UDC: 004.4:519.682 Info M str. 46-50

PRIMENA TEST-DRIVEN DEVELOPMENT-A U RAZVOJU SQL SERVER BAZE PODATAKA
APPLICATION OF TEST-DRIVEN DEVELOPMENT IN SQL DATABASE DEVELOPMENT

Marko Alavanja, Saša D.Lazarević

REZIME: Testiranje je jedan od najbitnijih delova izrade nekog sistema. Razvoj baze podataka je osnova za izradu bilo kakovogsoftverskog sistema. Da bi baza bila funkcionalna mora da bude pravilno izrađena, testrianjem baze podataka dobijamo željeni kvalitet. Kroz ovaj rad pokazaćemo kako iskoristiti nove pristupe i tehnologije koje se nude. Izradićemo bazu podataka pomoću Test-Driven Development pristupa i pokazati prednosti i mane korišćenja jednog modernog pristupa izrade baze podataka . Pristup Test-Driven Development u razvoju baze podataka je postao najkorišćeniji zbog uspesnosti izrade kvalitetnte baze podataka. Ovaj deo rada predstavlja teorijsku osnovu izrade SQL Server baze podataka pomoću ovog pristupa.
KLJUČNE REČI: testiranje, SQLServer, razvoj baza podataka, Test-driven development
ABSTRACT: Testing is one of most important part in develop of some sistems. Developing of database iz primary step for somesoftware sistem development. If that datebase is funcitonal must be deploy by steps, testing the database we get wanted quality.Through this paper, we will show how to take advantage of the new approaches and technologies on offer. We will create a database using the Test-Driven Development approach and show the advantages and disadvantages of using a modern approach to database creation. Test-Driven Development in database development is most used because it succesful in quality database development.This part of the paper presents the theoretical basis for creating an SQL server database using this approach.
KEY WORDS: testing, SQL Server, Database development, Test-driven development

1. UVOD

Razvoj baze podataka je jedan od najbitnijih delova razvoja bilo kog softverskog sistema. Od načina razvoja zavisi i sam rad baze kao i performanse koje se mogu postići. Da bi isprojektovali što bolju bazu podataka potrebno jevrišiti testiranja. Testiranje se radi da bi pre nego što se finalizira implementacija proveri rad baze podataka. Postoje tri tipa testrianja baze podataka a to su testiranje strukture, funkcionalnosti i performansi baze podataka. U ovom radu će se pristupiti izradi baze podataka pomoću pristupa *Test- Driven Development*(TDD). Ovaj pristup je prvi put prikazan i objašnjenu knjizi Kenta Beka (Kent Beck) iz 2003. godine. Način pisanja testova kod ovog pristupa je da pre svake implementacije pišu testovi. Testovise izvršavaju i ako prođu nastavlja se sa pisanjem,a ako "padnu" pristupa se re-implementiranju i pravljenu malih promena koje su dovoljne da prođu testovi. Zbog ogromnih rezultata koji je ovaj pristup ostvario svako ko želi ozbiljno da se posveti testiranju primeniće ovaj pristup. Test-Driven Development je spoj Test-First Discipline (TFD) i refakotrisanja. Kada se prvi put implementira nova funkcija, prvo pitanje koje se postavlja je da li postojećastruktura softvera najbolja mogućastruktura koja omogućava da se implemetira ta funkcionalnost. Ako jeste, upotrebiće se TFD. U suprotnom, refaktoriše se lokalno da bi se promenio deo strukture na koji utiče nova funkcija, što omogućava da se lakše doda ta novu funckicja.

