University of Zagreb

FILOZOFSKI FAKULTET

Siniša Pavlović

SUSTAV ZA RAZVOJ APLIKACIJA USMJERENIH NA BAZE PODATAKA

DOKTORSKI RAD

Zagreb, 2012.

University of Zagreb

FACULTY OF HUMANITIES AND SOCIAL SCIENCES

Siniša Pavlović

SYSTEM FOR DEVELOPING DATABASE ORIENTED APPLICATIONS

DOCTORAL THESIS

Zagreb, 2012

Sveučilište u Zagrebu

FILOZOFSKI FAKULTET

Siniša Pavlović

SUSTAV ZA RAZVOJ APLIKACIJA USMJERENIH NA BAZE PODATAKA

DOKTORSKI RAD

Mentor:

Vladimir Mateljan

Zagreb, 2012.

University of Zagreb

FACULTY OF HUMANITIES AND SOCIAL SCIENCES

Siniša Pavlović

SYSTEM FOR DEVELOPING DATABASE ORIENTED APPLICATIONS

DOCTORAL THESIS

Supervisor:

Vladimir Mateljan

Zagreb, 2012

SAŽETAK

Razvoj informacijskih sustava za rad s bazama podataka ima brojne specifičnosti, kako u dijelu razvojnog procesa, tako i u njihovu korištenju. Cilj ovog rada jest utvrditi specifičnosti aplikacija usmjerenih na baze podataka, te predložiti efikasan sustav za razvoj takvih aplikacija.

Predloženi sustav za razvoj aplikacija usmjerenih na baze podataka sastoji se od apstraktnog modela objekata koji zajedno čine jedinstvenu cjelinu obavljajući funkcije komunikacije s bazom podataka, lokalne pohrane podataka na klijentskoj strani, kontrole ispravnosti unesenih podataka, upravljanja podsustavom događaja i upravljanja transakcijskim mehanizmom i zaključavanjem slogova u bazi podataka. Osim apstraktnog modela, nužan element sustava za razvoj aplikacija usmjerenih na baze podataka čini korisničko sučelje, pa je stoga u ovom radu detaljno objašnjen skup elemenata korisničkog sučelja prilagođen zahtjevima unosa podataka, te njihov odnos sa događajnim podsustavom apstraktnog modela. U radu je opisan sustav repozitorija u kojem su zapisani elementi apstraktnog modela koji zajedno čine izvorni kod aplikacije i koji omogućuje kvalitetniji razvoj aplikacija usmjerenih na baze podataka, naročito onda kada se taj razvoj odvija s većim brojem članova u razvojnom timu. Sustav repozitorija može zapisivati izvorni kod u bazu podataka i na taj način iskoristiti prednosti transakcijskog mehanizma, pretraživanja i obrade velike količine strukturiranih podataka s ciljem boljeg praćenja dinamike razvoja i odnosa među elementima aplikacije.

Sustav za razvoj aplikacija usmjerenih na baze podataka opisan u ovom radu nije ovisan o nekoj konkretnoj tehnologiji i može biti implementiran na bilo kojoj hardverskoj i softverskoj platformi, pod uvjetom da ona omogućuje rad s relacijskom bazom podataka i odgovarajućim korisničkim sučeljem, pa ovaj rad može poslužiti kao osnova za izradu konkretnog sustava koji će biti korišten u praksi.

Ključne riječi

Aplikacija, baza podataka, razvoj aplikacija, korisnik, sustav, razvojna okolina, apstraktni model.

ABSTRACT

Development of database information systems has many special characteristics, in development process as well as in use. In this paper those special characteristics of database applications is determined and a system for the efficient develoment of such applications is proposed.

Proposed system for database application development consists of an abstract model of objects that together form a single unit performing the function of communication with the database, local data storage on the client side, controling the data validaton, the event management subsystem and transaction management mechanism with record locking in database. In addition to the abstract model, an important element of the system for database application development is the user interface and therefore in this paper a set of user interface elements adapted to the requirements of data entry, and their relationship with the event subsystem of abstract model is explained in detail. This paper describes a repository system in which elements of abstract model are stored which makes the source code of application and improves the quality of database application development, especially with a larger number of members in the development team. The system repository is able to write source code to the database and thus take advantage of the transaction mechanism, searching and processing large amounts of structured data in order to do a better control of development process dynamics and the relationship between the elements of the application.

System for database application development described in this paper is not dependent on any particular technology and can be implemented on any hardware and software platform, provided that it can work with relational database and appropriate user interface, so this paper can serve as a guide to produce a specific system implementation that will be used in practice.

Keywords

Application, database, application development, user, system, development environment, abstract model.

ZAHVALA

Prije svih želim zahvaliti svom mentoru prof. dr. sc. Vladimiru Mateljanu za podršku koju mi je pružio, strpljivo tolerirao sve moje greške, te me uvijek iznova usmjeravao na pravi put.

Isto tako, želim zahvaliti svojoj obitelji, supruzi Kristini i sinovima Tomislavu i Krešimiru, jer oni su podnijeli najveće žrtve i odricanja prilikom nastajanja ovog doktorskog rada i bez čije nesebične podrške ovaj rad ne bi bio moguć.

Također, želim zahvaliti i svojim roditeljima Ljubici i Josipu, te baki Ružici koji su me podržavali u mojim akademskim naporima i to još od najranijih dana i čvrsto vjerovali u mene.

Moram zahvaliti i svojim brojnim kolegama i prijateljima s kojima sam u praksi radio na razvoju aplikacija usmjerenih na bazu podataka i u čijem je društvu kroz brojne, ponekad i vrlo burne, rasprave ova ideja polako ali sigurno dobijala svoj konačni oblik.

Siniša Pavlović

SADRŽAJ

1. UVOD

1.1. OBRAZLOŽENJE TEME

Suvremeni informacijski sustavi služe rješavanju složenih zadaća i imaju veliki ekonomski značaj, a troškovi njihova razvoja i održavanja postaju sve kritičnija veličina (Birrell & Ould, 1988, 24 - 32; Jalote, 2005, 2 - 25), te nije svejedno koliko je vremena, financijskih sredstava i ljudskih resursa potrebno za njihovu izradu. No, bez obzira na njihovu važnost, mnogi projekti razvoja kasne, probijaju troškovne limite ili nisu odgovarajuće kvalitete (Sommerville, 1995, 1 - 9; Kan, 2002, 1 - 5; Beck, 1999, 3 - 6). Uzroke za to treba tražiti u neprestanom rastu računalnih sustava, koji rastu kako veličinom, tako i kompleksnošću (Brooks, 1995, 43,80; Brooks, 1986, 1069 - 1076).

Da bi se postiglo smanjenje složenosti procesa razvoja, potrebno je razmotriti od čega se ta složenost sastoji. Frederick Brooks smatra da se složenost razvoja aplikacija sastoji od dvije komponente:

(a) Esencijalna složenost (eng. essencial complexity) - ova složenost proizlazi iz složenosti problemskog područja koje aplikacija rješava.

(b) Slučajna složenost (eng. accidental complexity) - ova složenost proizlazi iz tehnologije i metodologije razvoja.

Brooks smatra da se esencijalna složenost ne može smanjiti ispod određenog nivoa jer je problem koji se rješava jednako složen bez obzira na reprezentaciju i metode rješenja, pa tako aplikacija koja mora izvršavati 30-tak točno određenih funkcija, ne može taj broj smanjiti. Esencijalna složenost proizlazi iz četiri osobine informacijskih sustava:

(1) Kompleksnost - informacijski sustavi imaju daleko veći broj mogućih stanja nego bilo koji drugi tehnološki sustav koji je izradio čovjek.

(2) Uskladivost - informacijski sustavi, kao i njihovi podsustavi, međusobno komuniciraju, te komuniciraju s vanjskim svijetom. Ta komunikacija je otežana

činjenicom da su je osmislili različiti ljudi, sa različitim interesima, ciljevima, tehničkim i drugim znanjima i slično.

(3) Promjenjivost - informacijski sustavi su konstantno i intenzivno izloženi promjenama, koje mogu dovesti i do narušavanja funkcionalnosti i uvođenju nestabilnosti i grešaka u sustav, čime se znatno usložnjava razvoj.

(4) Nevizualnost - informacijske sustave je teško prikazati vizualno, jer sadrže apstraktne dijelove. Iako vizualizacija može pomoći razumijevanju funkcioniranja samog sustava, postoje mnogi aspekti koji se ne mogu prikazati vizualizacijom.

Po Brooksu, značajna ušteda utrošenih resursa može se pronaći u dijelu slučajne složenosti, odnosno uporabe tehnologija i metodologija razvoja. On smatra da tu ne može postojati jedinstveno rješenje za sve vrste problema (tzv. srebrni metak) iako razmatra kandidate za to (jezici visokog nivoa, objektno-orjentirano programiranje, umjetna inteligencija, ekspertni sustavi, automatsko programiranje, grafičko programiranje itd). Iz toga proizlazi da specijalizacija, odnosno prilagodba, tehnologije i metodologija za pojedine vrste problema i informacijskih sustava može dati bolje rezultate od općih na dijelu slučajne složenosti (Brooks, 1986, 1069 - 1076).

Još iz ranih iskustava razvoja računalnih aplikacija postalo je jasno da postojeće metodologije razvoja računalnih aplikacija nisu bile dovoljno dobre i da su potrebna njihova poboljšanja te je stoga došlo do razvitka softverskog inženjerstva, kao posebne znanstvene discipline, koja se bavi unaprjeđenjem načina razvoja informacijskih sustava (Pfleeger & Atlee, 2009, 2 - 14; Selby, 2007, 5 - 7; Sharma, 2004, 1 - 8; Wohlin & Runeson & Höst, 1999, 1 - 4).

Jedan od značajnijih istraživača s područja softverskog inženjerstva, Somerville (Sommerville, 1995, 1 - 9) uvodi pojam softverskog procesa kao skupa aktivnosti i njihovih pripadajućih rezultata koji zajedno dovode do gotove računalne aplikacije. Te aktivnosti su:

(1) Specifikacija - određivanje funkcionalnosti koju gotova računalna aplikacija treba imati, zajedno sa ograničenjima koja se moraju poštivati.

(2) Razvoj - proces razvoja gotove računalne aplikacije, a koji mora biti u skladu sa specifikacijom.

(3) Validacija - proces vrednovanja aplikacije kako bi se moglo utvrditi da li radi bez pogrešaka i da li je izrađen u skladu sa specifikacijom.

(4) Evolucija - upravljanje promjenama u računalnoj aplikaciji, kako zbog izmjenjenih korisničkih zahtjeva, tako i zbog promjene u okolini.

Metodologije razvoja programske podrške odnose se na sve gore spomenute točke (1-4). Tema opisana u ovom radu odnosi se na aktivnost razvoja aplikacije i to specifično na alate kojima se taj razvoj postiže, odnosno na sustave za razvoj aplikacija kao integrirane grupe softverskih produkata koje pomažu efikasan razvoj računalnih informacijskih sustava.

Strojni jezik mikroprocesora postiže izuzetnu efikasnost izvođenja programskog koda ali je s druge strane zahtjevan za programiranje, te postoji realna potreba za lakšim i efikasnijim načinima programiranja koji dolaze u obliku programskih jezika (Scott, 2009, 5 - 39). Programski jezici su općenito podijeljeni na više i niže programske jezike, pri čemu su niži programski jezici po svojoj strukturi bliži načinu funkcioniranja računala, dok su viši programski jezici razumljiviji ljudima pa omogućuju veću efikasnost procesa razvoja računalnih aplikacija (Kernighan, Ritchie, 1988, 6-10).

Programski jezici imaju svoju povijest i određene razvojne faze, pa se programski jezici dijele u pet generacija (Committee on the Past and Present Contexts for the Use of Ada in the Department of Defense, 1997, 80 - 81; Stair & Reynolds, 2009, 172; Azarmsa, 1991, 21 - 24; Baum, 1992, 48 - 49; Parsons & Oja, 2008, 675 - 678).

(1) Programski jezici prve generacije odnose se na strojni jezik gdje se računalni program sastoji od niza instrukcija mikroprocesoru, a strukture podataka svode se na registre mikroprocesora i direktnom pristupu memoriji. Osim niske razine produktivnosti prilikom programiranja, javlja se problem portabilnosti aplikacija, jer bi se aplikacije morale prerađivati ukoliko bi se prenosile na računalo sa drugačijim mikroprocesorom.

(2) Programski jezici druge generacije uvode simboličke oznake, odnosno nazive, za odgovarajuće instrukcije mikroprocesora, kao i imena za pojedine registre, simboličke oznake mjesta za instrukcije skoka i tome slično, povećavajući na taj način čitljivost programskog koda.

(3) Programski jezici treće generacije daljnje su unaprjeđenje druge generacije, uvodeći programske strukture (iteracije, selekcije), strukture podataka te strukturalno i kasnije objektno-orjentirano programiranje. Predstavnici jezika treće generacije su Fortran, Cobol, C, C++, Pascal, Basic, Ada , Java, Python i slično. Bitna karakteristika jezika treće generacije je njihova općenitost i mogućnost korištenja u bilo kojem problemskom području. Programski jezici treće generacije često su temelj na kojem se razvijaju jezici četvrte i pete generacije.

(4) Programski jezici četvrte generacije predstavljaju daljnje unaprjeđenje programskih jezika, a dizajnirani su za korištenje isključivo za specifičnu namjenu, odnosno s idejom izrade usko-specijaliziranog jezika. Takvom specijalizacijom postiže se veća produktivnost unutar specificiranog problemskog područja, ali i otežan ili nemoguć razvoj aplikacija izvan takvog specijaliziranog područja. Primjeri za takve jezike su SQL (upiti na relacijsku bazu podataka), SPSS (jezik za statističku obradu podataka), ColdFusion (za izradu internetskih aplikacija) i slično.

(5) Programski jezici pete generacije temeljeni su na ideji programiranja bez pružanja detaljnog opisa algoritma koji rješava zadani problem, već se opisom problema, odnosno, pravilima i ograničenjima koji vrijede u aplikaciji, stvaraju preduvjeti za automatizirano pronalaženje rješenja i njegovo izvršenje. Programski jezici pete generacije koriste dostignuća umjetne inteligenciju (naročito mehanizme zaključivanja) i tehnike vizualizacije za postizanje tog cilja. Tipičan primjer jezika pete generacije je Prolog.

U razvoju programske podrške koriste se integrirana razvojna okruženja (eng. integrated development environmend - IDE) koja nude dodatnu pomoć u organizaciji razvoja aplikacija, te olakšavaju rad s izvornim kodom velike složenosti. Tako integrirana razvojna okruženja predstavljaju slijedeću fazu u formiranju razvojnih okruženja (Liang, 2010, 10; Darwin, 2004, 4 - 5).

Bitne karakteristike integriranih razvojnih okruženja (Qin & Xing & Zheng, 2008, 191 - 198; Kann, 2004, 373; Bidgoli, 2003, 610 - 612; Andrews, 2004, 42; Hammond & Robinson, 2000, 46 - 48; Hubbard, 2000, 9; Liberty & MacDonald, 2008, 16 - 17; Oualline, 2003, 11) su:

(1) Integriranost i međusobna uska suradnja uređivača izvornog koda, prevoditelja i ostalih alata. S obzirom na to da je prilikom razvoja programskih rješenja nužno paralelno koristiti više alata sa različitim funkcijama, integrirana razvojna okružja ih objedinjuju u jednu cjelinu i omogućuju njihovu međusobnu komunikaciju olakšavajući na taj način razvojni proces.

(2) Organiziranje elemenata aplikacije u projekt, a koji može biti organiziran hijerarhijski ili u repozitorije. Veća programska rješenja odlikuju se velikom složenošću i često se sastoje od velikog broja elemenata koji se nalaze u određenom međusobnom odnosu. Kvalitetna organizacija elemenata od kojih se aplikacija sastoji u smislenu cjelinu može značajno ubrzati razvoj aplikacije.

(3) Alat za dizajniranje korisničkog sučelja. Umjesto pozicioniranja elemenata korisničkog sučelja putem izvornog koda, integrirana razvojna okruženja omogućuju vizualno dizajniranje korisničkog sučelja koristeći računalnu grafiku.

(4) Sustav za kontrolu verzija na poslužitelju (eng. version control system). Pogreške u smjeru razvoja aplikacije čine nužnom mogućnost povratka na neku od prethodnih verzija izvornog koda. Sustav za kontrolu verzija čuva sve povijesne verzije izvornog koda aplikacije na poslužitelju, te je moguće pristupiti bilo kojoj od verzija koda.

(5) Mogućnost istovremenog razvoja izvornog koda aplikacije od strane više članova razvojnog tima. Istovremenost razvoja izvornog koda računalne aplikacije u nužna je uvjetima kada složenost aplikacije zahtijeva veći broj članova u razvojnom timu. Mogućnost istovremenog razvoja postiže se rezerviranjem i oslobađanjem pojedinih elemenata aplikacije ili spajanjem različitih verzija jednog elementa u novu cjelinu. Ovo je u uskoj vezi sa sustavom za kontrolu verzija.

(6) Integrirana podrška za testiranje. Vrednovanje programske podrške važan je element razvoja koji osigurava točnost i kvalitetu izvršavanja aplikacije, minimizirajući mogućnost greške, pa mnoga integrirana razvojna okruženja teže povezati pojedine elemente aplikacije i njihove pripadajuće testove.

(7) Integrirana podrška za dokumentiranje. Dokumentacija, kako razvojna tako i korisnička, nužna je za razumijevanje načina korisničkog upravljanja aplikacijom, kao i internih mehanizama same aplikacije, te je mogućnost integriranosti dokumentacije sa pripadajućim elementima aplikacije od velike pomoći razvojnim timovima, naročito u situacijama četsih promjena članova razvojnog tima.

Poduzeća, ovisno o njihovoj veličini, državne institucije i ostale organizacije imaju potrebu za unosom i pohranom velike količine podataka, od strane brojnih korisnika, istodobno omogućujući pretraživanje tako pohranjenih podataka, kao i sustav izvješćivanja i analize podataka. Najčešće rješenje za postizanje tih ciljeva jest korištenje relacijskih baza podataka (Allen & Creary & Chatwin, 2003, 3 - 6; O'Donnell, 2008, 8 - 10; Tymann & Reynolds, 2008, 140 – 141).

Baza podataka jest sustav čija je osnovna zadaća omogućiti pohranu podataka i te obradu i čitanje tih podataka na zahtjev. Relacijska baza podataka jest baza podataka koja se temelji na relacijskom modelu podataka (Date, 1995, 3 - 24). Relacijski model podataka osmislio je E.F. Codd sedamdesetih godina prošlog stoljeća i predstavlja zaokruženi teoretski aparat, a matematičke osnove ovog modela omogućuju elegantnu i preciznu definiciju pojmova i izvođenje zaključaka (Tkalac, 1993, 3 - 8). Osnovni elementi relacijskog modela su:

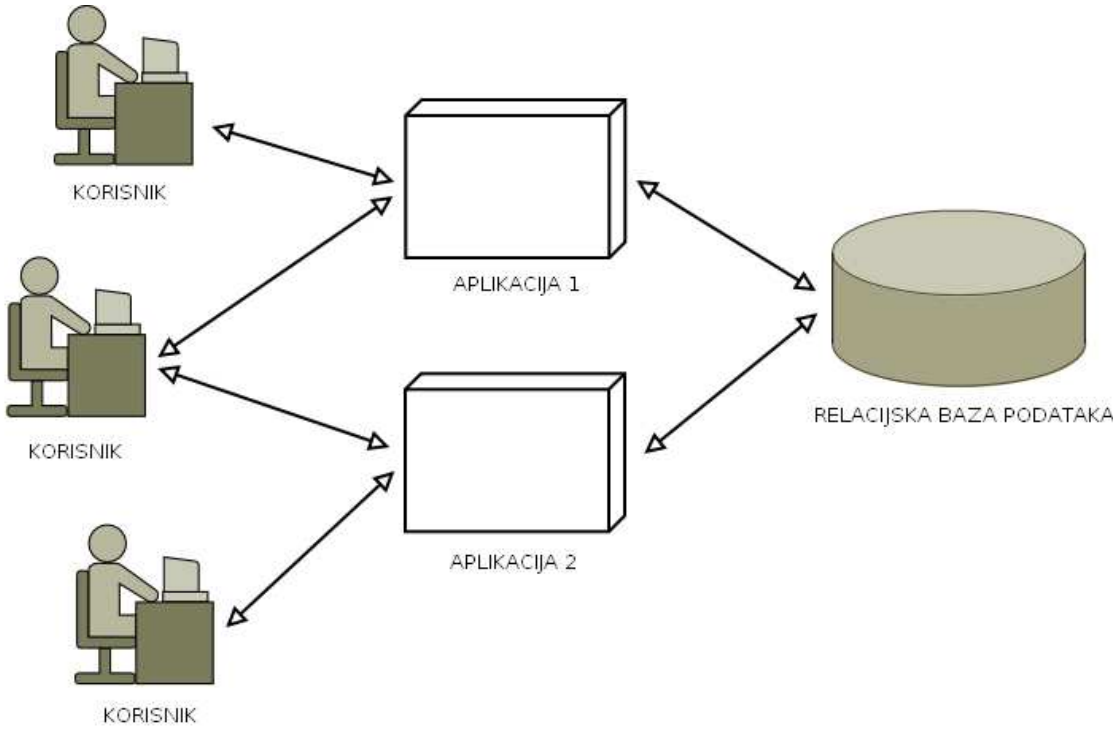
(1) Skup objekata baze podataka

(2) Skup operacija nad objektima baze podataka

(3) Skup pravila integriteta baze podataka

Korištenje relacijskih baza podataka omogućuje višekorisnički rad, transakcijski pristup izmjene podataka i očuvanja integriteta podataka (Date, 1995, 3 - 24; Tkalac, 1993, 3 - 8), te korištenje strukturiranog upitnog jezika (eng. Structured Query Language - SQL) koji je osmišljen za rad s relacijskom bazom u smislu unosa, izmjene i brisanja podataka, te postavljanja upita (Kedar, 2008, 2-1 - 2-2; ISRD Group, 2006, 101 - 107; Price, 2007, 552 - 557; Morton & deHaan & Gorman & Jargensen & Fink, 2009, 25 - 30).

