d3/blob/master/API.md#forces-d3-force - Dokumentacija za modelovanje D3 *force* sila

[9] https://docs.python.org/3/ - Python dokumentacija

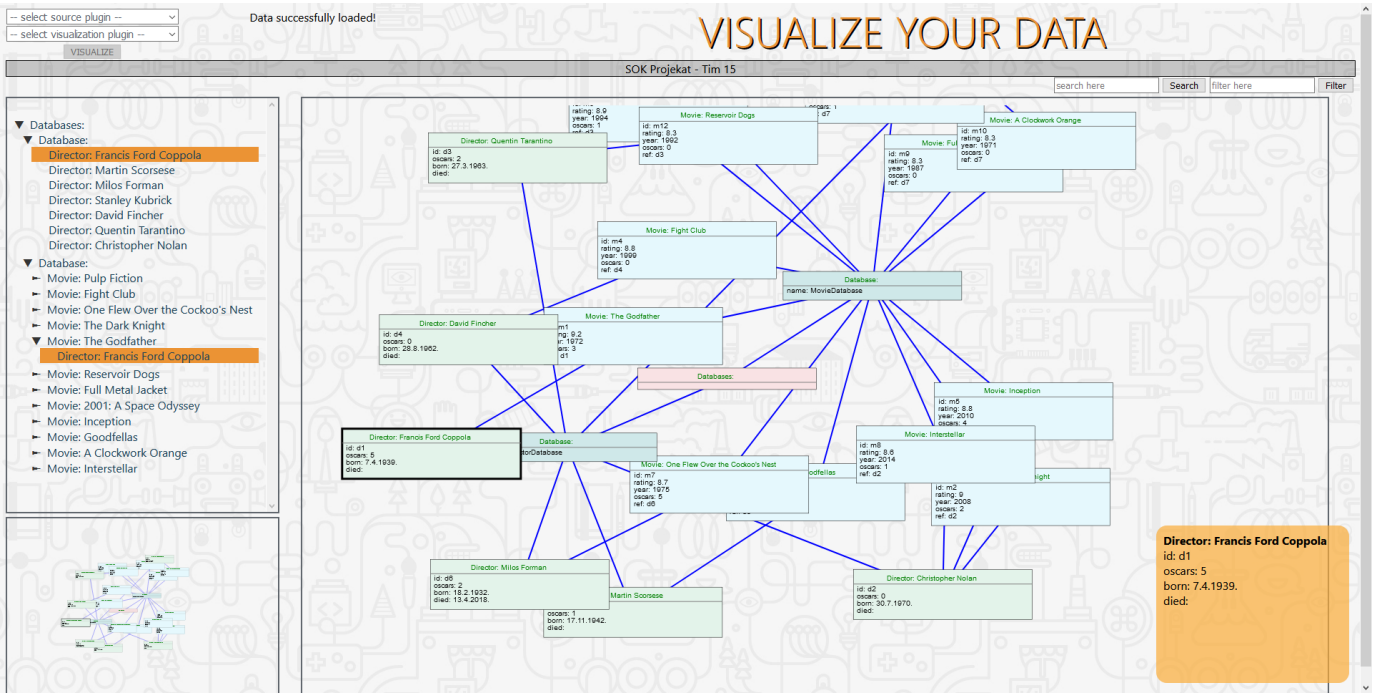
[10] https://setuptools.readthedocs.io/en/latest/ - Setuptools dokumentacija

[11] https://docs.python.org/3/library/xml.etree.elementtree.html - Parsiranje XML dokumenta u *ElementTree* stablo

[12] https://mbraak.github.io/jqTree/ - jqTree dokumentacija - Biblioteka koja omogućava hijerarhijski prikaz u formi stabla



Slika 3. *GraphIt* aplikacija. Korišćeni *plugin*-ovi: *DeezerDataLoader*, *SimpleVisualization*, *TreeView* i *BirdView*.



Slika 4. Slika *GraphIt* aplikacije. Korišćeni *plugin*-ovi: *XMLDataLoader*, *ComplexVisualization*, *BirdView* i *TreeView*.