2. PRIMENA TEST-DRIVEN DATABASE DEVELOPMENT-A

2.1. *Test-Driven Database Development*

Test Driven Database Development je pristup kod koga se prvo pišu testovi. To znači da programer prvo piše potpuno automatizovane testove koji se izvršavaju, aposle toga piše funkcionalni kod i refaktoriše ako je to potrebno. TDD baze podataka nije tako lako sprovesti kao TDD programskog koda. Prvi problem jesu alati. Iako neki alati kao što je DBUnit idu u korak sa tehnologijama idalje nisu toliko prihvaćene novine. Ali očekivanja su da će kroz par godina porasti interesovanje za ovim pristupom. Drugi problem je taj što ljudi koji se profesionalno bave razvojem baza podataka orjentisani na tradicionali način razvijanja baza podataka. Treći problem jeste taj da većina ljudi koji se bave razvojom baza preferiraju model-driven a ne test-driven pristup. To je zbog toga što test-driven pistup nije toliko rasprostranjen a i zbog toga što ti ljudi razmišljaju vizuelno i preferiraju model-driven pristup.

2.2. *Agile Model-Driven Development*

Agile Modeling (AM) je praktično orijentisana metodologija za efektivnije modelovanje i dokumentaciju softverskih sistema. AM je skup praktičnih postupaka vođenih principima i vrednostima koje bi softverski profesionalci trebalo da primenjuju iz dana u dan. To nije skup pravila kako kreirati model već daje savete kako biti efektivniji modelar. AM pristup ne smanjuje modelovanje, ustvari mnogo će se više modelovati nego pre upoznavanja sa AM-om. AM njie skup pravila, to je više kao umetnost a ne nauka. Agile model-driven development je agilna forma Model-driven development-a.Umesto stvaranja opsežnih modela pre pisanja koda, prave se agilni modeli koji su dovoljno dobri.AMMD je evoluciona altrenativa tradicionalnom MDD-u.

2.3. *Test-Driven Development i Agile Model-Driven Development*

Kako uporediti TDD sa AMMD-om. Razmotrimo neke pretpostavke:

- TDD skraćuje povratnu spregu programiranja, dok AMMD skraćuje povratnu spregu modelovanja;
- TDD pruža detaljnu specifikaciju(testove), dok AMMD pruža tradicionalnu specifikaciju (modele podataka);
- TDD promoviše razvoj visoko kvalitetnog koda, dok AMMD promoviše visoku komunikaciju sa ostalim učesnicima na projektu;

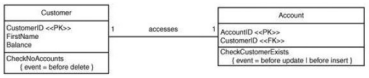
- TDD pruža konkretne dokaze da softver radi, dok AMMD pruža bolje međusobno razumevanje učesnika u razvoju softvera;
- TDD „razgovara“ sa programerima, dok AMMD „razgovara“ sa stručnjacima za podatke i domenskim ekspertima;
- TDD pruža konkretne povratne informacije o redosledu minuta narudžbe, dokAMDD omogućava usmenu povratnu informaciju o minutima narudžbe (konkretne povratne informacije zahtevaju od programera da bude zavistan od upotrebe nekih tehnika koje nisu AM tehnike);
- TDD pomaže da se održi čistastruktura softvera fokusirajući se na kreiranje izvršivih i testnih operacija, dok AMMD pruža mogućnost da se razmišlja o većimprojektantsko/arhitekturalnim problemima pre pisanja koda;
- TTD nije vizuelno orjentisan, dok je AMMD vizuelno orjentisan;
- Obe tehnike su nove pa ih tradicionlni programeri mogu smatrati kao pretnju;
- Obe tehnike podržavaju evolucioni razvoj;

Koji pristup od ova dva iskoistiti? Odgovor zavisi od toga kako je tim softverskih inženjera orijentisan.Ako jeviše orjentisan ka crtanju i vizuelizaciji onda treba upotrebiti AMMD, a akoje više orjentisan ka pisanju teksta onda je sigurno vama primamljiviji TDD. Naravno da se većina pristupa nalazi između ova dva slučaja tako da je kombinovana upotreba najbolje rešenje. Treba iskoristiti u datom trenutku onaj pristup koji po vama ima više smisla i upotrebiti u konkretnom projektu. Kratko rečeno najbolje je koristiti oba pristupa zajedno i tako iskoristiti ono najbolje iz njih.