Osim relacijske baze podataka, za ostvarenje ciljeva rada s velikim količinama podataka potrebne su i aplikacije koje su prilagođene potrebama krajnjih korisnika i rješavanju njihovih specifičnih problema, a koje za svoje ispravno funkcioniranje moraju komunicirati s bazom podataka, odnosno, moraju biti usmjerene na bazu podataka. Takve aplikacije posreduju između korisnika i baze podataka (Connolly & Begg, 2004, 17 - 22; Schmidt & Stogny, 1991, 185 - 187; Goodson & Steward, 2009, 10 - 18) (slika 1.1.).



Slika 1.1 - Aplikacije kao posrednici između korisnika i relacijske baze podataka

Ovakvo je posredovanje neophodno iz razloga što korisnik ne mora a često i ne može biti upoznat sa modelom podataka u relacijskoj bazi, pa je aplikacija prilagođena korisničkom pogledu na problematiku (Mace, 1986, 39 - 42; Bench-Capon & Soda & Tjoa, 1999, 521; O'Neil & O'Neil, 2000, 6 - 7). Isto tako, upravljanje samim modelom podataka može biti toliko kompleksno da je neophodno imati aplikaciju kao posrednika čak i u slučajevima kad su korisnici dobro upoznati s modelom podataka u bazi. Ova kompleksnost ogleda se kako u samom modelu podataka, tako i u upravljanju transakcijama, zaključavanju podataka, korisničkim prijavama i pravima, sortiranju i filtriranju podataka, poslovnim pravilima i tome slično.

1.2. SVRHA I CILJEVI ISTRAŽIVANJA

Cilj ovog istraživanja jest definirati sustav za razvoj aplikacija usmjerenih na baze podataka, tj. izgraditi poseban apstraktni model koji uzima u obzir sve specifičnosti razvoja aplikacija za rad s bazama podataka, te predstavlja unaprjeđenje u odnosu na dosadašnje sustave. Apstraktni model sastoji se od opisa podataka i programske logike uklopljene u odgovarajući sustav događaja s jedne i opisa korisničkog sučelja s druge strane, a dolazi u obliku sustava repozitorija kako bi se olakšalo pisanje velikih aplikacija u uvjetima razvojnih timova.

1.3. METOLOGIJA RADA

U radu su definirane specifičnosti aplikacija usmjerenih na baze podataka, kao i specifičnosti njihovog razvoja, a te specifičnosti imaju svoje uporište u apstraktnom modelu i ta je njihova povezanost prikazana. Apstraktni model je detaljno specificiran, te je razvijen softver, odnosno programski primjer, koji ilustrira osnovne značajke apstraktnog modela i njegove prednosti.

1.4. PREGLED LITERATURE

U literaturi koja razmatra problematiku razvoja aplikacija usmjerenih na baze podataka, većina autora bavi se metodologijama razvoja takvih aplikacija pri čemu se zanemaruje sam razvojni sustav kao nevažan, ili se autori dotiču teorije i tehnologije samih relacijskih baza podataka pa ne postoje radovi koji bi se bavili isključivo teorijom unutarnje organizacije sustava za razvoj aplikacija usmjerenih na bazu.

Što se tiče konkretnih sustava za razvoj, bez obzita radi li se o sustavima otvorenog ili zatvorenog koda, postoje brojni radovi koji se tiču načina korištenja, ali se opet ne dotiču unutarnje organizacije, a još manje postaviti kakvu kvalitetnu teorijsku osnovu za daljnje unaprjeđenje takvih sustava.

1.5. ZNANSTVENI DOPRINOS

Znanstveni doprinos ovog rada nalazi se u

a) utvrđivanju specifičnosti aplikacija usmjerenih na bazu podataka

b) izradi apstraktnog modela sustava za razvoj aplikacija usmjerenih na bazu podataka koji je usklađen sa specifičnostima takvog razvoja.

1.6. PRAKTIČNI DOPRINOS

Praktični doprinos ovoga rada odnosi se na definiranje apstraktnog modela kao polazišne osnove za implementaciju konkretnih sustava za razvoj aplikacija usmjerenih na bazu koje se mogu koristiti u realnim projektima. Ovaj rad pruža detaljan uvid kako u pretpostavke koje moraju biti zadovoljene da bi jedan takav sustav mogao postojati, tako i u način na koji treba funkcionirati.

2. APLIKACIJE USMJERENE NA BAZE PODATAKA I SUSTAVI ZA NJIHOV RAZVOJ

U tekstu koji slijedi navedene su osobine aplikacija usmjerenih na baze podataka, te osobine razvoja takvih aplikacija. Navedene su i osobine koje mora imati kvalitetan sustav za razvoj aplikacija usmjerenih na baze podataka, te su analizirani najznačajniji predstavnici postojećih sustava za razvoj aplikacija usmjerenih na baze podataka te utvrđen stupanj njihove prilagođenosti zadanim osobinama kvalitetnog sustava za razvoj takvih aplikacija.

2.1. OKOLNOSTI I OSOBINE APLIKACIJA USMJERENIH NA BAZE PODATAKA I NJIHOVOG RAZVOJA

Aplikacije usmjerene na baze imaju brojne dodirne točke sa aplikacijama koje nisu usmjerene na baze podataka, ali također i brojne osobine koje ih od njih razlikuju, a isto vrijedi i za proces njihovog razvoja. Slijedi detaljnija specifikacija osobina koje ih razlikuju, kako samih aplikacija, tako i procesa njihovog razvoja.

2.1.1. ISTOVREMENI RAD VIŠE KORISNIKA

Aplikacije usmjerene na baze podataka predviđene su za istodobno korištenje od strane velikog broja korisnika (Yeung & Hall, 2007, 294 - 296). Podaci koje jedan korisnik unosi u bazu nisu dostupni samo njemu nego i svim ostalim korisnicima. Pri tom se javlja nekoliko mogućih zahtijeva. Prvi zahtijev jest da su neki podaci ne smiju biti dostupni svim korisnicima. Drugi zahtijev odnosi se na postojanje ograničenja pristupa samo onim funkcijama aplikacije koji odgovaraju korisničkim ovlastima i zaduženjima.

Takvi zahtijevi rješavaju se korisničkim prijavama i pravima, gdje je svakom korisniku pridružena njegova vlastita prijava (korisničko ime i zaporka), te pripadajući skup prava koja mu omogućuju pregled ili ažuriranje pojedinih dijelova baze podataka. Korisnička prava i prijave mogu se definirati na strani baze podataka korištenjem njenih objekata i internih mehanizama prava pristup, ali i na strani aplikacije korištenjem kontrolnih programskih struktura (Tipton & Krause, 2007, 1508 - 1509).

2.1.2. ZAKLJUČAVANJE SLOGOVA I TRANSAKCIJE

Istovremeni rad većeg broja korisnika na istoj bazi podataka ima za posljedicu povremenu pojavu u kojoj dvoje ili više korisnika žele istovremeno mijenjati isti podatak. Relacijske baze podataka posjeduju mehanizam kojim takve situacije mogu biti riješene, a to su zaključavanja i transakcije.

2.1.2.1. ZAKLJUČAVANJE SLOGOVA

Mehanizam zaključavanja slogova osigurava da korisnici nekontrolirano međusobno ne ometaju proces zapisivanja, sa ishodom koji je teško predvidjeti. Mehanizam zaključavanja slogova neodvojiv je od mehanizma transakcija.

2.1.2.2. TRANSAKCIJE

Podaci u bazama podataka mogu biti zapisani u više tablica uz zahtijev da se moraju međusobno uskladiti. Proces zapisivanja takvih podataka može se sastojati od nekoliko koraka koji se ne smiju djelomično izvršiti, već se moraju izvršiti kao cjelina.

Baze podataka posjeduju mehanizam transakcija u kojem se sve operacije zapisivanja odvijaju unutar transakcije. Transakcija se sastoji od početka, operacija zapisivanja, te kraja. Ukoliko se unutar transakcije uspiju zapisati svi podaci koji joj pripadaju, sve promjene podataka izvršene unutar trajanja transakcije postaju trajne, a u slučaju neuspjeha završetka zapisivanja, podaci se vraćaju na stanje prije transakcije. Promjene podataka izvršene unutar trajanja transakcije postaju vidljive ostalim korisnicima tek kada transakcija završi.

Povezanost zaključavanja slogova i transakcija vidljiva je kod brisanje ili ažuriranje slogova u bazi podataka, pri čemu se automatizmom zaključavaju slogovi kako bi se spriječile istovremene izmjene podataka na istim slogovima od strane drugih korisnika. Slogovi se ponovo otključavaju nakon završetka transakcije.

Ovi se mehanizmi obično odvijaju bez znanja korisnika (osim signalizacije da su neki slogovi trenutno nedostupni) i integralni su dio relacijskih baza podataka, ali i infrastrukture aplikacija usmjerenih na bazu podataka koje moraju biti u stanju upravljati njima na ispravan i efikasan način.

2.1.3. KORISNIČKO FILTRIRANJE I SORTIRANJE PODATAKA

Razvojni timovi ne mogu predvidjeti sve okolnosti pod kojima aplikacija može biti korištena, a s druge strane, nužno je korisnicima omogućiti obavljanje njihova posla. Kako bi se korisnicima olakšalo izdvajanje potrebnih podataka iz baze podataka, te olakšalo njihovo sortiranje, nužno je imati standardiziran i konzistentan način kojim oni uvijek to mogu sami učiniti, bez potrebe da se zbog toga mijenja sama aplikacija. Korisnici moraju imati mogućnost samostalnog određivanja kriterija filtriranja i sortiranja podataka u bilo kojem dijelu aplikacije.

2.1.4. KONZISTENTNOST PODATAKA

U bazama podataka zapisuju se podaci koji se mogu nalaziti u određenoj međuzavisnosti i mogu biti podložni određenim ograničenjima.

2.1.4.1. OGRANIČENJA NA BAZI PODATAKA

Ograničenja nad podacima mogu biti definirana na strani baze podataka čime se osigurava da pogrešan podatak ne može ni na koji način biti upisan u bazu podataka, te se prilikom pokušaja zapisivanja podatka koji nije usklađen s ograničenjima javlja greška.

2.1.4.2. PROVJERE NA STRANI APLIKACIJE

Provjerama ugrađenim u samu aplikaciju koje se izvršavaju prije pokušaja zapisivanja u bazu podataka, osigurava se proces unosa podataka koji na učinkovit i transparentan način informira korisnika o pogreškama pri unosu podataka.

Sprječavanje unosa pogrešnih podataka moguće je ostvariti zabranom izlaska iz polja, redka ili forme na kojoj je nastao pogrešan unos.

2.1.5. AUTOMATSKO AŽURIRANJE STANJA U BAZI PODATAKA

Uporabom aplikacije od strane korisnika mijenja se podatkovno stanje, te baza podataka mora biti pravovremeno ažurirana kako bi njeno daljnje korištennje bilo ispravno. Provjere stanja mogu biti izvršavane i na strani baze podataka, te baza podataka može odbiti neku, s aspekta aplikacije, sasvim ispravnu promjenu.

Pravovremeno ažuriranje - može izvršiti na izričito korisnički zahtjev, ali to nije u potpunosti pouzdano rješenje jer ovisi o ljudskom faktoru koji može biti ne pouzdan. Bolje rješenje jest automatizirana validacija podataka i ažuriranje stanja u bazi podataka bez izričite korisničke intervencije.

2.1.6. ISTOVREMENI RAD ČLANOVA TIMA

Aplikacije usmjerene na bazu podataka su složene da bi bile povjerene razvoju samo jedne osobe, te stoga aplikacije uglavnom izrađuju razvojni timovi. Moraju se uskladiti aktivnosti članova tima, te im se mora omogućiti istovremeni rad u skladu s njihovim ulogama, pravima i zadaćama.

Uloge članova tima mogu biti određene na više razina. Prva razina odnosi se na razvoj aplikacije podijeljen na više logičkih podcjelina, gdje je razvoj svake od njih povjeren nekom članu ili grupi članova tima. Druga razina odnosi se na podjelu po vrstama aktivnosti, to jest, podijeli zadaća na modeliranje podataka, razvoj programske logike, razvoj korisničkog sučelja i izvještajnog podsustava.

Istovremeni timski razvoj podrazumijeva situaciju u kojoj više članova tima koji sudjeluju u razvoju aplikacija trebaju istodobno mijenjati iste dijelove aplikacije, što ne predstavlja problem ako svi članovi tima imaju mogućnost komunikacije, ali nije rijetkost da su članovi tima nemaju tu mogućnost i da im je otežana direktna komunikacija. Pažljivo određivanje protokola predviđenih za takve situacije važan je aspekt timskog razvoja aplikacije, a razvojni alat koji je u stanju efikasno podržati takav način razvoja znatno olakšava razvoj aplikacije.

2.1.7. ŽIVOTNI CIKLUS APLIKACIJE

Aplikacije usmjerene na bazu podataka mogu prolaziti kroz dug životni ciklus. Tijekom svog životnog ciklusa aplikacije se konstantno izmjenjuju i nadograđuju, kako zbog izmjena uvjeta u kojima djeluju organizacije koje ih koriste, tako i zbog izmjenjenih ciljeva i strategija samih organizacija. Izmjene vanjskih uvjeta mogu biti promjene na tržištu, ali i izmjena zakonskih i institucionalnih okvira. Proces izmjene aplikacije u skladu sa novim zahtjevima kao i korekcija grešaka u funkcioniranju aplikacija naziva se održavanje aplikacije.

Tijekom životnog ciklusa moguće je da se članovi razvojnih timova koji su započeli projekt u potpunosti zamijene novim članovima, te je vrlo bitno da stari članovi tima uspješno prenesu neophodna znanja novim članovima, a isto tako bitno je da postoji kvalitetna projektna dokumentacija.

Ako kvalitetan prijenos znanja i kvalitetna projektna dokumentacija ne postoje, može se dogoditi da novi članovi tima tijekom održavanja proizvedu ozbiljnje greške koje mogu prouzročiti veliku štetu, ili pak da se novi članovi i njihovi nalogodavci ne usude upuštati u veće izmjene aplikacije zbog straha od grešaka.

2.1.8. PROJEKTNO IZVJEŠĆIVANJE

Projektni izvještaji poput popisa korištenih tablica, izvještaja o formama za unos podataka, izvještaja o obradama, sastavni su dio projektne dokumentacije. U uvjetima izmjena na aplikaciji, potrebno je ažurirati i dokumentaciju, tako da ona u svakom trenutku bude usklađena sa stvarnom situacijom u aplikaciji.

Ukoliko se pretraživanja mjesta na kojima je potreno izvršiti promjenu izvornog koda, te istovrsna promjena izvornog koda na više mjesta u aplikaciji, vrše ručno, ona mogu oduzeti značajno vrijeme i ubrzano trošiti resurse razvojnog tima. Bolje je ukoliko se ove aktivnosti u cjelosti ili barem u većem dijelu, mogu izvesti automatizmom, kroz infrastrukturu razvojnog sustava.

2.1.9. UJEDNAČENA INFRASTRUKTURA

Aplikacije koje su usmjerene na baze podataka, koje imaju podsustave čijim se aktivnostima različito upravlja i koji prezentiraju informacije na različit način, imaju za posljedicu otežano korištenje. Ovo može biti posljedica načina razvoja aplikacije gdje razvojni alat nema standardizirane pristupe razvoju korisničkog sučelja, te su ga članovi razvojnog tima prisiljeni razvijati od početka. Za takve aplikacije ne postoji jedinstvena paradigma načina razvoja korisničkog sučelja, pa pojedini članovi ili grupe mogu razvijati vlastite načine uporabe umjesto jednog standardiziranog. Takve aplikacije izgledaju i koriste se vrlo različito u svojim pojedinim dijelovima, a korisnici moraju učiti više različitih načina za obavljanje istih ili sličnih aktivnosti, što rezultira nezadovoljstvom korisnika, ali i otežanim održavanjem aplikacije.

Kako bi se takvi problemi izbjegli, nužno je standardizirati način upravljanja aktivnostima, ali i standardizirati izgled pojedinih formi jedinstvenom infrastrukturom unutar cijele aplikacije.

2.1.10. KORISNIČKA SUČELJA RAZLIČITIH RAČUNALNIH PLATFORMI

Računalne platforme dolaze u brojnim oblicima, a korisnici i odjeli nekog poduzeća mogu biti međusobno dislocirani i služiti se raznim tehnologijama. Stoga postoji potreba za omogućavanjem pristupa jednoj te istoj aplikaciji na više različitih tehnologija.

Da bi jednom napisana aplikacija mogla biti korištena na većem broju platformi potrebno je razdvojiti programsku logiku aplikacije od korisničkog sučelja. Sve tehnologije ne posjeduju podjednake mogućnosti, te stoga funkcionalnost aplikacije treba uskladiti s mogućnostima ciljnih tehnologija.

2.1.11. ISTODOBNI PRISTUP VEĆEM BROJU RAZLIČITIH BAZA PODATAKA

Organizacije mogu koristiti više različitih podsustava koji su nastajali tijekom vremena, kreirani od različitih timova na različitim, međusobno nekompatibilnim tehnologijama, a podatke zapisuju u odvojenim bazama podataka. Organizacije ponekad nemaju izbora i moraju zadržati stare sustave, kako zbog nedostatka resursa, tako i zbog opasnosti da novi sustavi ne zamijene stare na odgovarajući način. U takvim uvjetima može se pojaviti potreba za aplikacijom koja je u stanju istovremeno pristupiti nekolicini takvih sustava i povezati njihove podatke u jednu cjelinu.

S druge strane, postoji mogućnost da razvojni timovi ne razvijaju aplikacije za točno određenog naručitelja, već razvijaju aplikacije koje su namijenjene prodaji na tržištu, te postoji želja za podrškom što je moguće širem spektru baza podataka različitih proizvođača, jer se time povećava moguće tržište.

Zato mora postojati fleksibilnost u povezivanju s bazama podataka i to na način da postoji mogućnost istodobnog pristupa na više različitih baza podataka.

2.1.12. OTVORENOST SUSTAVA ZA RAZVOJ APLIKACIJA USMJERENIH NA BAZE PODATAKA

Razvojni timovi ponekad se susreću sa problemom zatvorenosti sustava za razvoj aplikacija, pri čemu se zatvorenost sustava odnosi na nemogućnost nadogradnje osim pod uvjetima

propisanim od strane proizvođača, te nemogućnost migriranja na drugi razvojni sustav. U razvoj informacijskih sustava mogu biti uloženi znatni resursi što proces migracije čini preskupim jer bi se cijeli informacijski sustav morao raditi od početka. S druge strane, može se dogoditi da proizvođač razvojnog sustava proglasi sustav zastarjelim, ponudi novi razvojni sustav te uskraćivanjem podrške starom sustavu prisili korisnika na prelazak na novi sustav. Zatvoreni razvojni sustavi zapisuju izvorni kod aplikacija na šifrirane načine u posebnim formatima podataka čija struktura nije dokumentirana i poznata je samo proizvođaču sustava. Takvi razvojni sustavi sprječavaju slobodu odlučivanja o prelasku na novi sustav ili ostanak na starom.

Otvorenošću sustava za razvoj aplikacija usmjerenih na baze podataka moguće je izbjeći takve probleme zbog jednostavne činjenice da ne postoji monopol na izmjene razvojnog sustava, kao i zbog činjenice da se u otvorenim sustavima koriste otvoreni, standardizirani i dobro dokumentirani formati zapisa podataka, u ovom slučaju izvornog koda aplikacije. Otvorenost razvojnog sustava ujedno omogućuje i njegovu veću stabilnost zbog lakšeg uvida korisničke zajednice u njegove interne mehanizme, te povratnu informaciju u vidu korisničkih izmjena izvornog koda (eng. patches) razvojnog sustava (Raymond, 2001, 19-40).

2.2. OSOBINE KOJE MORA IMATI SUSTAV ZA RAZVOJ APLIKACIJA USMJERENIH NA BAZU PODATAKA

Osobine i specifičnosti koje imaju aplikacije usmjerene na baze podataka, kao i specifičnosti njihovog razvoja najbolje mogu biti podržane specijaliziranim razvojnim sustavima čije osobine odgovaraju zahtjevima razvoja takvih aplikacija. Slijedi popis osobina koje takvi razvojni sustavi moraju imati:

2.2.1. DINAMIČKO KREIRANJE KORISNIČKOG SUČELJA I LOGIKE APLIKACIJE

Postoje sustavi za razvoj aplikacija koji omogućuju korištenje vizualnih alata za dizajniranje korisničkog sučelja gdje se raspored i veličina elemenata korisničkog sučelja određuje ručno, te se određuje programska logika koja se povezuje s određenim događajima na formi. Ovakav način dizajniranja korisničkog sučelja je statičan, što ima svojih prednosti ali i nedostataka.

Nedostatak ovakvog pristupa jest da je unaprijed potrebno odrediti sve elemente i svu programsku logiku koja će biti povezana s formom. Ukoliko aplikacija upravlja podacima s promjenjivom strukturom koju nije moguće unaprijed predvidjeti, statičko korisničko sučelje ne može zadovoljiti potrebe aplikacije.

Rješenje se nalazi u mogućnosti dinamičkog kreiranja korisničkog sučelja i pripadajuće programske logike, što znači da mora postojati mogućnost definiranja programske logike koja u trenutku izvršavanja aplikacije može, po određenim pravilima i zahtjevima, generirati novo korisničko sučelje i logiku te ih izvršiti. Ova priprema korisničkog sučelja temelji se na analizi promjenjivih struktura podataka tijekom izvršavanja aplikacije.

Dinamičan pristup olakšava razvoj aplikacija i s višekorisničkog aspekta u uvjetima kada postoje različite korisničke ovlasti, te je moguće ne kreirati neke zaštićene dijelove korisnicima koji na njih nemaju ovlasti.

2.2.2. NEZAVISNOST KORISNIČKOG SUČELJA OD LOGIKE APLIKACIJE

Odvajanje korisničkog sučelja od programske logike realizirano je u nekim sustava za razvoj aplikacija usmjerenih na baze podataka, a temelji se na ideji da se korisničko sučelje i programska logika mogu promatrati kao dvije neovisne cjeline, te da se jedno korisničko sučelje može zamijeniti drugim sučeljem, pri čemu je programska logika nepromijenjena, te s druge strane, da je moguće jednu programsku logiku zamijeniti drugom bez potrebe za promjenom korisničkog sučelja.

Velika prednost ovakvog pristupom jest mogućnost izrade aplikacije koje se mogu izvršavati na više tehnoloških platformi bez dodatnog razvoja posebne programske logike za svaku od platformi.

2.2.3. IZMJENJIVI REPOZITORIJ ELEMENATA APLIKACIJE

Projekti na kojima istovremeno radi više ljudi s različitim ulogama u razvojnom procesu, te kroz duži vremenski period kroz koji se mijenja kako funkcionalnost, tako i razvojni tim, zahtijevaju složene sustave pohrane izvornog koda odnosno elemenata aplikacije.

2.2.3.1. ZAPISIVANJE IZVORNOG KODA U BAZU PODATAKA

Korištenje baze podataka za pohranu veće količine podataka u višekorisničkom okruženju unaprjeđuje sustav u cjelini, što znači da to vrijedi i za pohranu izvornog koda aplikacija. Aplikacije usmjerene na baze podataka zaključavaju podatke prije njihove izmjene, kako se ne bi dogodila kolizija istovremenom izmjenom istih podataka. Na isti način, u razvojnim timovima može se zaključati dio podataka u bazi koji se odnosi na neku formu kako bi se osigurala njena izmijena od strane samo jednog člana razvojnog tima. Isto vrijedi i za korisnička prava, gdje svi članovi tima ne smiju vidjeti ili mijenjati sve dijelove aplikacije. Zaključak je da sustav za razvoj aplikacija usmjerenih na baze podataka i sam mora biti aplikacija usmjerena na bazu i to u vidu repozitorija elemenata pohranjenih u relacijskoj bazi podataka.

2.2.3.2. NADOGRADIVOST REPOZITORIJA

Ne postoji sustav koji pokriva sve moguće specifičnosti organizacija rada pojedinih razvojnih timova. Potrebna je fleksibilnost u radu sa razvojnim sustavom, to jest razvojni sustav mora biti nadogradiv i izmjenjiv kako bi se mogao prilagoditi potrebama razvojnih timova.

2.2.4. KORISNIČKI UPITI I OBRADE NAD REPOZITORIJEM

Tehnička dokumentacija bitna je u razvoju aplikacija usmjerenih na baze podataka i sadrži popis i opis formi i izvještaja, kao i opis struktura podataka. Tehnička dokumentacija se može voditi ručno, ali to u praksi može rezultirati njenom neusklađenošću sa stvarnim stanjem.

Razvojni sustav sa repozitorijem, zapisan u relacijskoj bazi, omogućuje jednostavno generiranje tehničke dokumentacije, jer je svaka forma, izvještaj i tablica zapisana u relacijskim tablicama, pa se prema tome takva dokumentacija može izraditi i automatiziranim procesom. Ovakva mogućnost, zajedno sa izmjenjivim repozitorijem u kojem se elementima mogu pridodijeliti i dodatni opisi, predstavlja veliku pomoć u rukama razvojnog tima jer osigurava točnu, iscrpnu i pravovremenu tehničku dokumentaciju u bilo kojem dijelu razvojnog ciklusa aplikacije.

Zapis izvornog koda u bazi podataka može značiti i postavljanje korisničkih upita nad elementima aplikacije, a također može omogućiti i automatsku izmjenu nekog skup podataka u repozitoriju korištenjem SQL naredbi.

2.2.5. PRILAGOĐENOST DOGAĐAJNOG SUSTAVA LOGICI UNOSA PODATAKA

Aplikacije usmjerene na baze podataka nastale su zbog specifičnih potreba višekorisničkog unosa podataka, te fleksibilnog izvješćivanja. Takve aplikacije procesiraju velike količine podataka i moraju omogućiti istovremeni unos od strane većeg broja korisnika u uvjetima korištenja transakcijskog mehanizma. Podaci zapisani u bazu moraju biti validirani i zapisani automatizmom, jer korisnicima mora biti omogućen što brži unos.

Stoga je potrebno prilagoditi događajni sustav potrebama unosa podataka u bazu podataka, umjesto logici funkcioniranja korisničkog sučelja. Prijelaz procesa unosa podataka s jednog elementa korisničkog sučelja na drugi ili s jednog redka na drugi mora izvršiti odgovarajuću akciju kako bi taj podatak bio uspješno validiran i zapisan, a odluka o tome da li neki podatak treba biti ažuriran, umetnut, obrisan ili osvježen, ne smije biti funkcionalnost koju će razvojni timovi posebno razvijati, već takav mehanizam mora biti ugrađen u sam razvojni sustav.

2.2.6. KONTROLA PROCESA UNOSA PODATAKA

Validacija je od ključne važnosti u aplikacijama usmjerenim na bazu podataka jer je ispravnost unešenih podataka preduvjet njihovog zapisivanja u bazu podataka i preduvjet nastavka daljnjeg unosa podataka. Ukoliko podatak nije ispravno unesen, proces unosa podataka mora biti vraćen na element korisničkog sučelja koji sadrži pogrešan podatak i korisnik ga mora točno popuniti.

Zbog toga mora postojati programska kontrola procesa unosa podataka na korisničkom sučelju koja omogućuje automatsko usmjeravanje procesa unosa s bilo kojeg elementa korisničkog sučelja, na bilo koji drugi element korisničkog sučelja. Takvo usmjeravanje mora biti ugrađeno u razvojni sustav i ne smije zahtijevati dodatni razvoj. Sama činjenica da postoji validacija na nekom elementu unosa podataka ili redku, mora izazvati automatski povratak procesa unosa bez zapisivanja ukoliko validacija nije potvrdila točnost podatka. Validacije mogu biti ulazne i izlazne. Izlazne validacije aktiviraju se prilikom završetka procesa unosa podataka nekog elementa korisničkog sučelja i njihovo značenje se odnosi na zadovoljavanje svih uvjeta za uspješan završetak unosa podataka u elementu. Ulazne validacije aktiviraju se prije procesa unosa podataka u elementu korisničkog sučelja i njihovo značenje se odnosi na zadovoljavanje svih preduvjeta prije početka procesa unosa podataka u elementu.

2.2.7. FLEKSIBILNA DISTRIBUCIJA APLIKACIJE

Trajanje životnog ciklusa aplikacija usmjerenih na baze podataka, te brojnost korisnika koji aplikacijama mogu pristupati putem brojnih platformi čini takve aplikacije podložne učestalim izmjenama, te se postavlja pitanje učinkovite distribucije novih verzija aplikacije korisnicima.

Klasična distribucija aplikacije vrši se putem isporuke izvršnih datoteka na poslužitelje ili na klijentska računala, potrebna je i dodatna mogućnost pohrane izvršne verzije aplikacije u bazi podataka u obliku repozitorija zbog kvalitete istovremenog višekorisničkog pristupa podacima. U tom slučaju distribucija nove verzije aplikacije vrši se ažuriranjem izvršnih tablica u bazi podataka umjesto u datotečnom sustavu, a programi izvršitelji na klijentskim strojevima čitaju, tumače i izvršavaju takav zapis aplikacije. Distribucija putem izvršnih tablica u bazi podataka omogućuje čitanje samo onog dijela zapisa aplikacije koji se neposredno izvršava, a ne aplikaciju kao cjelinu.

2.2.8. OTVORENOST SUSTAVA ZA RAZVOJ APLIKACIJA USMJERENIH NA BAZU PODATAKA

Izborom razvojnog sustava organizacija može za taj sustav biti vezana na dugo vremena, te tako postaje podložna rizicima prestanka održavanja od strane proizvođača što u slučaju promjene hardvera može biti pogubno, a mora poštivati licenčna i druga pravila i uvjete koje propisuju proizvođači razvojnih sustava. Migracija na novi razvojni sustav je skupa, jer je vrlo često potrebno ponovo napisati cijelu aplikaciju.

Stoga je poželjno da razvojni sustav bude otvoren i transparentan, a to se može postići samo ukoliko razvojni sustav dolazi i u obliku izvornog, a ne samo izvršnog koda. Postoji nekoliko mogućih načina na koji se transparentnost razvojnog sustava može postići. Jedna od njih je da razvojni sustav bude licenciran nekom od licenci otvorenog koda, a druga je da proizvođač razvojnog sustava nudi opciju kupovine izvornog koda sa pravom izmjene za vlastite potrebe.

Otvorenost izvornog koda pomaže u razumijevanju načina funkcioniranja razvojnog sustava od strane razvojnog tima, te u tom razumijevanju ne smije biti skrivenih dijelova i crnih kutija već mora postojati i odgovarajuća dokumentacija koja detaljno objašnjava sve elemente razvojnog sustava.

2.2.9. UJEDNAČENA INFRASTRUKTURA

Zbog obima podataka kojima korisnici mogu pristupati potrebno im je omogućiti slobodno filtriranje i sortiranje podataka, kako bi lakše mogli pronaći podatke koji su im neophodni, pristupiti samo dijelu podataka koji im je bitan ili poredati podatke u odgovarajućem redoslijedu.

Izrada takve infrastrukture zasebno na svakoj formi poskupljuje izradu same aplikacije, a zbog troškova često i ne bi bilo provedeno na svakoj formi čime bi korisnici bili uskraćeni za jednu značajnu funkcionalnost u aplikaciji. Zasebna izrada infrastrukture filtriranja i sortiranja ima i još jednu lošu karakteristiku, a to je da postoji mogućnost da proces filtriranja neće identično biti proveden na svim formama već će postojati razlike u funkciji i dosegu.

Stoga je nužno da razvojni sustav posjeduje ugrađenu infrastrukturu za sortiranje i filtriranje kako bi se smanjili troškovi razvoja, povećala funkcionalnost aplikacije i uspostavila ujednačenost takve infrastrukture diljem cijele aplikacije.

2.2.10. MOGUĆNOST PRISTUPA SQL-U NA NISKOM NIVOU

Relacijske baze podataka raznih proizvođača imaju svoje specifičnosti sa upravljanjem transakcijama, specifičnosti sintakse dijalekta SQL jezika, te raznih optimizacijskih metoda. Zbog uvažavanja spomenutih specifičnosti ponekad je potrebno napisati neki dio logike aplikacije direktno u SQL dijalektu ciljne baze podataka.

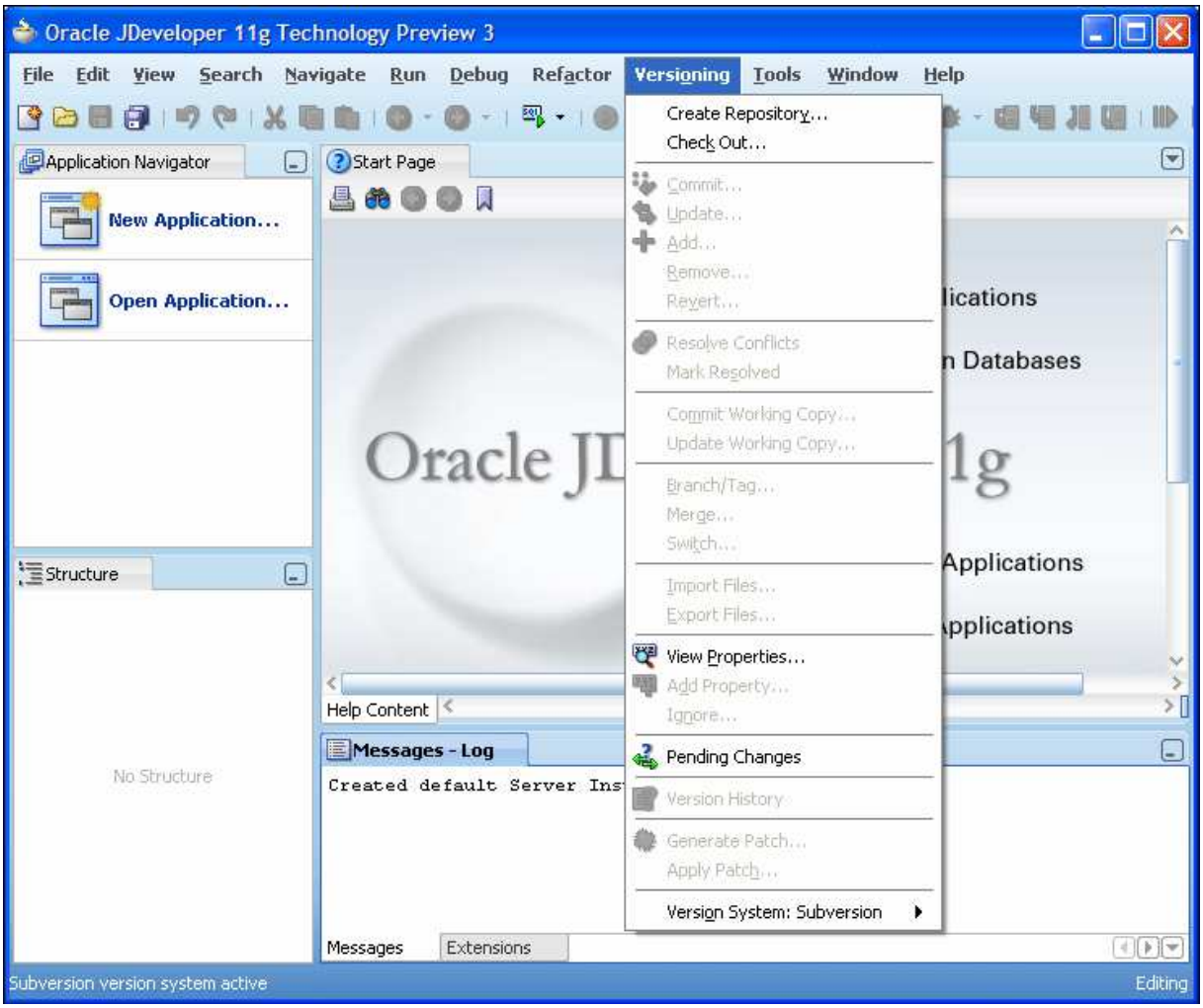
Razvojni sustav mora imati mogućnost direktnog pisanja SQL naredbi i njihove ugradnje u logiku aplikacije kako bi se postigli željeni efekti i takva mogućnost u praksi može činiti značajnu razliku u perfomansama i mogućnostima same aplikacije.

2.3. USPOREDBA ZNAČAJNIJIH SUSTAVA ZA RAZVOJ APLIKACIJA USMJERENIH NA BAZE PODATAKA

Postojeći sustavi za razvoj aplikacija usmjerenih na baze ne zadovoljavaju u potpunosti sve navedene osobine, što jest motivacija za pisanje ovog rada. Slijedi opis nekih značajnijih sustava za razvoj takvih aplikacija, te usporedba njihovih osobina sa prethodno navedenim osobinama koje sustav za razvoj aplikacija usmjerenih na bazu mora imati.

2.3.1. ORACLE JDEVELOPER

JDeveloper razvojni je sustav kompanije Oracle Corporation, nasljednik Oracle Forms sustava, a nastao je 1998 godine kupovinom JBilder razvojnog sustava kompanije Borland, te njegovim daljnjim razvojem. Glavna odlika JDevelopera je snažna naslonjenost na Java platformu i njene brojne komponente poput Swing ili EJB tehnologija, ali i Oracle Application Development Framework (tzv. Oracle ADF). Filozofija razvoja aplikacija temelji se uglavnom na objektno orjentiranom, vizualnom i deklarativnom pristupu s ciljem pojednostavljenja razvoja aplikacija (Oracle, 2012; Mills, 2009, 33-40; Koletzke, 2006, 45-52).



Slika 2.3.1 – Oracle JDeveloper

Kao i većina Java baziranih tehnologija i JDeveloper pruža brojne mogućnosti, ali uz cijenu velike kompleksnosti alata. Temeljen je dobrim dijelom na generatorima java koda, čime se

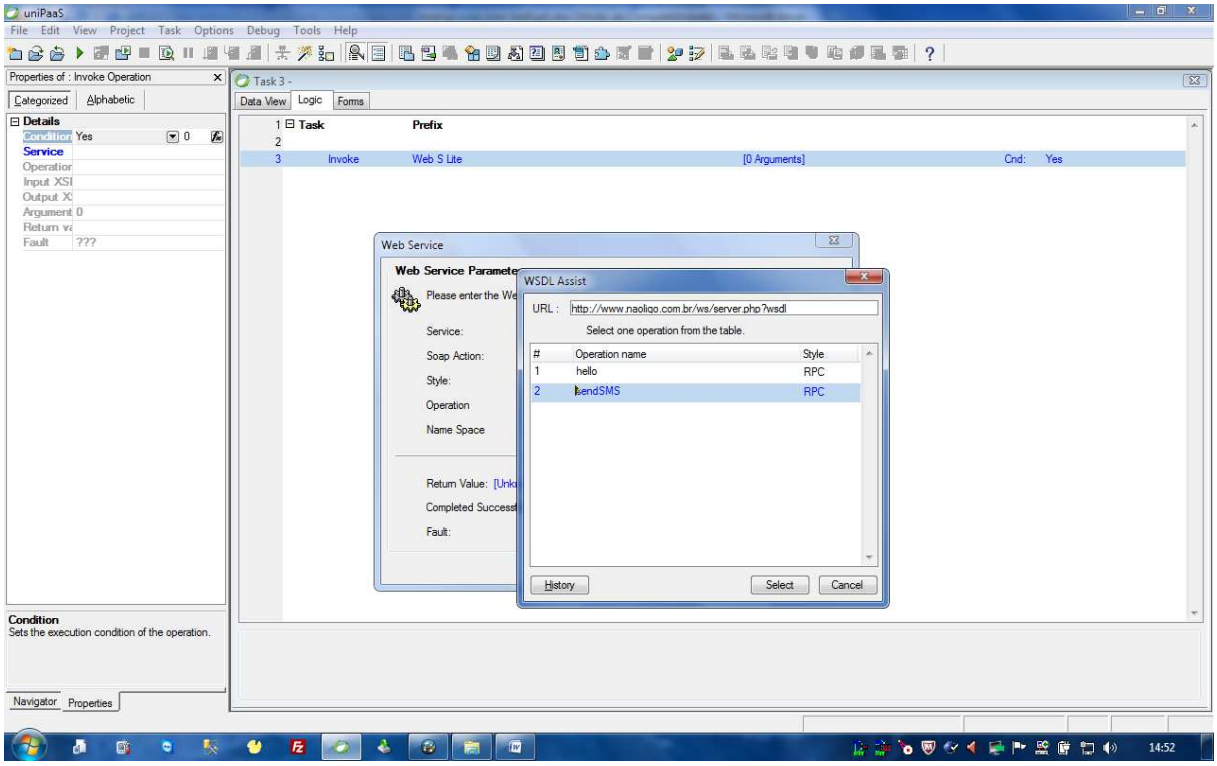
postiže velika brzina izrade prve verzije koda, ali i mala brzina kasnijih izmjena. MVC (Model View Controller) paradigma osigurava odvojenost modela podataka, korisničkog sučelja i programske logike (Haralabidis, 2012, 7-8; Vohra, 2009, 229-230; Vohra, 2008, 10, 19; Desbiens, 2009, 43-50).

Tablica 2.3.1 - Kratka usporedba željenih osobina razvojnog sustava i Jdevelopera

RB	ŽELJENA OSOBINA	KOMENTAR	OCJENA
1.	Dinamičko kreiranje korisničkog sučelja i logike aplikacije	Posljedica Java tehnologija i objektno orjentirane paradigme	DA
2.	Nezavisnost korisničkog sučelja od logike aplikacije	Model View Controller pristup	DA
3.	Izmjenjivi repozitorij elemenata aplikacije	Ne može se modificirati	NE
4.	Korisnički upiti i obrade nad repozitorijem	Ne postoje, osim pukog pretraživanja teksta	NE
5.	Prilagođenost događajnog sustava logici unosa podataka	Događajni sustav prilagođen je korisničkom sučelju i programiranju pomoću predložaka, atributa i objekata poslovne logike	NE
6.	Kontrola procesa unosa podataka	Ne postoji, potrebno je obratiti dodatnu pažnju za potpunu sigurnost.	NE
7.	Fleksibilna distribucija aplikacije	Samo izvršne datoteke	NE
8.	Otvorenost sustava za razvoj aplikacija usmjerenih na bazu podataka	Djelomična otvorenost, jer neke stvari dolaze u izvornom, a neke samo kao izvršni kod	DJELOMIČNO
9.	Ujednačena infrastruktura	Moraju se koristiti posebne filter kontrole	DJELOMIČNO
10.	Mogućnost pristupa sql-u na niskom nivou	Postoji mogućnost direktnog SQL-a	DA

2.3.2. UNIPAAS

UniPaaS predstavlja novo ime za razvojni sustav koji je nastao još 80-tih godina prošlog stoljeća, kada se zvao Magic, a poslije eDeveloper. Proizvela ga je izraelska kompanija Magic Enterprises, prvo za potrebe izraelske vojske, policije i državne uprave, a zatim i za globalno tržište (Magic Software, 2012; Schuppenhauer, 2007, 21-30).



Slika 2.3.2 – UniPaaS

UniPaaS je zaokružen sustav baziran na repozitorijima i napisan specijalno za razvoj aplikacija usmjerenih na baze podataka. Glavne prednosti to sustava su unos logike aplikacije u tablicama, kao i odličan događajni podsustav. U starijim verzijama imao je osobinu čuvanja izvornog i zvršnog koda u relacijskim tablicama, ali se u zadnjim verzijama od toga odustalo, te se danas kod čuva u XML datotekama.

Glavni nedostatak UniPaaS-a je njegova zatvorenost, kao i kruta licenčna politika Magic Enterprisea.