Kako kombinovati ova dva pristupa? Odgovor je da AMMD treba iskoristiti za kreiranje modela sa ostalim akterima u projektu, dok TDD treba iskoristiti za razvijanje čistog, upotrebljivog koda. Rezultat će biti da će te imati visoko kvalitetan, upotrebljiv sistem koji će zadovoljiti sve razumne potrebe krajnjih korisnika..

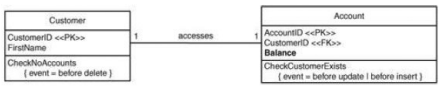
2.4. *Refaktorisanje baza podataka*

Refaktorisanje baze podataka je jednostavna promena sheme baze koja poboljšava strukturu ali zadržava semantiku ponašanja isemantiku informacija. Interesantno je da je baš zbog toga refaktorisanje baza podataka komplikovanije nego refaktorisanje programskog koda jer ono zadržava samo ponašanje semantike. Refaktorisanje je teško uraditi ako se ne pišu testovi. Može se slobodno reći da nema poente promene zvati refaktorisanja, ako ne postoje testovi da to potvrde. Promene se naravno mogu izvršiti, ali njih ne treba zvati refaktorisanje. Težina refaktorisanja zavisi i od toga da li je *single-application* ili *multi-application* baza podataka. Ako je *single-application* ondaaplikacija samo komunicira sa bazom i moguće je refaktorisati i razviti istovremeno. A ako je *multi-application*, onda baza komunicira sa mnogo različitih programa i nije moguće obezbedti istovremeno refaktorisanje i razvijanje kao u prvom slučaju. U ovom slučaju je potrebno da postoji neki tranzicioni period u kome su dostupna i stara shema i nova refaktorisana shema.Prethodno se može pokazati na jednostavnom primeru, koji je ilustrovan za oba slučaja.



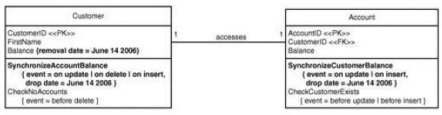
Primer: (Slika1)

Single-application: premešta se jedna kolona iz jedne u drugu tabelu. U ovom slučaju to je lako uraditi jer samo jedna aplikacija komunicira sa bazom i može se istovremeno refaktoristati i baza i programski kod aplikacije. Ovde je najbolje iskoristiti dva stručnjaka prvi će raditi na bazi i drugi koji će raditi na kodu aplikacije. Prvi korak je pokrenuti već napisane testove i videti da li “prolaze“. Sledeće šta treba uraditi jeste napisati testove,jer se koristTDD pristup. Treba napisati test za novu kolonu koja je premeštena u drugu tabelu a to je *Account. Balance*. Pokreće se test i on neće proći i zatim se uvrštava ta nova kolona u tabelu i pokreće se test koji prolazi. Refakotriše se test koji verifikuje da *Customer.deposit* radi pravilno sa *Account.Balance* kolonom kao sa *Customer.Balance* kolonom. Videće se da testovi neće proći.Stoga treba preraditi *Deposit* funkcionalnost da bi radila sa *Account.Balance*. I ostatak će se preraditi kao i što je bilo za *Customer.Balance*. Kada se pokrene aplikacija potrebno je uraditi kopiju za *Customer.Balance* kolonu za svaki slučaj i kopirati podakte u kolonu *Account. Balance*. Treba ponovo pokrenuti testove koji će proveriti da li je migracija obavljena valjano. Da bi kompletirali promene na shemi trebaizbaciti*Customer.Balance* kolonu, i pokrenuti sve testove i ispraviti ako je nešto potrebno.