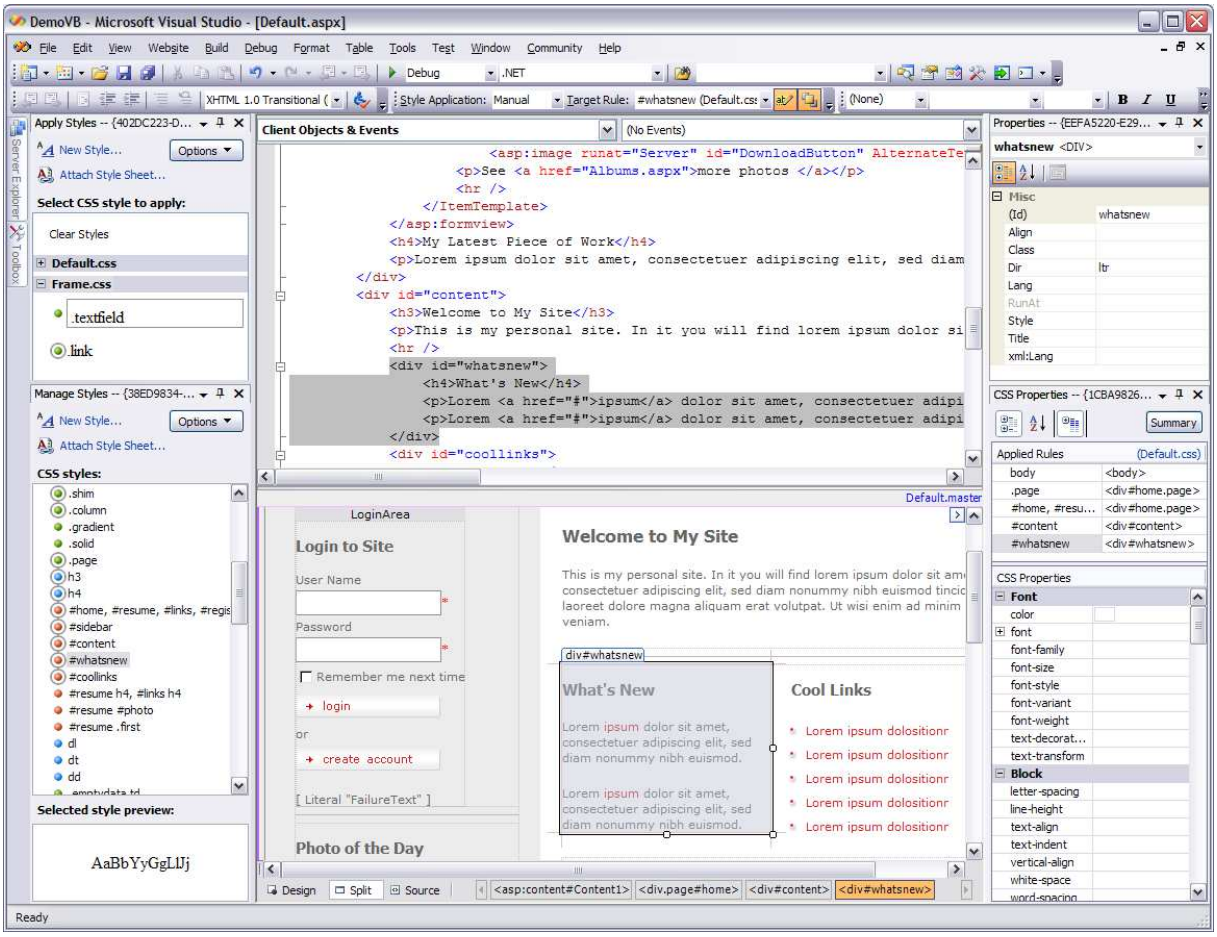
Tablica 2.3.2 - Kratka usporedba željenih osobina razvojnog sustava i UniPaaS-a

RB	ŽELJENA OSOBINA	KOMENTAR	OCJENA
1.	Dinamičko kreiranje korisničkog sučelja i logike aplikacije	Ne postoji, svaka forma (i programska logika) se kreira kroz dizajnerski alat na statički način.	NE
2.	Nezavisnost korisničkog sučelja od logike aplikacije	Nezavisnost postoji, programska logika može imati više korisničkih sučelja	DA
3.	Izmjenjivi repozitorij elemenata aplikacije	Ne postoji, sustav je zatvoren	NE
4.	Korisnički upiti i obrade nad repozitorijem	Moguće je filtriranje složenijim upitima, ali nije moguće raditi nikakve obrade	DJELOMIČNO
5.	Prilagođenost događajnog sustava logici unosa podataka	Događajni sustav je u potpunosti prilagođen logici unosa podataka	DA
6.	Kontrola procesa unosa podataka	Događajni sustav posjeduje sve kontrole ulaska i izlaska	DA
7.	Fleksibilna distribucija aplikacije	Ne postoji, samo izvršni kod	NE
8.	Otvorenost sustava za razvoj aplikacija usmjerenih na bazu podataka	Ne postoji, sustav je zatvoren i netransparentan.	NE
9.	Ujednačena infrastruktura	Postoji, ova infrastruktura je automatski ugrađena i ne mora se posebno programirati	DA
10.	Mogućnost pristupa sql-u na niskom nivou	Moguće je pisati direktni SQL	DA

2.3.3. VISUAL STUDIO .NET

Microsoft Visual Studio .NET popularan je razvojni sustav opće namjene koji podržava razvoj aplikacija za .NET platformu u više programskih jezika, od kojih su najpopularniji Visual Basic, C++ i C#. Podržava objektno orjentirano programiranje, vizualni pristup dizajniranju korisničkog sučelja, te posebnu kolekciju komponenti za pristup bazama podataka ADO.NET (Microsoft Corporation, 2012; Halvorson, 2010, 4-11; Banks, 2012, 5).

Razvoj aplikacija usmjerenih na baze podataka oslanja se na korištenje ADO.NET komponenti koje mogu pristupati brojnim relacijskim bazama podataka. ADO.NET podržava objektno-relacijske pretvarače (eng. object relational mapper) te automatsko generiranje SQL komandi ili objektni pristup podacima. Posjeduje repozitorij izvršnog koda koji se zapisuje kao kolekcija datoteka (Gousset, 2012, 1-8; Troelsen, 2012, 1-30; Lobel, 2012, 3-9; Agarwal, 2012, 1-20; Del Sole, 2012, 1-10; Stephens, 2012, 3-6).



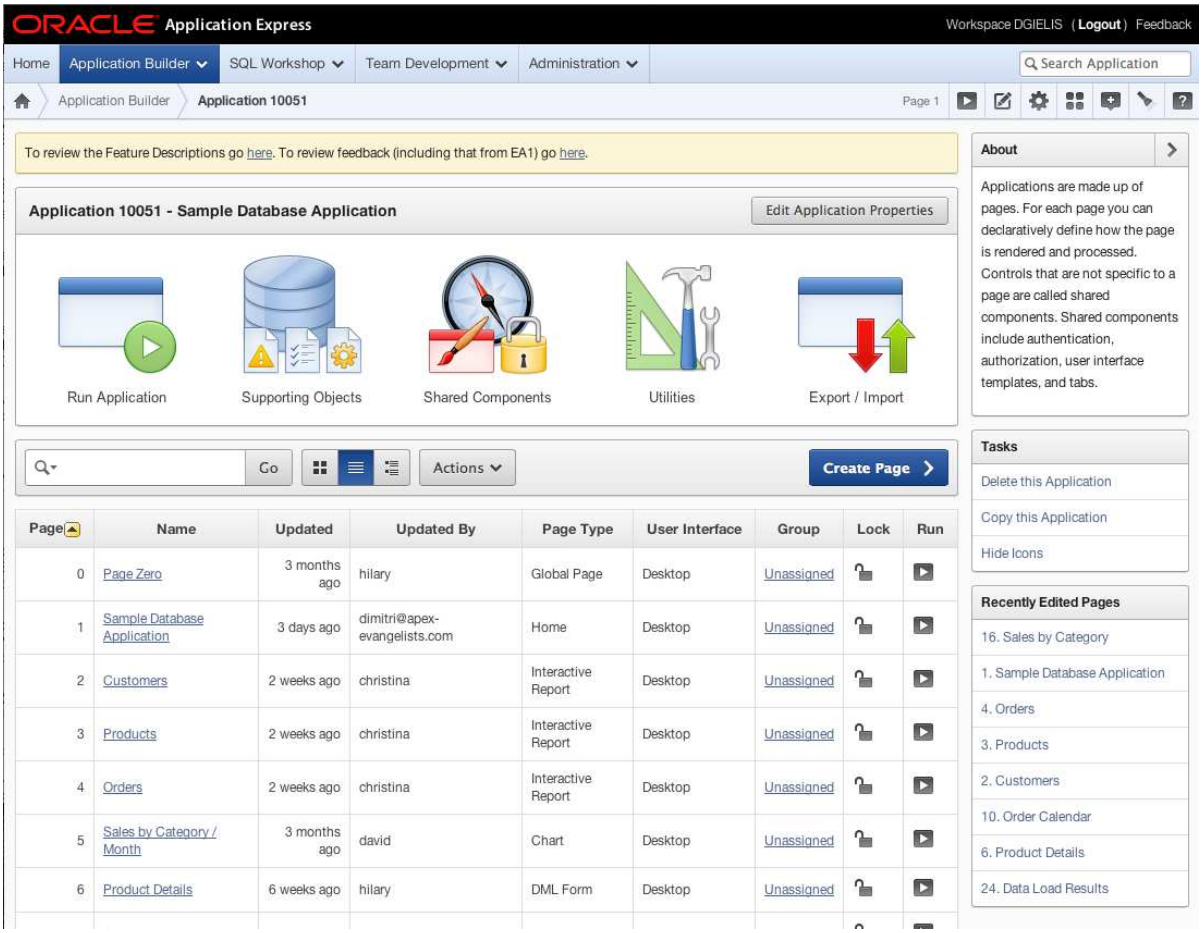
Slika 2.3.3 – Microsoft Visual Studio .NET

Tablica 2.3.3 - Kratka usporedba željenih osobina razvojnog sustava i Visual Studio .NET

RB	ŽELJENA OSOBINA	KOMENTAR	OCJENA
1.	Dinamičko kreiranje korisničkog sučelja i logike aplikacije	Zbog objektno orjentiranog pristupa programiranju i .NET platforme	DA
2.	Nezavisnost korisničkog sučelja od logike aplikacije	Model View Controller pristup	DA
3.	Izmjenjivi repozitorij elemenata aplikacije	Repozitorij se ne može modificirati	NE
4.	Korisnički upiti i obrade nad repozitorijem	Ne postoje, osim pretraživanja teksta	NE
5.	Prilagođenost događajnog sustava logici unosa podataka	Događajni sustav prilagođen je korisničkom sučelju i programiranju pomoću objekata.	NE
6.	Kontrola procesa unosa podataka	Ne ostoji, potrebno je obratiti dodatnu pažnju za potpunu sigurnost.	NE
7.	Fleksibilna distribucija aplikacije	Samo izvršne datoteke	NE
8.	Otvorenost sustava za razvoj aplikacija usmjerenih na bazu podataka	Djelomična otvorenost, jer neki dijelovi dolaze u izvornom, a neki samo kao izvršni kod	DJELOMIČNO
9.	Ujednačena infrastruktura	Postoji standardna infrastruktura sortiranja i filtriranja	DA
10.	Mogućnost pristupa sql-u na niskom nivou	Postoji mogućnost direktnog SQL-a	DA

2.3.4. APEX

APEX ili Oracle Application Express je popularni sustav za razvoj aplikacija usmjerenih na baze podataka ali isključivo za web platformu, a sastoji se od kolekcije objekata na Oracle bazi podataka koja komunicira s vanjskim svijetom preko Oracle aplikacijskog servera ili klasičnog web servera (Oracle, 2012b; Ahmed, 2011, 1-9; Gault, 2011, 1-29).



Slika 2.3.4 – Oracle APEX

Prednost ovakvog pristupa nalazi se u oslanjanju na relacijsku bazu podataka u pogledu interne funkcionalnosti sustava. APEX omogućuje brzo kreiranje aplikacija koje nemaju velike zahtjeve na kontroli podataka i kompleksnoj logici (Fox, 2011, 1-3; Zehoo, 2011, 1-10; Greenwald, 2008, 1-8).

Nedostatak APEX-a sastoji se u tome što nije u potpunosti prilagođen logici unosa podataka u relacijsku bazu, već je prije prilagođen web tehnologijama (Cimolini, 2012, 15-25).

Tablica 2.3.4 - Kratka usporedba željenih osobina razvojnog sustava i Apex-a

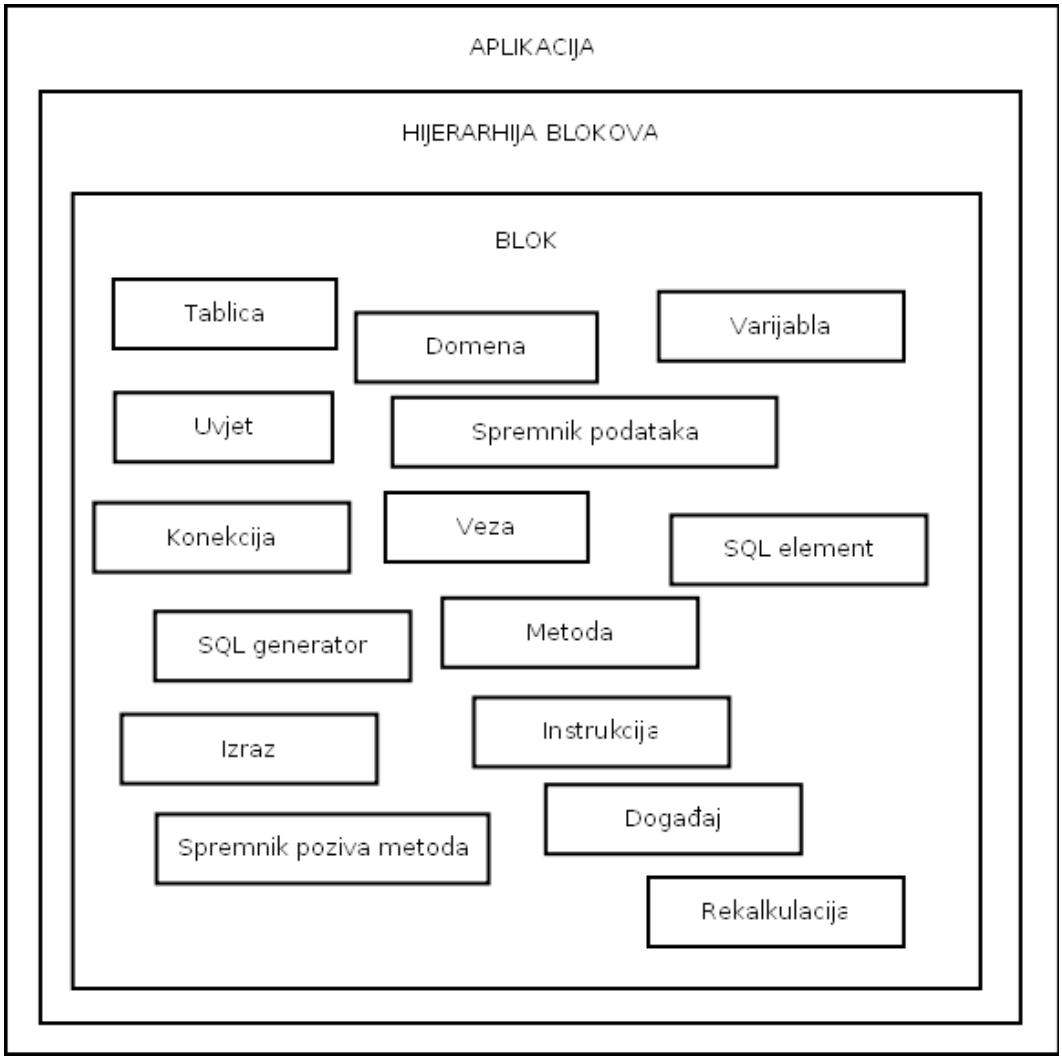
RB	ŽELJENA OSOBINA	KOMENTAR	OCJENA
1.	Dinamičko kreiranje korisničkog sučelja i logike aplikacije	Ne postoji, forme se ne mogu izrađivati programski.	NE
2.	Nezavisnost korisničkog sučelja od logike aplikacije	Korisničko sučelje je usko vezano uz logiku aplikacije	NE
3.	Izmjenjivi repozitorij elemenata aplikacije	Repozitorij je nepromjenjiv	NE
4.	Korisnički upiti i obrade nad repozitorijem	Ne postoje, osim pretraživanja teksta	NE
5.	Prilagođenost događajnog sustava logici unosa podataka	Ne postoji, događajni sustav je prilagođen web tehnologijama	NE
6.	Kontrola procesa unosa podataka	Djelomično, ali ne postoji potpuna kontrola	DJELOMIČNO
7.	Fleksibilna distribucija aplikacije	Ne postoji, distribucija moguća samo u obliku web aplikacije	NE
8.	Otvorenost sustava za razvoj aplikacija usmjerenih na bazu podataka	Sustav dolazi u obliku izvornog koda i teoretski ga je moguće nadograđivati, iako nije tome namijenjen	DJELOMIČNO
9.	Ujednačena infrastruktura	Postoji standardna infrastruktura za filtriranje i sortiranje podataka	DA
10.	Mogućnost pristupa sql-u na niskom nivou	Postoji mogućnost direktnog SQL-a	DA

Iako je gore navedeni izbor razvojnih sustava nepotpun jer se odnosi samo na neke popularnije sustave, može se vidjeti da su oni samo djelomično usklađeni sa osobinama koje mora imati sustav za razvoj aplikacija usmjerenih na bazu podataka.

Cilj ovog rada je definirati otvoreni sustav koji je u potpunosti prilagođen zahtjevima razvoja aplikacija usmjerenih na baze podataka, to jest jedan apstraktni model koji ima potencijal, ovisno o konkretnoj implementaciji u konkretnoj tehnologiji, zadovoljiti sve gore navedene osobine.

3. APSTRAKTNI MODEL

Apstraktni model sastoji se od kolekcije elemenata čije je međusobno djelovanje u skladu s principima i specifičnostima razvoja aplikacija usmjerenih na baze podataka. Elementi slijede principe objektno-orijentiranog dizajna i zamišljeni su da se osigura njihova što jednostavnija implementacija na postojećim razvojnim okruženjima.



Slika 3.1 – Shematski prikaz elemenata apstraktnog sustava

3.1. ELEMENTI APSTRAKTNOG MODELA

Formalniji opis elemenata prikazan je u obliku skupa bitnih atributa i metoda sa opisom algoritama.

Slika 3.1 prikazuje shematski prikaz elemenata apstraktnog sustava. Slijedi kratak opis svakog od elemenata.

(1) APLIKACIJA - Aplikacija je krovni element apstraktnog modela koji okuplja ostale elemente. Glavna zadaća aplikacije je kreiranje i pozivanje blokova, te o čuvanje sadržaja globalnih varijabli

(2) BLOK - Blok je osnovni gradivni element na koji se pozivaju ostali elementi koji sudjeluju u realizaciji nekog podskupa funkcionalnosti odnosno osnovne logičke cjeline aplikacije. Blokovi mogu biti organizirani u hijerarhijske cjeline.

(3) TABLICA - Tablica u apstraktnom modelu je element čija je zadaća pobliže specificirati tablicu ili pogled u bazi podataka i način na koji se ona koristi unutar bloka. Tablica u apstraktnom modelu može biti središnja ili vezana.

(4) VARIJABLA - Varijabla je element koji pobliže određuje bazna polja iz tablica koje se koriste, nebazne varijable bloka, parametre i vezu prema globalnim varijablama aplikacije.

(5) DOMENA - Domena je element tipa podatka. Postoji nekoliko osnovnih tipova podataka: numerički, znakovni, datumsko-vremenski, logički i generički. Domena određuje veličinu i preciznost podatka, kao i format njegovog prikaza na zaslonu. Domena sudjeluje u transformaciji zapisa podataka između baze, spremnika podataka i zaslona.

(6) UVJET - Uvjet je element koji ima dvojaku ulogu. S jedne strane služi za određivanje kriterija filtriranja podataka u bloku, a s druge strane za opisivanje veza između tablica koje se koriste u bloku.

(7) VEZA - Veza (eng. join) je element koji opisuje način na koji se međusobno povezuju tablice. Ako se ne kreiraju veze, povezivanje ide preko WHERE klauzule SQL-a. Ukoliko se navede veza, onda ona može biti unutarnja (INNER JOIN klauzula SQL-a) ili lijeva-vanjska (LEFT OUTER JOIN klauzula SQL-a). Moguće je definirati i druge vrste veza među tablicama ukoliko su podržane SQL dijalektom ciljane baze podataka.

(8) SPREMNIK PODATAKA - Zadaća spremnika podataka je pohrana podataka dohvaćenih iz baze, kao i čuvanje vrijednosti svih nebaznih varijabli. Spremnika podataka sudjeluje u svakom čitanju i zapisivanju vrijednosti. Spremnik je zadužen za evidenciju korisničkih promjena nad dohvaćenim podacima.

(9) KONEKCIJA - Konekcija je element koji služi za ostvarivanje veze prema bazi podataka, putem korisničke prijave. Konekcija interno otvara kursore za komunikaciju s relacijskom bazom, šalje SQL naredbe, dohvaća podatke, te počinje i završava transakcije.

(10) SQL ELEMENT - Zadaća SQL elementa je uskladiti opis modela podataka u bloku (tablica, polje, uvjet) sa pojedinim zahtjevom za SQL naredbom: selektiranjem, brisanjem, ažuriranjem ili dodavanjem podataka u bazu, te poziva procedura iz baze. Svaka SQL naredba ima svoj SQL element u bloku.

(11) SQL GENERATOR - SQL generator služi za konstrukciju SQL naredbe koja se putem elementa konekcije šalje prema bazi podataka, a na temelju SQL elementa.

(12) METODA - Zadaća metode unutar bloka jest objediniti određeni programski kod u jednu cjelinu kako bi se mogao pozivati s više mjesta. Metoda može imati ulazne i izlazne parametre, te lokalne varijable i povratnu vrijednost. Metode je moguće pozivati i rekurzivno.

(13) INSTRUKCIJA - Instrukcija odgovara pojedinim naredbama u klasičnim programskim jezicima. Instrukcije izvršavaju pojedine jednostavnije zadaće poput uvjetnog grananja, pridruživanja vrijednosti nekoj varijebli i slično.

(14) IZRAZ - Izraz je element koji sadrži tekstualnu reprezentaciju računskog izraza koji je u nekom trenutku moguće izračunati. To, primjerice, može biti neka formula ili poziv neke funkcije, metode ili bloka.

(15) SPREMNIK POZIVA METODA - Spremnik poziva metoda je sastavni dio spremnika podataka, a njegova uloga je čuvati podatke nužne za izvršavanje svakog pojedinog poziva metode. Takvi podaci su instrukcija koja se trenutno izvršava, vrijednosti lokalnih varijabli i slično.