(Slika2)

Multi-application: ova situacija je mnogo komplikovanija u odnosu na prethodnu. Da bi se obavilo refakotisanje treba uraditi sve kao u predhodnom slučaju samo što se ne briše kolona*Customer.Balance*. Umesto toga imamo obe kolone tokom tranzicionog perioda u kome se daje vreme da svi razvojni timovi unaprede i ponovo razviju svoje aplikacije.Izgled sheme u tranzicionom periodu dat je na slici 3 s tim što se mora napomenuti da imaju dva trigera koja će se koristiti u tranzicionom periodu da bi kolone bile sinhronizovane.



(Slika3)

Nakon tranzicionog perioda obrisaće se originalna kolona i trigeri i dobiti rezultat shemu kao na slici 2. Ovo brisanje je moguće uraditi samo ako se sve istestira i sigurno je da će sve normalno raditi.

Kategorije refaktorisanja baza podatka:

- Structural – promena definicije jednog ili više prikaza tabela,
- Data Quality – promena koja poboljšava kvalitet podataka sadržanih u bazi podataka,

- Referential integrty –promena koja osigurava da referencirani red postoji u drugoj tabeli i/ili da osigura da red koji više nije potreban propisno obrisan,
- Architectural – promena koja poboljšava ukupan način na koji spoljni programi komuniciraju sa bazom podataka,
- Method – promena metoda(uskladištene procedure, funkcija i trigera) koja poboljšava kvalitet,
- Non-refactioring transformation – promena u shemi baze podataka koja utiče na semantiku baze.

Moguće je olakšati refaktorisanje u multi-application slučaju. U ovom slučaju shema baze podataka povezana je i saaplikacijom, perzistentim okvirima, ORM alatima, drugim bazama podataka, testnim kodom i čak sama sa sobom. Efektivan način da se smanji povezanost sheme baze podataka jeste da se enkapsulira pristup bazi. To se radi tako što eksterni programi pristupaju bazi preko perzistentnog sloja. Može biti implementiran na razne načine kao što su: data access objekti(DAOs) koji implementiraju potreban SQL kod, pomoću okvira, pomoću uskladištenih procedura i pomoću veb servisa. Ne mogu da se smanje na nulu,ali može da se svede na nešto što je upravljivo.

2.5. *Slojevi enkapsulacije baze podataka*

Sloj enkapsulacije baze podataka skrivaju detalje implementacije baze podataka, uključujući i fizičke sheme poslovnog koda. Ustvari pruža poslovne objekte sa postotjanim servisima – mogućnost čitanja podataka, upisivanja podataka, brisanja podataka iz izvora podataka,bez neophodnosti da poslovni objekti znaju bilo šta o konkretnoj shemi baze. Idealno, poslovni podaci ne znaju ništa o svom skladištenju već se to “po automatizmu” dešava. Slojeve enkapsulacije ne treba izbegavati, oni se obično koriste i u prostim i složenim aplikacijama. Slojevi enkapsulacije su bitni za svakog AMMD programera koji treba biti svestan da postoje i spreman je da ih koristi.

Efektivan sloj enkapsulacije baze podataka pruža neke pogodnosti:

- Smanjuje povezanost između aplikacione objektne sheme i sheme baze podataka povećavajući sposobnosti programera da evoluira bilo koju od njih. Treba napomenuti da će povezanosti i dalje postojati, ali će biti drastično manja.
- Implementira sav data-related kod na jednom mestu, uključujiući i povezivanje podataka koje implementira između objekta i sheme podataka, što olakšava podršku svim promenama sheme baze podataka koje se dešavaju ili podržavaju promene vezane za performanse.
- Pojednostavljuje posao programera aplikacija. Sa slojem enkapsulacije baze podataka na pravom mestu, programerima ostaje samo da iskodira programski kod, a ne programski kod plus SQL kod.
- Omogućavaju programeru aplikacija da se fokusria na poslovne probleme a agilnom arhitekti baze podataka da se fokusira na bazu podataka. Naravno da obe grupe moraju da nastave da rade zajedno, ali svaka od njih može bolje da se fokusira na svoj deo posla.
- Daje vam zajedničko mesto, pored same baze podataka, za implementaciju poslovnih pravila orijentisanih na podatke.
- Koristi prednosti specifične baze podataka i funkcija, povećavajući performanse aplikacije.

A i ima naravno i neke nedostatke:

- Zahtevaju neku vrstu investicija, nebitno da li se ulaže novac, vreme ili napor.Morate ili da napravite, kupite ili preuzmete sloj za enkapsulaciju podataka.
- Često zahtevaju prilično čista mapiranja. Sloj enkapsulacije podataka može se pokvariti kada preslikavanja između apikacionih objekata i shema podatakapostanu složena.
- Oni mogu pružiti premalu kontrolu nad pristupom bazi podataka. Neke strategije enkapsulacije podataka, kao što je Enterprise JavaBeans-ov pristup perzistentnosti kojim se upravlja kontejnerom (CMP) (Roman, Ambler i Jevell 2002), previše inkapsuliraju pristup bazi podataka. Na primer, sa CMP-om ne postoji kontrolakada se sačuvava objekat —obično se sačuvava automatski kad god se promene vrednosti atributa objekta, iako možda to ne želite.

2.5.1 Arhitektura slojeva enkapsulacije

Postoje različite arhitekture, prva i najprostija jeste jedna aplikacija radi sa jednom bazom.U ovoj situaciji postoji najveći potencijal za fleksibilnost, tako datim trebada izabere strategiju implementacije, kao što su objekti za pristup podacima ili okvir postojanosti, koja najbolje odgovara vašoj situaciji. Štaviše, trebalo bi da budete u poziciji da razvijate i svoju shemu objekata i shemu baze podataka dok implementirate nove zahteve.

Mnogo realnija situacija je da više aplikacija pristupa jednoj bazi podataka. Ova situacija je česta u organizacijama koje imaju centralizovanu bazu podatka sa kojom sve aplikacije rade. Jos jedna realna situacija je ta da postoji više aplikacija koje rade sa više baza podataka. U tom slučaju pristupa se svim bazama koje postoje ako uopste postoje.

U ovim slučajevima postoji mogućnost da neke aplikacije neće iskoristiti sve prednosti slojeva enkapsulacije i umesto toga direktno će pristupati podacima. Neki od mogućih razloga su:

- Sloj enkapsulacije napisan je u nekom jeziku kome zastarele aplikacije ne mogu da pristupe.
- Nisu prepravljene neke od zastarelih aplikacija da bi koristile slojeve enkapsulacije baze podataka.
- Želite da koristite tehnologije, kao što je mogućnost masovnog učitavanja ili okvir za izveštavanje, koje zahtevaju direktan pristup šemi baze podataka. Imajte na umu da ovo može motivisati vaš tim da ponekad obiđe sloj enkapsulacije.

Poenta je da će neke aplikacije moći da iskoristeslojeve enkapsulacije, a neke neće. Još uvek postoje prednosti jer smanjujete spajanje i time smanjujete troškove razvoja i opterećenje održavanja.

Neke aplikacije već imaju postavljen sloj enkapsulacije, i u tom slučaju trebalo bi razmisliti o ponovnoj upotrebi tog pristupa umesto razvijanja svog. Imajući jedan sloj enkapsulacije koji sve aplikacije koriste za pristup svim izvorima podataka, potecijalno se smanjuje napor koji je potreban da se sheme baza podataka evoluiraju pomoću refaktorisanja baze poda-

taka jer postoji samo jedan sloj enkapsulacije za ažuriranje. Ako kupite sloj za enkapsulaciju, možda će te moći da smanjite ukupne troškove za licence ako radite samo sa jednim dobavljačem. Potencijalni nedostatak je što bi tim odgovoran za održavanje sloja enkapsulacije mogao postati usko grlo ako ne može ili ne želi da radi na agilan način.