(16) DOGAĐAJ - Događaj je element čija je zadaća reagirati na događaje koji se odvijaju unutar bloka. Kada se dogodi unaprijed registrirani događaj, izvršava se pridružena programska logika. Svaki događaj ima pridruženu metodu koja je nositelj programske logike.

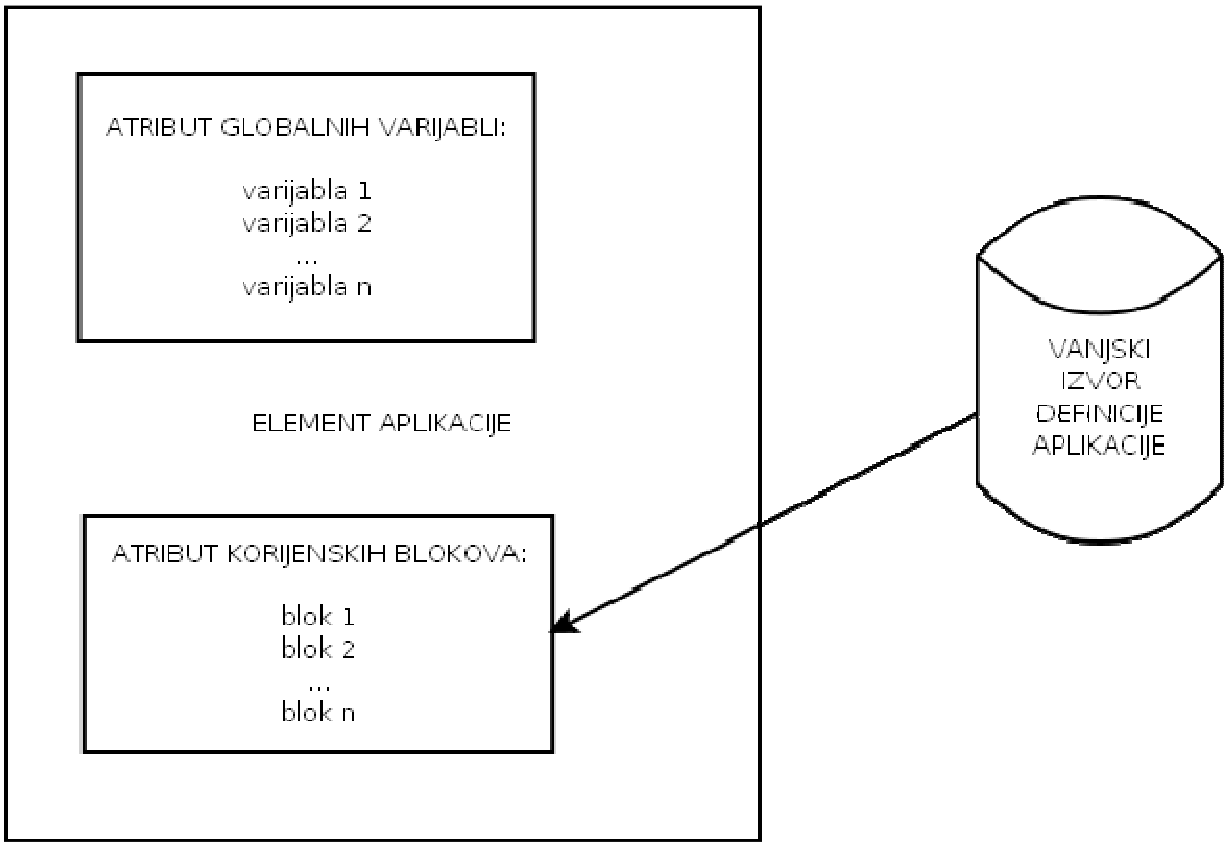
(17) REKALKULACIJA - Rekalkulacija je element koji ima ulogu koju imaju formule u proračunskim tablicama. Promijena vrijednost neke varijable izaziva promjene niza drugih varijabli. Element rekalkulacije brine i o inicijalnim vrijednostima varijabli.

3.2. APLIKACIJA

Aplikacija je krovni element apstraktnog modela koji objedinjuje ostale elemente u cilju izvršavanja zadane funkcionalnosti aplikacije usmjerene na bazu podataka. Element aplikacije zadužen je za čuvanje vrijednosti globalnih varijabli aplikacije, te za proces učitavanja i izvršavanja blokova.

Globalne varijable mogu imati pohranjene vrijednosti koje se mogu čitati i ažurirati iz bilo kojeg bloka. U globalnim variablama mogu biti pohranjene informacije o korisničkoj prijavi, korisnikovoj organizacijskoj jedinici, sigurnosnom nivou i slično. Blokovi globalnim varijablama ne mogu pristupati direktno, već se veza između blokova i globalnih varijabli ostvaruje definiranjem varijable bloka koje imaju referencu ciljne globalne varijable. Punjenjem i čitanjem tako definiranih lokalnih varijabli pune se i čitaju globalne varijable.

Apstraktni model mora imati mogućnost pohrane aplikacije na vanjskim izvorima podataka. Aplikacija može biti pohranjena u lokalnoj datoteci, ali i u relacijskoj bazi podataka na udaljenom poslužitelju, ili na internetskom poslužitelju, što zahtijeva veliku fleksibilnost dohvaćanja definicije blokova. Stoga aplikacija ima mogućnost učitavanja bloka iz unaprijed definiranog izvora. Prilikom poziva bloka, koji se specificira po imenu, iz vanjskog izvora se dohvaća zapis o bloku i njegovim pripadajućim elementima, te se na temelju tog zapisa kreiraju stvarni elementi spremni za izvršavanje. Izvor se može definirati kao mjesto gdje se nalazi definicija aplikacije, odnosno odakle se aplikacija poziva. To može biti datotečni sustav gdje se nalazi datoteka s aplikacijom, relacijska baza u kojoj se nalaze tablice s definicijom elemenata aplikacije ili internetska adresa preko koje je moguće dohvatiti aplikaciju. Dohvaćanjem definicije bloka iz vanjskog izvora, kreira se element bloka i pridodjeljuje elementu aplikacije. Izvršavanje započinje pozivom odgovarajućih metoda bloka.



Slika 3.2 – Aplikacija s dohvatom i izvršavanjem blokova te globalnim varijablama

3.3. BLOK

Za ispravno funkcioniranje apstraktnog modela nužno je ogranizirati aplikaciju u manje cjeline koje odgovaraju logici funkcioniranja aplikacije.

Aplikacije usmjerene na rad s bazama podataka rade sa podacima koje možemo prikazati u obliku skupa relacija koje su u logičkom odnosu i nad čijim n-torkama se vrše različite operacije u skladu s potrebama problemskog područja. Struktura aplikacije slijedi relacije, pa su forme za pregled i unos podataka organizirane oko pojedinih relacija. Unos zaglavlja i stavki dokumenata mora biti predstavljena dvjema relacijama (jednom za zaglavlje i drugom za stavke) i one predstavljaju dvije odvojene cjeline sa različitom programskom logikom i ograničenjima, iako su u međusobnom čvrstom odnosu roditelj-djeca.

Predloženi apstraktni model slijedi takav princip, te organizira aplikaciju u manje cjeline koje se mogu nazvati blokovima, a svaki blok se odnosi na jednu i samo jednu relaciju (s pripadajućom shemom i n-torkama). No, kako se nad jednom relacijom može obavljati više različitih korisničkih funkcija, odnosno, relacija može imati više uloga u aplikaciji, onda nije

nužno da se jedna relacija pojavljuje u samo jednom bloku. Relacija dokument može se pojaviti u bloku liste dokumenata, ali i u bloku unosa zaglavlja jednog dokumenta.

ZAGLAVLJE DOKUMENTA		
Broj dokumenta	Datum dokumenta	Vrsta dokumenta
1	01.10.10	Ulazni
2	02.10.10	Izlazni

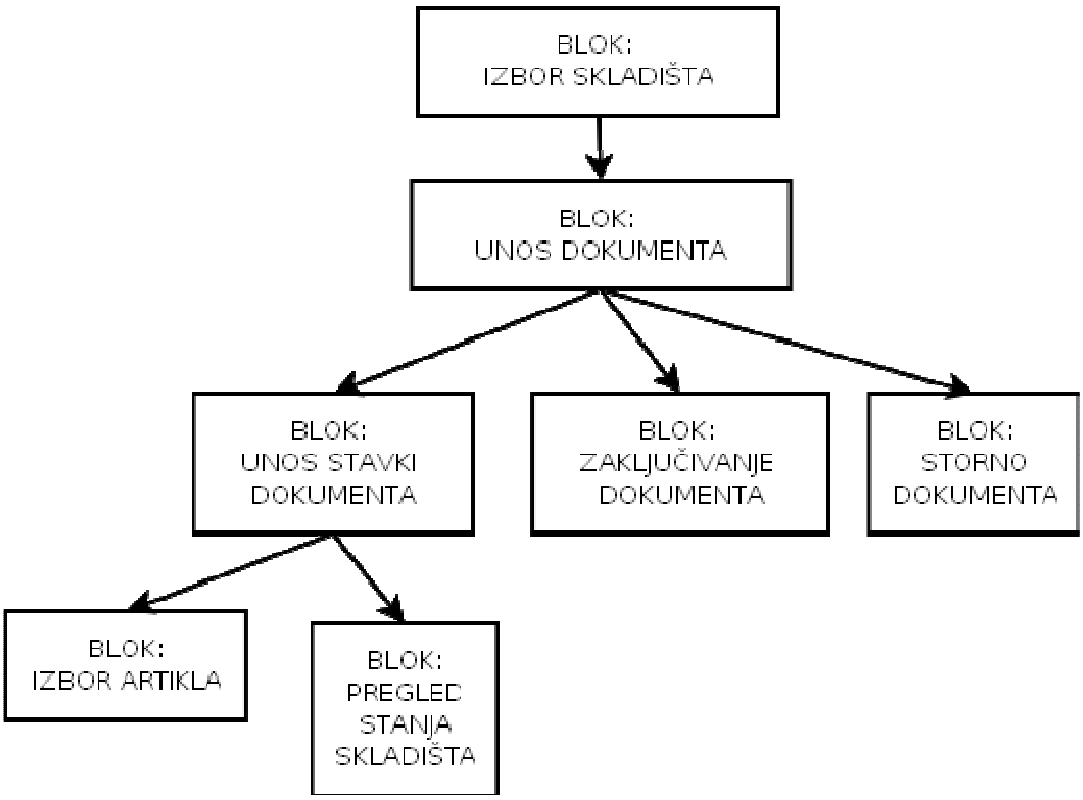
STAVKA DOKUMENTA			
Rbr stavke	Broj dokumenta	Artikl	Količina
1	1	Košulja	100
2	1	Hlače	40
3	1	Kaput	20
1	2	Košulja	7
2	2	Hlače	11

Slika 3.3a – Primjer relacija zaglavlja i stavke dokumenata

Na slici 3.3a može se vidjeti da zaglavlja predstavljaju jednu relaciju, a stavke dokumenata drugu, te bi za njihov pregled i ažuriranje bila potrebna najmanje dva bloka.

Kako je ovaj apstraktni model namijenjen radu s konkretnim implementacijama relacijskih baza podataka, tako relacije s kojima rade blokovi treba definirati preko objekata baze, a to su tablice, pogledi, polja u tablici i slično.

Blok je definiran kao osnovna gradivna jedinica apstraktnog modela. Iz primjera sa slike 3.3a vidi se da se prikazani blokovi nalaze u odnosu roditelj-dijete, te da blok stavaka može imati samo stavke dokumenta koje pripadaju dokumentu izabranom u bloku zaglavlja. Prilikom ažuriranja podataka jedne stavke, može postojati potreba za direktnim referenciranjem na podatke sa zaglavlja. Takva mogućnost međusobnog komuniciranja blokova unaprjeđuje potencijal apstraktnog modela, a postiže se hijerarhijskim strukturama blokova, sa korijenskim blokom na vrhu (slika 3.3b).



Slika 3.3b – Hijerarhijska struktura blokova

Hijerarhijska struktura blokova se sastoji od barem jednog bloka, ili više njih organiziranih u strukturu stabla. Svaki blok može pristupiti podacima svih roditeljskih blokova, te pozvati izvršenje direktno podređenih blokova. Blok ne može pristupiti podacima svoje djece, niti podacima blokova koji su na istom hijerarhijskom nivou.

Hijerarhije omogućuju grupiranje blokova koji zajednički sudjeluju u realizaciji nekog podskupa funkcionalnosti aplikacije u jednu čvršću cjelinu koja interno ima intenzivnu međusobnu komunikaciju blokova, olakšava održavanje, te doprinosi preglednosti aplikacije. Hijerarhija je predstavljena korijenskim blokom, te se izvršavanjem korijenskog bloka izvršava cijela hijerarhijska struktura. Bitno je napomenuti da element aplikacije po imenu može startati samo korijenske blokove ili blokove koji nisu u hijerarhiji. Ostali blokovi mogu se pozivati isključivo preko nadređenog roditelja.

Osim uloge osnovnog gradivnog elementa, blok služi kao poveznica ostalih elemenata i koordinira njihov rad. Blok je početna i završna točka izvršavanja ostalih elemenata, poziva

metode i reagira na događaje, te brine o izvršnom ciklusu gdje se uspostavlja ispravan redoslijed izvršavanja događaja.

3.4. OPIS MODELA PODATAKA

Sustavi za razvoj aplikacija usmjerenih na baze podataka moraju sadržavati opise struktura podataka s kojima rade, stoga apstraktni model sadrži nekoliko elemenata koji služe toj zadaći.

3.4.1. TABLICE BLOKA

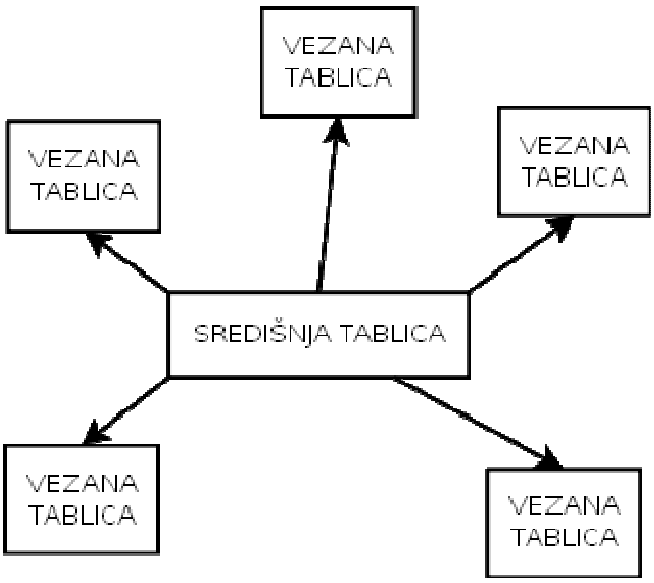
Podaci su u relacijskim bazama podataka zapisani u tablicama. Tablice predstavljaju imenovane objekte baze sa precizno definiranim poljima, tipovima podataka, indeksima, te stranim i primarnim ključevima. Osim tablica, u relacijskim bazama postoje i pogledi (eng. view) koji funkcioniraju poput tablica, a zapravo su sastavljeni od upita na jednu ili više njih.

Element tablice služi za precizno određivanje tablice iz baze podataka koja će biti korištena u bloku. Tablicama u bazi pristupa se preko baznih konekcija koje su u apstraktnom modelu definirane elementima konekcija.

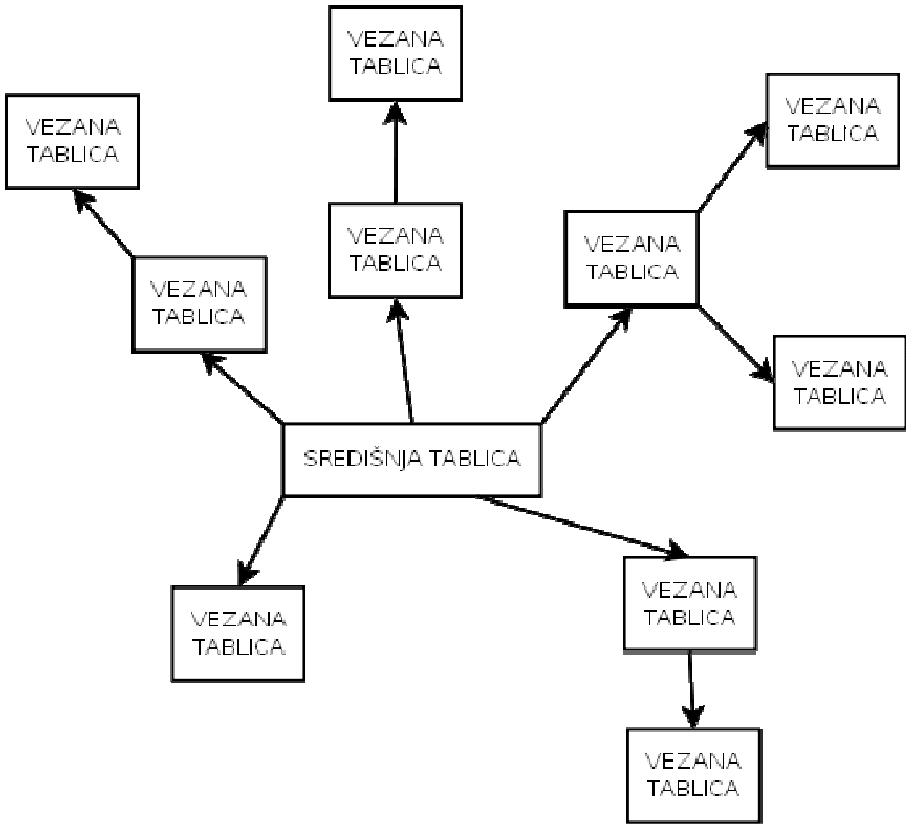
Tablice u bloku mogu preuzeti ulogu središnje ili neke od vezanih tablica. Jedna tablica iz baze podataka može imati više uloga unutar jednog bloka, pri čemu se za svaku od uloga definira zaseban element tablice. To je ujedno i razlog zbog kojeg svaki element tablice mora imati i svoj nadimak (tzv. alias) pod kojim će biti korišten u SQL naredbama.

3.4.2. SREDIŠNJA TABLICA

Apstraktni model preuzima logiku definiranja tablica iz discipline skladištenja podataka u kojoj se koriste zvjezdasta shema i shema snježne pahuljice. U obje sheme postoji jedna središnja tablica, takozvana tablica činjenica (eng. fact fable) s kojom su preko stranih ključeva povezane druge tablice, tzv. dimenzije ili vezane tablice.



Slika 3.4.2a – Zvjezdasta shema



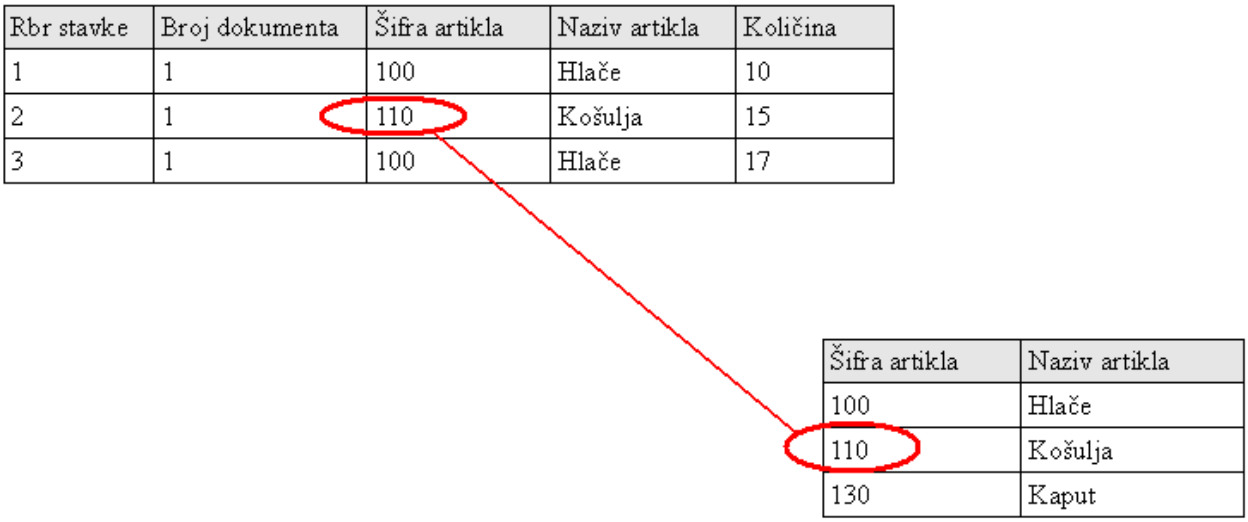
Slika 3.4.2b – Shema snježne pahuljice

Ovaj model je učinkovit, jer primarni ključ središnje tablice služi kao primarni ključ cijele sheme i pojednostavljuje manipulaciju podacima. Iz istog je razloga evidentno da blok ne može imati više od jedne središnje tablice.

No blok ne mora nužno imati središnju tablicu, nego se vezane tablice mogu povezivati i na vrijednost nekog virtualnog polja. Ako nije navedeno ime tablice u bazi, onda se tablica mora definirati kao direktni SQL element.

3.4.3. VEZANE TABLICE

Vezane tablice ili dimenzije su tablice koje se vežu na središnju tablicu po principu stranog ključa. Osim na središnju tablicu, vezane tablice se mogu povezivati s drugim vezanim tablicama ili s nekom vrijednošću varijable.



Slika 3.4.3 – Primjer stavaka dokumenta sa šifarnikom artikala

Vezane tablice dohvaćaju samo jedan slog koji proširuje podatke sloga središnje tablice. U konkretnoj implementaciji apstraktnog modela, korištenjem objektno-orjentirane paradigme, moguće je svaku od ovih podvrsta definirati kao zasebne klase, odnosno podelemente.

3.4.4. VARIJABLE

Varijable u bloku podijeljene su na tri vrste: bazna polja, virtualna polja i parametre. Varijabla sama po sebi ne sadrži vrijednosti, već se podacima u varijablama pristupa posredno preko metoda Uzmi i Postavi(vrijednost) koje komuniciraju sa spremnikom podataka u bloku.