2.5.2 Strategije implemetacije sloja enkapsulacije

Bez obzira da li se kupuje, razvija ili preuzima sloj za enkapsulaciju baze podataka, ključno je da administratoribaze podtaka kao i programeri aplikacija da razumeju različite strategije implementacije. Postoje četiri osnovne strategije koje bi trebalo razmotriti da koristite, a to su:

- *Brute force*
- Data access objekti (DAOs)
- Prezistentni okviri
- Servisi

U nastavku izneće se pregled svake strategije, pokazati kako pročitati jedan objekat iz baze pomoću strategija.

Brute force- ovaj pristup nije strategija enkapsulacije baze podataka, to je ono što se primenjuje kada nema sloja enkapsulacije baze podataka. Međutim to je validna opcija za pristup bazi podataka i zato je ubrojana među strategijama enkapsulacije. Naprotiv brute-force je najčešće korišćen pristup zbog svoje jednostavnosti i zato što pruža programerima potpunu kontrolu nad tim kako njihovi poslovni objekti komuiciraju sa bazom podataka.Zbog svoje jednostavnosti, ovo je veoma dobar pristup na početku projekta kada su zahtevi za pristup bazi podataka prilično jednostavni. Kako pristup bazi podataka postaju složeniji, mnogo bolje opcije su DAOs i perzistenti okviri. Osnovna strategija koja stoji iza brute-force pristupa je da poslovni objekti direktno pristupaju izvorima podataka, najčešće slanjem SQL ili OQL koda bazi podataka.

Data access objekti(DAOs)- enkapsuliraju pristupnu logiku baze podataka zatevanu od poslovnih objekata. Tipičan pristup je da postoji jedan data access objekat za svaki poslovni objekat. Izvodi se pomoću pravljenja klase koja implementira SQL ili neki drugi kod potreban za pristup bazi podataka, slično kao i brute force. Glavna prednost u odnosu na brute force je da poslovne klase nisu direktno povezane sa bazom, umesto toga, data access clase su povezane. Sasvim je uobicajeno i jednostavno da sami razvijate data accces objekte a mogu se i koristiti neki industrrisjki standardi. Prvo što je potrebno jeste da poslovni objekti proslede sebe kao parametar DAO-u. DAO zatim dobija vrednost za primarni ključ unutar baze podataka, a ako nije poznat, DAO treba da dobije dovoljno informacija od poslovnog objekta da bi ga identifikovao u bazi.DAO zatim gradi upit koji onda poziva u bazi podataka i sortira podatke u rezultujućem skupu ažuriranjem poslovnog objekta. DAO je nešto lakši za prihvatanje za agilnom DBA nego brute force jer je logika pristupa bazi podataka koncentrisana na jednom mestu. Programerima aplikacija biće i dalje podtreban neki nivo mentorstva, a moraće i dalje da se pomogne u postavljanju standarda kodiranja.

Perzistentni okviri – često nazivan i perzistenti sloj u potpunosti enkapasulira pristup bazi podataka iz poslovnih objekata. Umesto pisanja koda za implementaciju logike potrebene za pristup bazi podataka, definišu se metapodatci koji predstavljaju mapiranja.Meta podatci predstavljaju mapiranja svih poslovnih objekata kao i povezanost izmedju njih, takođe moraju postojati. Na osnovu ovih meta podataka, perzistntni okvir generiše pristupni kod bazi podataka koji je potreban da bi postojali poslovni objekti.U zavisnosti od okvira, ovaj kod se generiše na dva načina. Prvi je dinamički u vremenu izvođenja, a drugi statički u formi objekta za pristup podatcima, koji se zatim kompajliraju u aplikaciju. Prvi pristup pruža bolju fleksibilnost dok drugi pruža bolje performanse. Perzistentni okviri imaju različite karakteristike. Jednostavne će podržavati osnovne funkcialnosti kreiranje, čitanje, ažuriranje, brisanje (CRUD) za objekte kao i osnovnu kontrolu transakcija i konkurentosti. Napredne uključuju robusno upravljane greškama, prikupljanje veza sa bazom podataka, keširanje, XML podršku, mogućnosti generisanja šema i maprianja, i podršku za industrijsku-standardnu tehnologiju kao što je EJB. Važno pitanje u vezi sa dizajnom perzistentnih okvira je da li je perzistentost implicita i eksplicitna. Sa implicitnim pristupom, okvir automatski perzistira na poslovnim objektima bez njihovog znanja, ne moraju da traže da budu sačuvani, pročitani ili bilo šta drugo, jednostavno se dešava. Savršen primer je *Enterprise JavaBean* (EJB) koncept kontejnera perzistentosti: da bi se automatski opstao, entitetski bean treba da implementira standardni interfejs (kolekciju operacija) i da bude opisan u deskriptoru primene (XML dokument koji sadrži meta podatke) .Eksplicitnim pristupom, koji je daleko najčešći, objekti treba da naznače (ili da imaju nešto drugo da naznače) kada treba da budu sačuvani. Važno je primetiti da poslovni objekat treba samo da stupi u interakciju sa okvirom, većina okvira obično ima klasu pod nazivom *Database,PersistenceFramework* ili *PeristenceBroker* koja implementira svoj javni interfejs. Pristup perzistentihokvira čini posao agilnog administratora baze podataka malo složenijim, ali mnogo manje napornim. Od vas će se očekivati da instalirate, ako je potrebno,perzistetni okvir. Takođe ćete morati da radite sa administrativnim objektom da biste definisali i održavali metapodatke za mapiranje. U slučaju eksplicitno kontrolisanih perzistentnih okvira, programerima aplikacija će biti potrebno mentorstvo u korišćenju okvira, što je često veoma jednostavan zadatak.

Servisi- radi diskusije, servisisu operacija koju nudi računarski entitet i koju mogu pozvati drugi računarski entiteti. U vreme pisanja ovog teksta, najpopularnija arhitektonska strategija su veb servisi, međutim, kao što vidite na sledećoj listi, to je samo jedna od nekoliko uobičajenih strategija koje su vam dostupne. Servisi se obično koriste za inkapsuliranje pristupa zastareloj funkcionalnosti i podacima, a u industriji postoji jasna preferencija da se prave nove aplikacije koje prate arhitekturu zasnovanu na veb servisima kako bi se olakšala ponovna upotreba putem integracije sistema.

Neki od servisa su:

- *Common Object Request Broker Architecture* (CORBA). CORBA je popularizovana ranih 1990-ih kao preferirani pristup za implementaciju distribuiranih objekata na platformama koje nisu Microsoft. Danas se CORBA koristi, najvećim delom, kao tehnologija za omotavanje starih računarskih sredstava (čak se koristi za omotavanje pristupa DCOM aplikacijama).

- *Customer Information Control System* (CICS) Transaction. CICS je bio monitor transakcijskog procesora (TP) koji je popularizovao IBM 1970-ih. Danas je to i dalje rastuća tehnološka platforma za kritične poslovne aplikacije.
- *Distributed Component Object Model* (DCOM).DCOM je popularizovan početkom 1990-ih kao preferirani pristup za implementaciju distribuiranih komponenti u Microsoft okruženjima. DCOM je protokol koji omogućava softverskim komponentama da komuniciraju direktno preko mreže na pouzdan, bezbedan i efikasan način. DCOM je i dalje važan deo Microsoft platforme.
- *Electronic data interchange* (EDI).EDI je standardizovana razmena elektronskih dokumenata između organizacija, na automatizovan način, direktno iz računarske aplikacije u drugu. EDI je popularizovan sredinom 1980-ih od strane velikih proizvođača i njihovih dobavljača, a i danas čini osnovu mnogih aplikacija koje su kritične za misiju. Čini se da se EDI polako zamenjuje veb servisima.
- *Stored procedures*. Uskladištene procedure implementiraju funkcionalnost, obično kolekciju SQL iskaza, unutar baze podataka. Sačuvane procedure su popularizovane sredinom 1980-ih i danas su važeća tehnologija implementacije.
- *Web services*. Veb servisi su samostalne poslovne funkcije, napisane prema striktnim specifikacijama, koje rade preko Interneta koristeći XML. Uobičajene platforme za veb servise su *Microsoft .NET* i *Sun ONE* tehnologija.