Bazna polja u bloku odnose se na polja iz središnje tablice i vezanih tablica koje su definirane u bloku. Tip baznog polja mora odgovarati tipu polja iz definicije tablice u relacijskoj bazi. Bazno polje sadrži informaciju o tome na koju tablicu se odnosi, kako se zove polje iz tablice, da li je to polje dio SELECT, INSERT ili UPDATE instrukcija, da li polje sudjeluje u redoslijednoj definiciji (ORDER BY klauzula), da li je dio definicije grupiranja (GROUP BY klauzula), te da li se na polje primjenjuje neka od agregatnih funkcija SQL-a. Osim ovoga, bitan je podatak i to da li je polje dio ključa po kojem se može jednoznačno definirati slog u tablici, a koji se naziva identifikator redka. Neke baze podataka imaju i implicitno određeni identifikator svakog sloga u tablici (tzv. ROWID), pa bazno polje može preuzeti i vrijednosti takvog indikatora.

Ne nalaze se svi podaci koji su bitni za funkcioniranje bloka u tablicama relacijske baze. Postoji potreba za privremenim varijablama, međuzbrojevima, različitim indikatorima i slično, koje nije potrebno pamtiti u bazi. Stoga postoje virtualne varijable u kojima su podaci pohranjeni u memoriji i traju samo za vrijeme izvršavanja bloka. Virtualna polja su podijeljena u tri osnovne kategorije. Prva kategorija su virtualna polja na nivou bloka, što znači da ta varijabla ima jednu vrijednost koja vrijedi za cijeli blok. Slijedeća kategorija su virtualna polja na nivou redka, tj. polja koja imaju zasebnu vrijednost za svaki redak glavne tablice u bloku. I konačno, treća kategorija su globalne varijable, koje su zapravo reference na globalne varijable na nivou aplikacije s ciljem da mogu biti korištene u bloku. Osim na nivou bloka, varijable mogu biti i lokalne varijable unutar neke od metoda bloka što se naziva dosegom varijable. Blok mora komunicirati sa vanjskim svijetom, a način da se to osigura jest korištenje parametara. Blok može imati definirane parametre i to ulazne, izlazne te ulazno-izlazne. Parametri se ponašaju kao virtualna polja bloka.

Varijablama se pristupa preko njihovog imena, koje mora biti jedinstveno unutar bloka za virtualne i bazne varijable. Jedini izuzetak su lokalne varijable metoda pa dvije metode mogu imati lokalne varijable jednakog imena.

Element varijable mora sadržavati sve neophodne podatke o karakteristikama varijable, njezinu podvrstu, pripadajuću tablicu i polje, tip podatka, format prikaza na korisničkom sučelju i slično, a element varijable može čuvati i podatke koji su važni za generiranje korisničkog sučelja.

Apstraktni model definira krovni element varijable, a onda još i nekoliko podvrsta, odnosno podelemenata. Varijable, s obzirom na njihovu tipičnu uporabu mogu dolaziti u nekoliko varijanti, odnosno podelemenata. Oni se međusobno razlikuju u vrijednostima nekih atributa koji su specificirani u tablici 3.4.4.

Tablica 3.4.4 - Osobine podelemenata

Podelement	Da li je dio redka?	Ulazni parametar?	Izlazni parametar?	Ime globalne varijable	Bazno polje	Bazna tablica
Bazno polje	Ne	Ne	Ne	Prazno	Ime polja	Ime tablice
Virtualna varijabla redka	Da	Ne	Ne	Prazno	Prazno	Prazno
Virtualna varijabla bloka	Ne	Ne	Ne	Prazno	Prazno	Prazno
Globalna varijabla	Ne	Ne	Ne	Ime globalne varijable	Prazno	Prazno
Ulazno izlazni parametar	Ne	Da	Da	Prazno	Prazno	Prazno
Ulazni parametar	Ne	Da	Ne	Prazno	Prazno	Prazno
Izlazni parametar	Ne	Ne	Da	Prazno	Prazno	Prazno

3.4.5. UVJET

Element uvjeta koristi se za određivanje kriterija filtriranja podataka unutar bloka i za definiranje kriterija spajanja tablica.

Dvije su osnovne kategorije uvjeta unutar bloka; uvjeti definirani od strane članova razvojnog tima koji se ne mogu mijenjati tijekom izvođenja aplikacije jer o njima ovisi ispravnost izvođenja, te korisnički, koje korisnik može samostalno dodavati, mijenjati i brisati i koji u

pravilu služe za dodatnu restrikciju opsega podataka u bloku a u skladu s korisničkim potrebama.

Postoje tri vrste uvjeta, odnosno tri podelementa:

(1) Uvjeti s operacijom

Uvjeti s operacijom se odnose na dovođenje polja iz bloka u odnos s nekim drugim poljem ili izrazom. Operacije mogu biti klasične operacije uspoređivanja =, <, <=, >, >=, <> ili tipične SQL operacije IS NULL, IN, BETWEEN i LIKE. Ako je prvo polje koje se uspoređuje polje iz baze, a i ovisno o vrsti SQL elementa, tada se ovaj uvjet ugrađuje direktno u SQL element putem SQL generatora. Ako ne, onda se uvjet evaluira iza komunikacije s bazom na već dohvaćenim podacima i dodatno ih filtrira.

(2) Uvjeti klauzule

Uvjeti klauzule su uvjeti koji su predviđeni isključivo za ugradnju u SQL elemente i to u znakovnom obliku u skladu sa sintaksom SQL dijalekta ciljane relacijske baze podataka. Parametri uvjeta su tablica, tekst uvjeta i lista argumenata koja se primjenjuje na tekst, odnosno vezanih varijabli (eng. bind variables).

(3) Uvjeti s izrazom

Uvjeti s izrazom primjenjuju se isključivo na već dohvaćene podatke i ne ugrađuju se u SQL elemente. Značenje uvjeta je da ako slog zadovoljava uvjet i izraz vraća logičko "da" onda slog ostaje u spremniku, u protivnom se briše. Ovaj je način filtriranja obično sporiji nego onaj u SQL elementu jer se izbor željenih slogova ne događa na poslužitelju s bazom podataka, već se lokalno dohvate svi podaci i tek onda vrši odabir željenih slogova.

3.4.6. VEZA

Veza je element koji pobliže opisuje način povezivanja tablica u SQL SELECT naredbama. Ukoliko se ovaj element ne definira, onda se veza između tablica izvršava u WHERE klauzuli. Ukoliko se definira veza, mora se specificirati još i to da li je veza unutarnja (INNER JOIN klauzula SQL-a) ili lijeva-vanjska (LEFT OUTER JOIN klauzula SQL-a). Osim definiranja elementa veze, potrebno je još i povezati elemente uvjeta sa elementima veza.

3.5. POHRANA PODATAKA U BLOKU

Podaci su, zbog kompleksnosti mehanizama pohrane i čitanja, u bloku zapisani u zasebnom spremniku podataka, odnosno spremniku bloka.

3.5.1. SPREMNIK BLOKA

Bazne varijable zapisane su u dva asocijativna niza. Jedan niz služi za čuvanje vrijednosti baznih varijabli kakve su dohvaćene iz baze podataka. Drugi asocijativni niz služi za spremanje naknadno promijenjenih vrijednosti baznih polja, koje još nisu zapisane u bazu. Usporedbom tih dvaju nizova moguće je vidjeti koja su se polja mijenjala, te kakva im je originalna vrijednost. Ključevi ovih asocijativnih nizova su identifikatori redka i imena baznih varijabli.

Identifikator redka može biti i NULL vrijednost (konstanta NOVI_REDAK). To znači da je redak nov i još nezapisan u bazu, a identifikator će mu biti dodijeljen prilikom zapisivanja.

Osim konstante NOVI_REDAK, u upotrebi je još i konstanta NEMA_REDKA, a koristi se isključivo kod pokazivača na trenutni identifikator redka na nivu bloka, što znači da blok trenutno nije ni u jednom od postojećih redaka. Redci u spremniku ne mogu imati identifikator redka jednak konstanti NEMA_REDKA.

Osim samih vrijednosti, postoji još i redoslijedna lista identifikatora redaka, gdje su identifikatori poredani po redoslijedu željenog sortiranja podataka i kao takvi su prezentirani i korisniku. Svaka promjena kriterija sortiranja u bloku zahtijeva samo promjenu u ovoj redoslijednoj listi, dok sami podaci u spremniku ostaju nepromijenjeni.

Virtualne varijable bloka, kao i parametri bloka, zapisane su u asocijativnom nizu čiji je ključ ime varijable, jer se nalaze na nivou bloka. Virtualne varijable redka su zapisane u asocijativnom nizu čiji je ključ identifikator redka i ime varijable, jer se vrijednost čuva za svaki redak zasebno. Globalne varijable nisu zapisane u spremniku već se čitaju i pišu s nivoa aplikacije. Spremnik bloka zapisuje podatke u skladu s definicijom načina zapisivanja tipa podatka.

Osim pohrane podataka, spremnik može upravljati i podacima vezanim uz izvršavanje metoda unutar bloka, korištenjem spremnika poziva metoda, koji je sastavni dio spremnika podataka.

3.5.2. SPREMNIK POZIVA METODE

Spremnik poziva metode pomoćni je objekt spremnika podataka koji u sebi sadrži podatke potrebne za ispravno izvršavanje metode, poput trenutne instrukcije, povratne vrijednosti ili vrijednosti lokalnih varijabli. Svaki pojedini poziv metode ima svoj zasebni spremnik sa lokalnim podacima, čak i u slučaju da je riječ o istoj metodi. Na ovaj način realizirani su i rekurzivni pozivi metoda.

Spremnici se čuvaju u atributu liste lokalnih poziva spremnika podataka. Poredani su po redoslijedu pozivanja, pa je prvi element liste prva pozvana metoda, a posljednji posljednja. Završetkom izvršavanja, spremnik poziva metode se briše iz liste poziva metoda.

3.5.3. STATUS REDKA

Spremnik poziva metoda ima još jednu značajnu funkciju: izračun statusa redka prije poziva metode. S obzirom na to da se metode pozivaju implicitno prilikom aktiviranja događaja, javlja se potreba da se ocijeni status redka s obzirom na to treba li redak insertirati u bazu, izbrisati ga ili modificirati. SQL elementi koji obavljaju ovu zadaću pozivaju se eksplicitno ovisno o ocijenjenom statusu redka, pa se evaluacija potrebe zapisivanja sloga obavlja automatski prilikom pokretanja metode (ovo je moguće i isključiti, pa se onda evaluacija ne izvršava) i ne mijenja se tijekom njenog izvođenja. Rezultati te evaluacije pohranjeni su u spremniku metode, kao i mehanizmi koji tu evaluaciju obavljaju.

3.5.4. RAZINE POHRANE BAZNIH PODATAKA U SPREMNIKU

Bazni podaci se u spremniku pohranjuju na tri razine. Prva razina odnosi se na podatke dohvaćene iz baze podataka. Druga razina su promijenjeni bazni podaci. Ovi podaci još nisu zapisani u bazi podataka, a promjena se dogodila ili direktnim unosom od strane korisnika ili promjenom podataka od strane programske logike. Promijenjeni bazni podaci znače da je slog potrebno ažurirati u bazi podataka. Treća razina pamćenja odnosi se na rekalkulirane podatke. Rekalkulirani podaci ne utječu na status potrebe ažuriranja sloga u bazi podataka, jer služe za inicijalizaciju podataka, ili za rekalkulaciju kad je stvarno promijenjen neki drugi podatak.

Ne može istodobno postojati promijenjeni i rekalkulirani podatak jedne bazne varijable, već se međusobno isključuju.

Prioritet priikom čitanja neke bazne varijable je takav da se prvo čita promijenjeni podatak, ukoliko on ne postoji onda rekalkulirani, a ukoliko ne postoji ni rekalkulirani, onda se čita dohvaćeni.

3.5.5. TIPOVI PODATAKA

S obzirom na to da aplikacije usmjerene na baze podataka rade s tablicama u relacijskoj bazi čija su polja precizno definirana tipom podatka, te s druge strane, s obzirom na to da problemska područja koja takve aplikacije rješavaju obično zahtjevaju preciznu manipulaciju podacima (računovodstvene i financijske aplikacije, na primjer) onda je i u apstraktnom modelu potrebno jasno definirati tipove podataka. Tu ulogu preuzima element domene odnosno tipa podatka, zajedno sa svojim podvarijantama koje se odnose na osnovne tipove podataka.

Osnovni tipovi podataka apstraktnog modela su:

(1) Znakovni tip

(2) Numerički tip

(3) Logički tip

(4) Datumsko-vremenski tip

(5) Generički tip

U apstraktnom modelu podaci se zapisuju na tri mjesta: u polju tablice u bazi, u spremniku podataka u bloku te na kontrolama korisničkog sučelja. Isti podatak na ta tri mjesta može biti zapisan na različite načine. Logički podatak može u bazi biti zapisan sa 0 ili 1, u spremniku bloka sa logičkim vrijednostima Istina ili Laž, a na korisničkom sučelju sa znakovnim nizovima "Da" ili "Ne". Stoga je, za svaki tip podatka nužno definirati slijedeće metode konverzije:

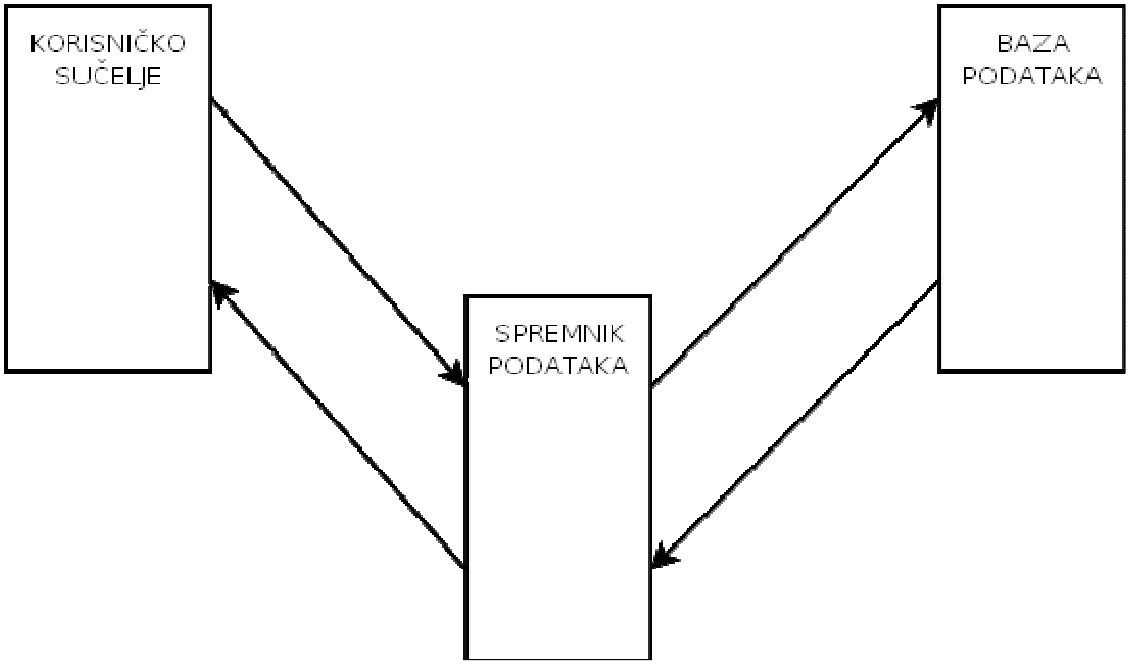
(1) Iz baze u spremnik bloka

(2) Iz spremnika bloka u bazu

(3) Iz spremnika bloka na korisničko sučelje

(4) Iz korisničkog sučelja u spremnik bloka

Spremnik bloka je srednji element u komunikaciji, te podaci ne putuju direktno iz baze podataka na sučelje i obrnuto (slika 3.5.1).



Slika 3.5.1 – Sučelje, spremnik i baza podataka, tijek podataka

Tablica 3.5.1 - Tipovi podataka u bazi podataka, spremniku i korisničkom sučelju

Rbr	Tip podatka	Zapis u bazi	Zapis u spremniku	Zapis u sučelju
1	Znakovni	Niz znakova	Niz znakova	Niz znakova
2	Numerički	Numerički	Numerički	Niz znakova
3	Logički	0 ili 1	Logički	"Da" ili "Ne"
4	Datumsko vremenski	Datumsko vremenski	Datumsko vremenski	Niz znakova
5	Generički	Nema	Objekt	Nema

Ovo vrijedi za znakovni, numerički, logički i datumsko-vremenski tip podatka ali ne i za generički tip. Generički tip podatka namijenjen je čuvanju podataka koji se ne bi trebali zaposivati u bazu podataka niti prikazivati na korisničkom sučelju, već su namijenjeni čuvanju objekata konkretne implementacije apstraktnog modela s ciljem mogućeg

referenciranja na iste, dakle interno za potrebe funkcioniranja samog bloka. U generičkom tipu mogu se čuvati neki od objekata korisničkog sučelja te direktno u izrazima zvati njihove atribute i metode i na taj način utjecati na ponašanje aplikacije. Moguće je zapisivati podatke generičkih tipova u bazu i prikazivati na sučelju, no onda za svaki pojedini slučaj za to treba napisati poseban programski kod (tzv. serijalizacija objekata).

Znakovni i numerički tip podatka imaju i precizno određenu duljinu. U slučaju znakovnog tipa duljina je maksimalan broj znakova koje polje može primiti, a u slučaju numeričkog tipa to je maksimalan broj znamenki. Numerički tip poznaje još i preciznost, a to je broj decimalnih mjesta.

Svi tipovi osim generičkog imaju i format prikaza (eng. picture). Format prikaza je znakovni niz koji pobliže opisuje kako se prikazuju podaci na korisničkom sučelju. Broj 12569.7 se može prikazati kao "12569.70", kao "12,569.70" ili na neki drugi način. Specifikacija formata prikaza nije dio definiranja apstraktnog modela, već ovisi o konkretnoj implementaciji, pa neće biti posebno opisivana.

3.6. KOMUNIKACIJA S BAZOM PODATAKA

Apstraktni model ima tri elementa koji su zaduženi za proces komunikacije s bazom podataka: element konekcije, SQL element i SQL generator.

3.6.1. KONEKCIJE

Relacijske baze podataka ostvaruju komunikaciju s vanjskim svijetom putem objekata konekcije. Objekti konekcije sadrže podatke o prijavi (korisničko ime, lozinka i poslužitelj), te se isključivo preko njih šalju SQL naredbe i primaju povratne informacije. One su isto tako zadužene za definiranje početka i kraja transakcije. Apstraktni model predviđa postojanje elementa konekcije koji na sebe preuzima ove zadaće.

Iako same konekcije mogu biti definirane na nivou aplikacije, ipak je nužno definirati s kojim konekcijama radi blok kako bi mogao upravljati transakcijama. To omogućuje jednom bloku istovremeni rad sa više otvorenih konekcija ali i različitih baza podataka istovremeno.

Konekcije u apstraktnom modelu preuzimaju na sebe i dodatne zadaće: na sebe vežu generatore SQL koda, koji definiciju SQL elementa apstraktnog modela pretvaraju u konkretan SQL kod koji se onda putem konekcije šalje bazi podataka.

S obzirom na međusobnu različitost konkretnih implementacija relacijskih baza podataka raznih proizvođača, nužno je da svaka od njih ima posebno prilagođen element konekcije.

3.6.2. SQL ELEMENTI

SQL elementi bloka imaju zadaće čitanja podataka iz baze i punjenja spremnika bloka, te pohranu podataka iz spremnika bloka u bazu podataka.

SQL elementi se pozivaju eksplicitno, na zahtjev, sastavni su dio bloka i unaprijed su definirani. Svaki SQL element unutar sebe sadrži SQL naredbu koja se šalje prema relacijskoj bazi. Iako se SQL naredbe u SQL elementima mogu zadavati i direktnim SQL-om, SQL naredbe nije potrebno tako pisati, već se konačan oblik formira iz ostalih elemenata bloka: tablica, polja, izraza i uvjeta, putem SQL generatora. Polja koja se navode u SQL elementima odgovaraju atributima polja koja određuju da li je polje dio selekcije, dodavanja ili izmjene podataka. Postoji više podvrsta SQL elemenata:

(1) SQL selekcija

SQL selekcija služi za dohvat kako veće količine podataka, tako i podataka jednog sloga putem SQL SELECT naredbe koja se šalje bazi podataka. Osnovna karakteristika SQL selekcije jest dinamička konstrukcija WHERE klauzule uzimajući u obzir elemente uvjeta u bloku s pripadajućim poljima. SQL selekcija može poslužiti i za dohvat podataka povezanih tablica.

(2) SQL selekcija sloga

SQL selekcija sloga služi isključivo za ponovno čitanje redka iz baze podataka (osvježavanje redka) i to na temelju identifikatora redka, ne uzimajući u obzir uvjete bloka. SQL selekcija sloga sadrži SQL SELECT naredbu.

(3) SQL zaključavanje

SQL zaključavanje slično je SQL selekciji s tom razlikom što generira i dodatnu FOR UPDATE klauzulu u SELECT naredbi, koja služi za zaključavanje dohvaćenih podataka. Ovo zaključavanje traje sve do završetka transakcije, čime se osigurava od promjene podataka od strane drugih korisnika.

(4) SQL zaključavanje sloga

SQL zaključavanje sloga djeluje slično SQL selekciji sloga, s tom razlikom što zaključava izabrani slog.

(5) SQL dodavanje sloga

SQL dodavanje sloga služi za pohranu još nezapisanog novog sloga u bazu podataka, tj. onog koji još nema identifikator redka i sadži SQL INSERT naredbu. Nakon izvršavanja red se označava kao zapisan i dobija svoj identifikator.

(6) SQL izmjena sloga

SQL izmjena sloga ažurira već postojeći slog u bazi novim sadržajem i sadži SQL UPDATE naredbu.

(7) SQL brisanje sloga

SQL brisanje sloga briše slog iz baze podataka korištenjem SQL DELETE naredbe, ali isto tako briše redak i iz spremnika podataka bloka.

3.6.3. SQL GENERATOR

Zadaća SQL generatora jest kreirati SQL naredbe koje će element konekcije slati bazi podataka. Kako je ovaj sustav zamišljen na ideji da mora moći komunicirati s više različitih relacijskih baza, te s obzirom na to da one komuniciraju različitim dijelektima SQL-a, to znači da svaka od tih relacijskih baza mora imati zasebni SQL generator.

Nakon što su definirani SQL elementi bloka, za svaki od njih se generira SQL naredba za komunikaciju s bazom podataka. U slučaju da se blok dinamički promijeni (na primjer da se doda novi korisnički definirani uvjet) SQL naredbe se mogu ponovno generirati kako bi se prilagodile nastalim promjenama i to kroz metode izgradnje.

SQL generator koristi atribute sql elementa koji sudjeluju u izgradnji i njih koristi kao polaznu osnovu za formiranje SQL naredbe. S obzirom na to da generiranje ovisi o ciljnoj bazi podataka, algoritam za taj proces neće biti opisivan.

3.7. METODE, INSTRUKCIJE I IZRAZI

Apstraktni model definira posebnu infrastrukturu za izvođenje programske logike unutar bloka. Infrastrukturu čine elementi metode, instrukcije te element izraza koji surađuju s elementima događaja.

3.7.1. METODA

Instrukcije su organizirane u metode, čija uloga odgovara pojmu funkcija odnosno procedura. Svaka instrukcija mora nužno pripadati nekoj od metoda u bloku. Element događaja mora imati pripadajuću metodu (koja može biti definirana i implicitno) koja sadrži programsku logiku. Metode imaju ulazne i izlazne parametre, te lokalne varijable.

Svaki poziv metode svara poseban prostor za pohranu privatnih podataka u spremniku bloka i tu se čuvaju vrijednosti parametara i lokalnih varijabli.

3.7.2. INSTRUKCIJA

Instrukcija kao element bloka, predstavlja najmanju izvršnu jedinicu programske logike bloka. Instrukcije su vezane na metodu, te su poredane slijedno jedna za drugom. Izvršavanje metode počinje prvom instrukcijom, a završava posljednjom instrukcijom ili instrukcijom koja eksplicitno označava kraj izvršavanja metode.

Postoje posebne instrukcije za iteraciju i selekciju (uvjetno izvršavanje), koje sadrže neki blok instrukcija. Postoji način za označavanje blokova instrukcija kako bi se izbjegla potreba za definiranjem zasebne metode i taj način označavanja odgovara ključnim riječima "begin" i "end" (ili sličnima) iz klasičnih programskih jezika. Svaka instrukcija ima brojčani podatak o svom nivou. Početna instrukcija u metodi ima nivo 0, a svaka slijedeća instrukcija isto. Ako se pojavi instrukcija iteracije, koja također ima nivo 0, tada blok instrukcija koji pripada toj iteraciji ima nivo 1, dakle prvi slijedeći. Blok instrukcija vrijedi sve dok se ponovno ne pojavi instrukcija s manjim nivoom.

Slijedi ilustracija sa selekcijom ako-onda-inače:

(1) Instrukcija (nivo 0)

(2) Instrukcija Ako-onda (nivo 0)

(3) Podinstrukcija (nivo 1)

(4) Podinstrukcija (nivo 1)

(5) Instrukcija inače (nivo 0)

(6) Podinstrukcija (nivo 1)

(7) Podinstrukcija (nivo 1)

(8) Instrukcija (nivo 0)

Instrukcije 3 i 4 čine zaseban blok instrukcija koji se izvršava ako je zadovoljen uvjet instrukcije 2. Ako taj uvjet nije zadovoljen tada se izvršava blok instrukcija 6 i 7 koji pripadaju instrukciji 5.

Svaka instrukcija ima atribut uvjetnog izvršavanja koji sadrži logičku konstantu "da" ili "ne" ili logički izraz. Prije svakog izvršenja instrukcije evaluira se uvjetni atribut i ako je rezultat evaluacije pozitivan instrukcija se izvršava, u protivnom ne.

Instrukcija ima više vrsta, te će ovdje biti pobrojane značajnije od njih. Mogu se definirati i druge instrukcije i dodati u apstraktni model ukoliko za to postoji potreba. Slijedi opis vrsta instrukcija zajedno sa pripadajućim specifičnim metodama.

(1) Instrukcija pridruživanja - ova instrukcija služi za pridruživanje vrijednosti argumenta (izraza ili varijable) nekoj varijabli. Na primjer, varijabli iznosa pridružuje se vrijednost izraza cijena*količina.

(2) Instrukcija evaluacije - ova instrukcija služi za izračun vrijednosti nekog izraza bez pridruživanja te vrijednosti polju. Istrukcija služi za izvršenje izraza u kojem se pojavljuje funkcija ili poziv nekog drugog mehanizma koji ostavlja posljedice samim svojim izvršavanjem, bez povratne vrijednosti.

(3) Instrukcija komentiranja - služi za zabilježbu vrijednosti polja i izraza tijekom izvršavanja radi kasnije analize. Ova instrukcija je korisna kada treba detaljno pratiti tijek rada i odrediti greške u izvršavanju metode. Vrijednosti se obično zapisuju u neku datoteku, ali moguća su i druga mjesta, odnosno drugi ciljni mediji.

(4) Instrukcija vanjskog poziva - ova instrukcija služi za poziv metode koja se nalazi izvan samog bloka, u nekom vanjskom modulu korisničkog sučelja ili sučelja za ispis na printer ili slično.

(5) Instrukcija "ako" - ovo je instrukcija uvjeta. Ako je uvjet zadovoljen izvršavaju se instrukcije u podbloku instrukcija koji slijedi, inače se izvršava prva slijedeća instrukcija istog nivoa.

(6) Instrukcija "inače" - ova instrukcija može slijediti "ako-onda" instrukciju. Izvršava se samo ako se pripadajuća "ako-onda" instrukcija nije izvršila. Ova instrukcija također na sebi može imati uvjet pa bi tada njezino značenje bilo "inače ako-onda". Takvih "inače" instrukcija može biti više u nizu, pa se onda svaka slijedeća može izvršiti samo ako se nije izvršila ni jedna prethodna.

(7) Instrukcija iteracije - ova instrukcija provjerava uvjet i ako je ispunjen izvršava blok instrukcija koji slijedi. Nakon izvršenja se ponovno provjerava uvjet i cijeli proces se ponavlja do trenutka dok uvjet nije zadovoljen, kada se izvršava prva slijedeća instrukcija iza pripadajućeg bloka instrukcija.

(8) SQL instrukcija - ova instrukcija predstavlja eksplicitni poziv nekog od SQL elemenata bloka. Kao argument navodi se o kojem je SQL elementu riječ.

(9) Instrukcija poziva bloka - ovom instrukcijom poziva se blok po imenu. Blok koji se poziva mora biti korijenski blok neke hijerarhije blokova.

(10) Instrukcija poziva podbloka - ovom instrukcijom poziva se neki od podređenih blokova unutar hijerarhije bloka. Mogu se pozivati samo direktno podređeni blokovi.

(11) Commit instrukcija - ovom instrukcijom eksplicitno se završava transakcija i konekciji se šalje COMMIT naredba čime se promjene unutar transakcije označavaju kao trajne.

(12) Rollback instrukcija - ova instrukcija eksplicitno šalje ROLLBACK naredbu konekciji, čime se završava transakcija, a promjene unutar transakcije označavaju kao netočne i radi se povratak stanja u bazi prije početka transakcije.

(13) Povratna instrukcija - s obzirom na to da element metode može vratiti povratnu vrijednost, definirsns je povratna instrukcija koja može prekinuti izvršenje metode i proslijediti povratnu vrijednost.

3.7.3. IZRAZI

Proces opisivanja programske logike bloka s ciljem ispunjavanja zadane funkcionalnosti aplikacije uključuje korištenje aritmetičkih i logičkih operacija nad vrijednostima varijabli, korištenje funkcija s parametrima i slično. Apstraktni model za tu svrhu ima predviđen element izraza.

Izraz se općenito može shvatiti kao znakovni podatak koji opisuje zadanu operaciju koristeći imena polja, logičke i aritmetičke operatore, konstante i pozive funkcija. Izračunavanjem, odnosno evaluacijom izraza dobiva se povratna vrijednost izraza. Povratnu vrijednost izraza moguće je zapisati u varijablu, iskoristiti za evaluaciju nekog uvjeta, koristiti kao parametar neke instrukcije ili kao atribut nekog od elemenata bloka. Isti izraz može se koristiti na više mjesta u bloku, jer je svaki izraz zasebni element, a svako korištenje izraza vrši se preko reference.

Primjer izraza je "IZNOS*TECAJ*PDV()" gdje su IZNOS i TECAJ polja, a PDV() funkcija (odnosno metoda) unutar bloka.

Izrazima je moguće eksplicitno definirati tip podatka koji vraćaju, što je važno za komunikaciju na relaciji baza podataka - spremnik bloka - korisničko sučelje.

Prilikom evaluacije izraza, izraz je prethodno potrebno parsirati, ali ovdje taj dio neće biti posebno opisivan jer ovisi o tehnologiji konkretne implementacije apstraktnog modela. Ukoliko se implementacija izvodi u nekom od dinamičkih skriptnih jezika, parsiranje može biti riješeno samom arhitekturom jezika. Statički jezici mogu imati zahtjevniju i kompleksniju metodu parsiranja.

3.8. DOGAĐAJI

Događajni sustav apstraktnog modela sastoji se od dvije važne komponente: elementa događaja i elementa rekalkulacije. Element događaja povezuje događaje unutar bloka sa programskom logikom, a element rekalkulacije određuje ovisnost vrijednosti varijabli na promjene vrijednosti drugih varijabli.

3.8.1. DOGAĐAJ

Logika izvršavanja bloka sama po sebi nije slijedna, već je vođena događajima. Blok reagira na događaje koji su izazvani korisnikovim korištenjem aplikacije. Zbog same prirode aplikacija koje rade s bazama podataka, nužno je da struktura događaja odražava strukturu dohvaćanja podataka, strukturu slogova i polja, te da bude ogranizirana na način koji će olakšati opravljanje blokom.

Događaj je element bloka koji ima definiranu metodu (implicitno ili eksplicitno jer više događaja može dijeliti istu metodu), te samim tim pridružen neki skup instrukcija.

Kad se pojavi zahtjev bloku za izvršenje neke vrste događaja, onda blok prvo mora pogledati da li postoji definiran, odnosno, registriran odgovarajući događaj sa pripadajućom logikom i ako postoji izvršava se metoda koja pripada događaju. Ukoliko takav događaj ne postoji zahtjev za izvršenjem se ignorira. Događaji se registriraju registracijskim ključem koji u sebi sadrži tip događaja i neki drugi podatak ako je nužan, ime varijable na primjer.

Povratna vrijednost metode događaja važna je za izvršavanje bloka, jer povratnom vrijednošću događaj informira blok o tome da li je izvršenje uspješno ili ne. Tako metoda događaja "događaj poslije redka" mora vratiti logičku vrijednost "da" da bi unos podataka mogao prijeći u drugi red. U suprotnom, to znači da unos redka nije ispravan i ne dozvoljava se prelazak u slijedeći red.

Apstraktni model predviđa više tipova događaja koji se međusobno razlikuju po registracijskom ključu:

(1) Događaj opisa bloka

Događaj opisa bloka jest prvo što se izvršava kod bloka i služi za definiranje početnih objekata, import vanjskih modula i njihova inicijalizacija i slično. Ovdje se definiraju objekti korisničkog sučelja. Ovaj se događaj poziva samo jednom na početku izvršavanja bloka.

Registracijski ključ: tip događaja .

(2) Događaj dohvata na bloku

Događaj dohvata na bloku služi za početni dohvat podataka jednom selekcijom iz baze, a u skladu sa definiranim uvjetima na bloku. Tipično se radi o punjenju podataka iz glavne tablice. Izvršava se jednom za cijeli blok. Ponovno izvršenje se može dogoditi na zahtjev.

Registracijski ključ: tip događaja .

(3) Događaj dohvata na redku

Događaj dohvata na redku služi za dohvat podataka iz povezanih tablica i izvršava se za svaki redak u spremniku podataka posebno. Može služiti i za popunjavanje podacima virtualnih polja.

Registracijski ključ: tip događaja.

(4) Događaj prije bloka

Događaj prije bloka se izvršava nakon događaja dohvata na bloku i redku i izvršava se jednom kod poziva bloka. S obzirom da se događa nakon dohvata, manipulira sa već dohvaćenim podacima.

Registracijski ključ: tip događaja .

(5) Događaj poslije bloka

Događaj poslije bloka izvršava se prilikom izlaska iz bloka. On je bitan jer se u njemu može spriječiti izlazak iz bloka ukoliko je to potrebno.

Registracijski ključ: tip događaja .

(6) Događaj prije redka

Događaj prije redka događa se prilikom pozicioniranja na novi redak. Također se događa prilikom pozicioniranja na prvi redak nakon događaja prije bloka.

Registracijski ključ: tip događaja .

(7) Događaj poslije redka

Događaj poslije redka događa se prilikom izlazka iz jednog redka i pozicioniranja na drugi redak, te prilikom pokušaja izlaska iz bloka jer se prije izlaska iz bloka mora zapisati sadržaj redka. Ovaj događaj služi za izvršavanje SQL elemenata koje mijenjaju podatke u bazi, te se postavljaju provjere ispravnosti redka, pa je moguće spriječiti izlazak iz redka i njegovo spremanje u bazu podataka ukoliko svi podaci u redku nisu ispravni, putem povratne vrijednosti metode.

Registracijski ključ: tip događaja .

(8) Događaj prije varijable

Ovaj događaj se odvija prije ulaska u varijablu. Pod pojmom ulaska u varijablu podrazumijeva se pozicioniranje na varijablu bilo preko korisničkog sučelja, bilo nekom naredbom. Registracijski ključ sadrži dodatak varijeble jer se ovaj događaj registrira posebno za svaku varijablu.

Registracijski ključ: tip događaja + varijabla.

(9) Događaj poslije varijable

Ovaj događaj se odvija prije izlazka iz varijable i ima sličnu ulogu kao i događaj poslije redka, jer također može spriječiti izlazak iz varijable ukoliko vrijednost varijable nije ispravna.

Registracijski ključ: tip događaja + varijabla.

(10) Događaj akcije na varijabli

Ovaj dagađaj se izvršava na korisnički zahtjev iz korisničkog sučelja. Tipičan primjer za to je slučaj tipke na korisničkom sučelju koja je povezana s nekom varijablom. Pritiskom na tipku aktivira se događaj na akcije dotičnoj varijabli.

Registracijski ključ: tip događaja + varijabla.

(11) Vanjski događaj

Vanjski događaj omogućuje vanjskim modulima definiranje vlastitih događaja u bloku, koji se pozivaju pod svojim imenom. Ako korisničko sučelje ima potrebu za događajem koji bi se izvršio prilikom minimizacije forme onda se takav događaj (pod unaprijed definiranim imenom u modulu korisničkog sučelja) može povezati s blokom. Vanjski događaj predstavlja infrastrukturu za proširenje događajnog sustava apstraktnog modela u konkretnim implementacijama.

Registracijski ključ: tip događaja + vanjski modul + vanjski događaj + dodatni parametri.

(12) Događaj promjene vrijednosti varijable

Događaj promjene vrijednosti varijable izvršava se prilikom promjene vrijednosti varijable nekom instrukcijom ili promjenom putem korisničkog sučelja. Ukoliko se promjena događa kao posljedica korisničkog upisa, onda se ovaj događaj odvija nakon događaja poslije varijable.

Registracijski ključ: tip događaja + varijabla.

(13) Događaj greške

Ako se prilikom izvršavanja bloka dogodi greška, poziva se ovaj događaj. Greške mogu biti od grešaka iz baze podataka, do korisničkih grešaka koje su isprogramirane logikom bloka.

Registracijski ključ: tip događaja.

(14) Događaj izlaska iz bloka

Ovaj događaj izvršava se kod zahtjeva za izlaskom iz bloka putem akcije na korisničkom sučelju (kad korisnik želi zatvoriti formu unosa) i zapravo služi kao provjera da li blok smije biti završen. Blok će biti završen ukoliko metoda događaja vrati logičku vrijednost da. Izvršava se prije lanca događaja varijabla - redak - blok.

Registracijski ključ: tip događaja .

3.8.2. REKALKULACIJA

Uporaba aplikacija usmjerenih na baze podataka zahtijeva automatizirano izračunavanje nekih vrijednosti koje su ovisne o promjenama drugih vrijednosti. Ako postoje tri varijable, količina, cijena i iznos, one se nalaze u međusobnoj ovisnosti. Promjena cijene ili količine ima za posljedicu promjenu iznosa, dok promjena iznosa može utjecati na, primjerice, količinu. Ovakve međuovisnosti poznate su i iz proračunskih tablica (eng. spreadsheet) gdje se ostvaruju formulama. Apstraktni model u tu svrhu predviđa element rekalkulacije.

Algoritam izračunavanja rekalkulacija sastavni je dio elementa spremnika podataka, a element rekalkulacije služi kao nositelj podataka, odnosno opisa rekalkulacije. Rekalkulacija povezuje ciljnu varijablu čija se vrijednost mijenja sa elementima koji mijenjaju tu vrijednost.

Rekalkulacija se odvija u dva osnovna slučaja:

(1) Tijekom inicijalizacije - prilikom dodavanja novog redka javlja se potreba postavljanja inicijalnih vrijednosti.

(2) Promjenom vrijednosti neke druge varijable - ako se promjeni vrijednost neke druge varijable (izvorna varijabla), potrebno je promijeniti vrijednost ciljne varijable.

Bitno je istaknuti da se rekalkulacija odvija slično valovima: rekalkulacija vrijednosti jedne varijable može pobuditi rekalkulaciju neke druge varijable. Tako ciljna varijabla može postati izvorna za neku drugu rekalkulaciju.

3.9. IZVRŠNI CIKLUS BLOKA

Izvršni ciklus bloka predstavlja samo srce apstraktnog modela. Izvršni ciklus upravlja aktivacijom odgovarajućih događaja u bloku, te upravlja zahtjevima za preuzimanjem fokusa elemenata korisničkog sučelja. Drugim riječima, određuje da li se i kada smije ući ili napustiti blok, redak ili pojedino polje.

Pravilno upravljanje događajima i fokusima u aplikacijama usmjerenim na baze podataka je od ključne važnosti. Princip konzistentnosti podataka zahtjeva da se svaka promjena redka mora ažurirati u bazi podataka prilikom napuštanja tog redka. Neažuriranje podataka prilikom izlaska iz redka dovelo bi do nekonzistentnosti podataka na korisničkom sučelju s onima u bazi podataka i dovelo u pitanje svrhovitost aplikacije.

Kada korisnik preko korisničkog sučelja postavi zahtjev za premještanjem fokusa u drugo polje ili redak, ili zatvaranje bloka, mora se bezuvjetno aktivirati ispravan niz (lanac) događaja koji će izvršiti predviđenu programsku logiku kao i ažuriranje podataka u relacijskoj bazi i eventualno zabraniti promjenu fokusa korisniku. Ovo je važno kad podaci u polju ili redku nisu ispravni i ne smiju biti zapisani u bazu podataka. Fokus tada mora biti vraćen na neispravan redak ili polje i zahtijevati od korisnika ispravan unos, ili odustajanje od promjene podataka.

Događaji koji sudjeluju u lancu su:

(Ab) Događaj prije bloka

(Ar) Događaj prije redka

(Av) Događaj prije varijable

(Bv) Događaj poslije varijable

(Br) Događaj poslije redka

(Bb) Događaj poslije bloka

Događaji su navedeni po redoslijedu izvođenja. U lancu se događaji odvijaju na bloku, redku i varijabli. Razlikuju se slijedeće situacije:

(S1) Ulazak u blok - prilikom ulaska u blok određuje se i redak, te varijabla koja dobija fokus. Redoslijed događaja je Ab,Ar,Av.

(S2) Promjena redka i fokusa varijable - iz redka r1 i varijable v1 fokus se prebacuje na varijablu v2 i redak r2. Redoslijed događaja je Bv(v1), Br(r1), Ar(r2), Av(v2).

(S3) Promjena fokusa varijable na istom redku - u redku r mijenja se fokus iz varijable v1 i varijablu v2. Redoslijed događaja je Bv(v1),Av(v2).

(S4) Izlazak iz bloka - izlazi se iz bloka, ali i redka i varijable. Redoslijed događaja je Bv,Br,Bb.

Potrebno je napomenuti da događaji mogu vratiti i informaciju o neuspjehu pa se tada lanac događaja prekida i fokus se ne prenosi dalje. Ako u situaciji S2 događaj Br(r1) zaključi da unos redka nije ispravan, fokus ostaje na redku r1 i varijabli v1.

Fokus se mijenja akcijama korisnika na korisničkom sučelju. Korisničko sučelje daje bloku zahtjev za promjenu fokusa, blok izvršava predviđeni lanac događaja i pridruženu programsku logiku, te vraća informaciju korisničkom sučelju o novom fokusu. Za komunikaciju s blokom koristi slijedeće metode koje su direktno izazvane korisničkim akcijama:

(a) Obradi_akciju - aktivira događaj akcije na varijabli (recimo pritiskom na neku unaprijed određenu tipku itd)

(b) Obradi_odustajanje - izvršava odustajanje od promjena podataka u tekućem redku i vraća ih na vrijednosti prije promjene

(c) Obradi_brisanje - izvršava brisanje redka i mijenja fokus na drugi redak (situacija S2)

(d) Obradi_umetanje - izvršava umetanje novog redka i mijenja fokus na taj redak (situacija S2)

(e) Obradi_fokus - izvršava promjenu fokusa (situacije S1,S2 i S3)

(f) Zatvori_blok - izvršava zatvaranje bloka odnosno izlazak (situacija S4)

Metoda obradi_fokus zaključuje da li je riječ o ulasku u blok, pokušaju izlaska iz njega ili redovnoj promjeni fukusa. U skladu s tim poziva jednu od tri metode:

(e1) Fokus_ulazak - ova se metoda poziva prilikom prvog ulaska u blok i pridodjeljuje se prvi fokus. Obrađuje situaciju S1.

(e2) Fokus_promjena - ova se metoda poziva prilikom promjene fokusa varijable ili redka. Obrađuje situacije S2 i S3.