Servisi će verovatno zahtevati da agilni administratiori baze podatka razumeju koje su servisi dostupni. Ove informacije bi trebalo da budu dostupne od administratora preduzeća i/ili arhitekata preduzeća, a još bolje, trebalo bi da budu slobodno dostupne kroz spremište za ponovnu upotrebu. Za servise koji obuhvataju pristup podacima, verovatno će se očekivati da agilni DBA radi sa programerima aplikacija na pisanju koda za pozivanje togservisa.

3. ZAKLJUČAK

U ovom radu prikazano je kako se i pomoću kojih koraka izrađuje baza podataka pomoću TDD pristupa. Postavljene su teorijske osonove koje je potrebno proučiti pre same izrade baze podataka. Objašnjena je i razlika između pristupa koji su slični TDD pristupu i izvršena je komparacija između datih pristupa. Cilj rada je bio da se TDD približi i da se kroz dobijene rezultate pokažu prednosti korišćenja ovog pristupa. Izraženi su i nedostaci sa kojima se susrećemo, ali koji su prihvatljivi u odnosu na prednosti koje dobijamo korišćenjem ovog pristupa. U nastavku istraživanja implementiranu bazu potadaka pomoću TDD pristupa trebamo da analiziramo i ispitamo perfromanse.

4.LITERATURA

[1] Scott W. Ambler, Agile Database Techniques: Effective Strategies for the Agile Software Developer, John Wiley & Sons (2003)
[2] Stéphane Faroult with Pascal L'Hermite, Refactoring SQL Applications, O'Reilly (2008)
[3] Max Guernsey III, Test-Driven Database Development: Unlocking Agility, Addison- Wesley (2013)
[4] Scott W. Ambler, Pramod J. Sadalage, Refacotring Databases: Evolutionary Database Desing, Addison Wesley Professional (2006)
[5] Slobodan Mirković, Saša D. Lazarević: Test driven development patterns and test code refactoring, Info M : časopis za informacionu tehnologiju i multimedijalne sisteme, ISSN 1451-4397, 2013, god. 12, sv. 45, str. 46-54.
[6] Sanja Kostić, Saša D. Lazarević: Comparative analysis of the programming code quality developed using test driven development and conventional method, Info M : časopis za informacionu tehnologiju i multimedijalne sisteme, ISSN 1451-4397, 2018, god. 17, br. 67, str. 2229.

Marko Alavanja, student master studija, Fakultet organizacionih nauka.
Kontakt: markoalavanja12@gmail.com
Oblasti interesovanja: softversko inženjerstvo, programiranje, Java, SQL

Prof. dr. Saša Lazarevic, redovni profesor, Fakultet organizacionih nauka.
Kontakt: lazarevic.sasa@fon.bg.ac.rs
Oblasti interesovanja: softversko inženjerstvo, programiranje, .NET, SQL

CIP – Каталогизација у публикацији Народна библиотека Србије, Београд 659.25:004
INFO M : časopis za informacione tehnologije i multimedijalne sisteme = journal of Information technology and multimedia systems / glavni i odgovorni urednik Miroslav Minović. - [Štampano izd.]. - God. 1, br. 1 (2002)- . - Beograd : Fakultet organizacionih nauka, 2002- (Smederevo : Newpress). - 30 cm
Dva puta godišnje. - Je nastavak: Info Science = ISSN 1450-6254. - Drugo izdanje na drugom medijumu: Info M (Online) = ISSN 2683-3646
ISSN 1451-4397 = Info M (Štampano izd.)
COBISS.SR-ID 105690636