(e3) Fokus_izlazak - ova se metoda poziva prilikom svakog pokušaja izlaska iz bloka (ne znači da mora uspjeti). Obrađuje situaciju S4.

3.10. DINAMIČKO KREIRANJE I PROMJENA BLOKA TIJEKOM IZVRŠAVANJA APLIKACIJE

Iako se korisnički zahtjevi konstantno mijenjaju, kako zbog promjene okruženja u kojima organizacija djeluje, tako i njenog organizacijskog ustroja, aplikacije usmjerene na baze podataka obično imaju fiksno određenu strukturu, ali gledano u nekom trenutku. To bi značilo

da je aplikaciju moguće graditi na temeljima konačnog skupa blokova sa unaprijed definiranim elementima tablica i polja.

U praksi, međutim, postoje situacije kada blok i njegove pripadajuće elemente treba izgraditi dinamički s obzirom na neku prethodnu korisničku akciju, ili treba izvršiti neku promjenu trenutnom bloku. Apstraktni model stoga mora omogućiti dinamičko kreiranje i promjenu bloka tijekom izvršavanja aplikacije.

Princip za dinamičko kreiranje ili promjenu je taj da u procesu uvijek sudjeluju dva bloka:

(a) BLOK IZVRŠITELJ - blok koji izvršava promjenu, odnosno onaj koji sadrži logiku promjene.

(b) BLOK PRIMATELJ - blok nad kojim se vrši promjena.

Naredbe za izvršenje dinamičkih operacija nad blokovima mogu se podijeliti u četiri grupe:

(1) Naredbe za ispitivanje postojeće strukture bloka

(2) Naredbe za kreiranje novih elemenata u bloku

(3) Naredbe za izmjenu postojećih elemenata u bloku

(4) Naredbe za brisanje elemenata iz bloka

Dvije su osnovne vrste dimanmičkih operacija nad blokovima:

(1) Kreiranje novog bloka

Kod kreiranja novog bloka blok izvršitelj kreira potpuno novi blok i izvršava ga. Nakon izvršavanja bloka primatelja, kontrola se vraća ponovo bloku izvršitelju, koji onda može ponovo kreirati novi blok ili završiti. Tijek kontrole je slijedeći:

Izvršitelj -> Primatelj -> Izvršitelj

(2) Izmjena postojećeg bloka

Izmjena postojećeg bloka odnosi se na situacije kad blok primatelj već postoji i ima kontrolu, ali na korisnički zahtjev je potrebno izmijeniti nešto u funkcioniranju bloka. Tu se može raditi o izmjeni kriterija filtriranja (korisnički filteri), redoslijedu sortiranja ili, recimo, skrivanju nekih kolona unutar bloka primatelja. Na korisnički zahtjev se aktivira blok izvršitelj koji na primatelju obavlja tražene izmjene i vraća kontrolu izvršenja izmijenjenom primatelju. Tijek kontrole izgleda ovako:

Primatelj -> Izvršitelj -> Primatelj

4. KORISNIČKO SUČELJE

Korisničko sučelje je komunikacijski kanal kojim se vrši komunikacija između korisnika i aplikacije i predstavlja jedan od najvažnijih čimbenika koji određuju konačnu kvalitetu aplikacije. Ta kvaliteta može biti objektivna i odnositi se na skup operacija koje korisnik može izvršiti s aplikacijom, kao i na skup informacija koje od aplikacije može dobiti. Postoji i subjektivna kvaliteta, a ona se odnosi na "lakoću" i "efikasnost" kojom korisnik može postići svoje ciljeve u radu s aplikacijom. Značaj koji korisničko sučelje kao mjera kvalitete same aplikacije ima danas, najbolje se može vidjeti po tome koliko velike kompanije poput Applea i Microsofta ulažu napora u razvoj grafičkog korisničkog sučelja za svoje proizvode, posebno pazeći da ono što oni zovu korisnički doživljaj (eng. user expirience), to jest osjećaj koji korinik ima kad radi s aplikacijom, bude što bolji.

Kvaliteta korisničkog sučelja može pozitivno ili negativno utjecati na uspješnost projekta. Frustrirani korisnici mogu odbiti koristiti aplikaciju ili još gore, aktivno ometati projekt, dok zadovoljni korisnici mogu svojim prijedlozima pomoći u daljnjem razvoju aplikacije, te u organizaciji širiti pozitivno ozračje u svezi s aplikacijom.

Izvori korisničke frustracije mogu biti brojni. Nezadovoljstvo može izazvati nekonzistentnost ponašanja korisničkog sučelja u različitim dijelovima aplikacije, poput primjerice, pokretanja ispisa tipkom u jednom, a menijem u drugom dijelu aplikacije. Nezadovoljstvo može izazvati netransparentnost u ponašanju aplikacije, ako korisnik nije u stanju shvatiti što se treba slijedeće dogoditi nakon neke pokrenute akcije. Ukoliko ne može završiti neku akciju (recimo zbog nekog neunesenog podatka) a nema informaciju o tome što nije ispravno, to će biti negativno iskustvo.

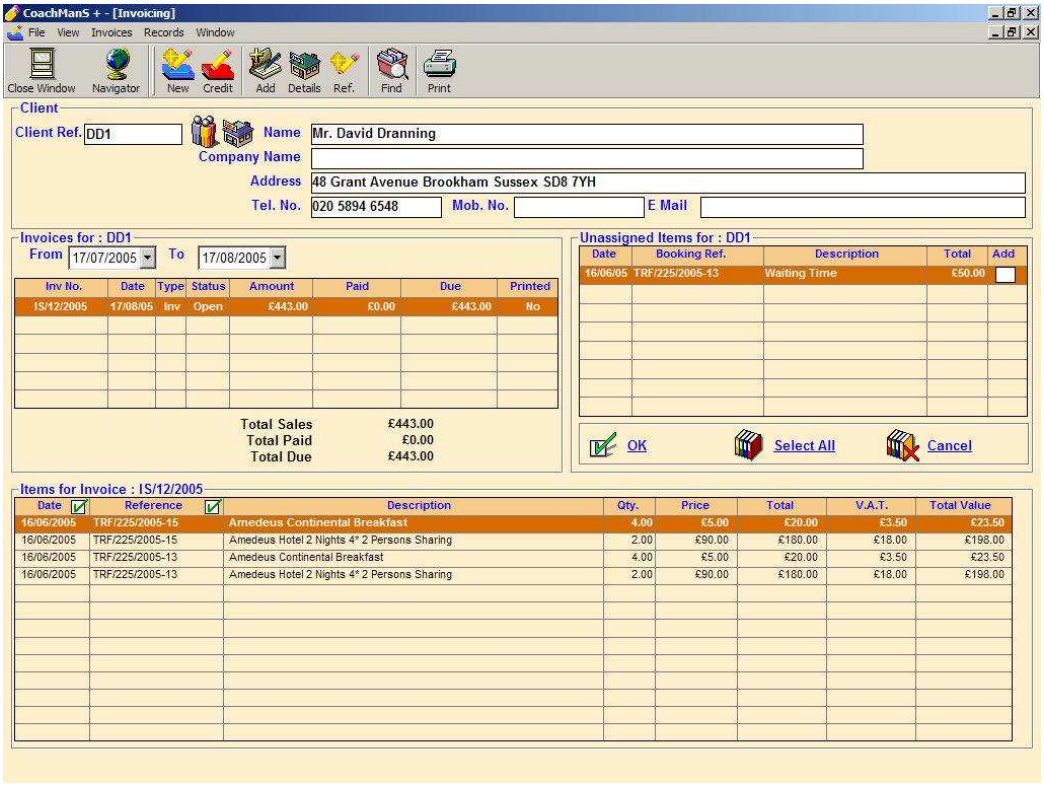
S druge strane, ukoliko korisničko sučelje izgleda poznato i razumljivo čak i u dijelovima aplikacije u kojima se korisnik prvi puta nalazi, ukoliko unos podataka teče neometano i glatko i korisnik može na jednostavan način dobiti sve informacije, tada će iskustvo biti pozitivno.

Postoje dvije različite vrste korisničkog pristupa aplikaciji, pa stoga i pristup izgradnji korisničkog sučelja mora odgovarati tom pristupu. Prvi pristup odnosi se na korisnike koji

redovito rade s aplikacijom i kojima je potrebno da na brz i pristupačan način mogu obaviti česte zadaće, makar to značilo zapamtiti neke prečice. Drugi pristup odnosi se na povremene korisnike koji se ne sjećaju svih detalja od posljednjeg boravka u aplikaciji i koji se moraju prisjećati načina na koji su nešto obavili, a zadaća aplikacije je pomoći im u tome. Korisničko sučelje može se graditi s idejom da se pokriju obje vrste pristupa aplikaciji, ali moguće je i odlučiti se samo za jedan pristup ukoliko će najveći broj korisnika na taj način koristiti aplikaciju.

Aplikacije usmjerene na bazu podataka specifične su zbog masovnog višekorisničkog unosa podataka, kao i zahtjevnog izvještavanja, pa korisničko sučelje mora biti prilagođeno tim potrebama. Posebno treba napomenuti da postoji sukob između potrebe da aplikacija kontrolira konzistentnost i ispravnost unosa podataka s jedne, te potrebe da korisnik brzo i jednostavno unosi podatke s druge strane. Korisničko sučelje aplikacija usmjerenih na bazu podataka mora pronaći kompromis između te dvije krajnjosti.

U ovom poglavlju bit će opisani preduvjeti za izgradnju korisničkog sučelja za aplikacije usmjerene na bazu u svjetlu apstraktnog modela, te način na koji se korisničko sučelje može povezati sa apstraktnim modelom.



Slika 4.1 – Primjer korisničkog sučelja jedne aplikacije usmjerene na bazu podataka

4.1. PREDUVJETI ZA REALIZACIJU KORISNIČKOG SUČELJA

Razina složenosti razvoja aplokativnog softvera značajno se povećala zbog popularizacije grafičkog korisničkog sučelja koje je razvoju aplikacija nametnulo nove standarde. Pojavom grafičkog okruženja, mogućnosti korisničkog sučelja su se drastično povećale i uvelike olakšale korištenje računala prosječnom korisniku. No s druge strane, razvojni timovi su se suočili sa složenijom infrastrukturom za izradu korisničkog sučelja ugrađenom u operativni sustav, nego što je to bio slučaj ranije.

Stoga su se pojavile specijalizirane programske biblioteke (obično objektno orjentirane) s ciljem olakšavanja razvoja grafičkog korisničkog sučelja, a također treba spomenuti i pojavu vizualnih alata za dizajniranje korisničkog sučelja koji omogućuju slaganje sučelja metodom povuci-i-ispusti (eng. drag and drop). S obzirom na povećanje efikasnosti razvoja grafičkog korisničkog sučelja korištenjem specijaliziranih programskih biblioteka, ne treba čuditi da se većina današnjeg softvera temelji na njihovu korištenju. Konkretna implementacija ovog

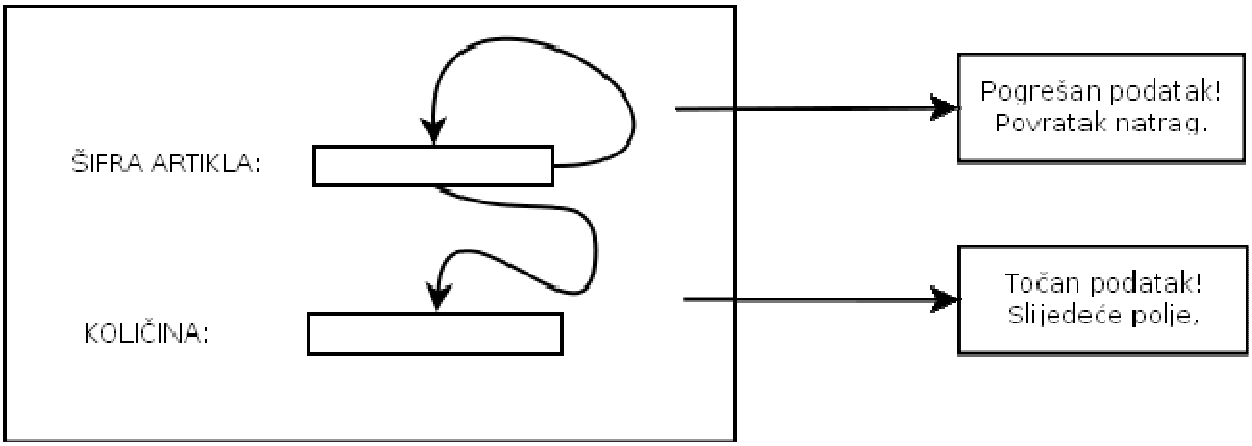
sustava za razvoj aplikacija usmjerenih na baze podataka, posljedično, također mora koristiti ove programske biblioteke za realizaciju korisničkog sučelja.

No, ove biblioteke su predviđene pisanju aplikacija opće namjene, a da bi se mogle koristiti u sustavu za razvoj aplikacija temeljenom na apstraktnom modelu moraju biti u stanju u potpunosti podržati potrebe izvršnog ciklusa bloka, a to znači da moraju zadovoljiti slijedeće preduvjete:

(1) Aktivno upravljanje fokusom

Najznačajnija osobina koju biblioteka korisničkog sučelja mora imati, a da bi se mogla razmatrati kao kandidat za implementaciju ovog sustava jest način na koji upravlja fokusom. Pod fokusom se podrazumijeva informacija o tome koji element korisničkog sučelja je trenutno aktivan i gdje se može vršiti unos podataka. Ukoliko na nekoj formi treba unijeti informaciju o šifri artikla, tada korisnik ulazi u polje za unos (čime to polje dobija fokus) i počinje unositi traženu informaciju. Ukoliko iza toga treba unijeti količinu, tada se fokus premješta na polje za količinu i proces unosa ide dalje.

Pod aktivnim upravljenjem fokusom podrazumijeva se programsko određivanje fokusa u nekom trenutku. Obično se fokus mijenja korisničkom akcijom, no u aplikacijama usmjerenim na baze podataka nužno je ponekad zadržati fokus u polju zbog neispravnog unosa, ili programski odrediti fokus na neko drugo polje pod određenim uvjetima, a sve to mimo volje korisnika. Isto tako, ukoliko je riječ u o unosu podataka u sučelje u obliku tablice, gdje svaki redak tablice predstavlja redak u tablici baze podataka, ponekad je važno ne dozvoliti prelazak fokusa u drugi red dok tekući red nije unešen ispravno, u skladu s programskom logikom aplikacije. Programska biblioteka za izradu korisničkog sučelja koja ne može upravljati fukusom na taj način nije pogodna za implementaciju apstraktnog modela.

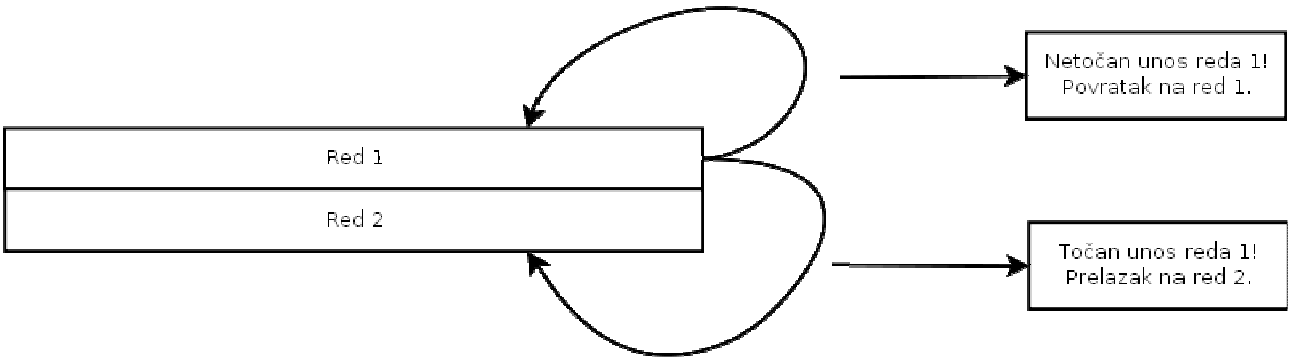


Slika 4.1.1 – Primjer prelaska fokusa sa šifre artikla na količinu sa povratkom fokusa i kontrolom

(2) Osjetljivost na promjene fokusa

Osim programskog određivanja fokusa, bitna je i osjetljivost na promjene fokusa, pri čemu se misli na reakciju promjene fokusa nekom programskom logikom. Ukoliko je korisnik unio šifru artikla i mišem pokušao promijeniti fokus u polje unosa količine, tada sustav treba taj pokušaj promjene fokusa registrirati. Ukoliko postoji programska logika koja prilikom izlaska iz polja šifre artikla provjerava da li takva šifra postoji u bazi podataka i ne smije pustiti unos podataka dalje ukoliko ne postoji, tada će se takva programska logika moći ispravno izvršiti samo ukoliko sustav pravodobno registrira sve promjene fokusa. Kod tabličnog unosa podataka, prelazak u novi red treba odmah zapisati podatke iz prethodnog reda u bazu, a to ne bi bilo moguće ukoliko sustav na vrijeme ne može raspoznati da je fokus prešao u idući red.

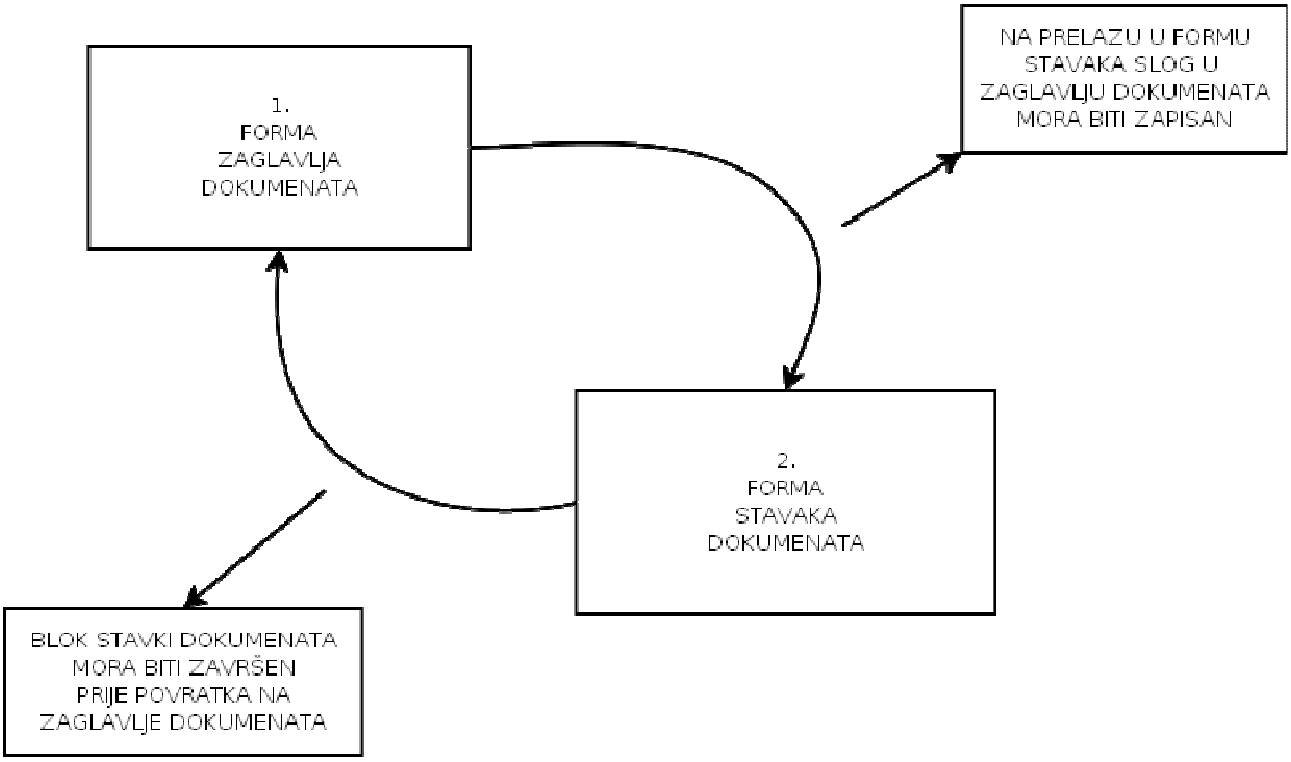
Ova je osobina programske biblioteke korisničkog sučelja nužna kako bi se mogli realizirati događaji iz poglavlja o izvršnom ciklusu bloka i lancu događaja prije i poslije bloka, redka i polja.



Slika 4.1.2 – Primjer s tabličim unosom (prelazak u novi red)

(3) Modalnost formi

Konzistentnost unosa podataka na različitim formama koje su u hijerarhijskom odnosu zahtijeva da se iz podređene forme ne smije izaći na nadređenu prije nego što se dovrši unos podataka. Stoga je potrebno proglasiti formu modalnom, odnosno reći programskoj biblioteci korisničkog sučelja da fokus ne smije izaći iz forme i prijeći na drugu formu, osim u slučaju da se forma zatvori, to jest unos završi. U protivnom, moglo bi doći do konfliktnih situacija u vezi sa sučeljem i komunikacijom s bazom, narušio bi se lanac događaja opisan u poglavlju o izvršnom ciklusu bloka, jer bi se dozvolio izlaz iz bloka bez da se završe svi događaji na njemu.



Slika 4.1.3 – Modalnost formi – primjer dokumenta i stavaka

Izvršni ciklus bloka zahtijeva strogu kontrolu nad događajima programske biblioteke korisničkog sučelja, posebice onih koji se tiču upravljanja fokusom. Kako bi se olakšala implementacija korisničkog sučelja, ali i kasnije korištenje u procesu razvoja aplikacije, nužno je izvršiti značajnu prilagodbu programske biblioteke potrebama apstraktnog modela. S obzirom na to da je većina takvih programskih biblioteka napisana u objektno-orjentiranim jezicima, prirodno je usmjeriti takvu prilagodbu u pravcu razvoja novih klasa (odnosno elemenata) koje koriste originalne klase iz programske biblioteke, te preuzimaju na sebe zadaću ispunjavanja preduvjeta za korištenje u okviru izvršnog ciklusa bloka. Slijedi tekst koji pobliže opisuje takve elemente.

4.2. PANEL

Za ispravno izvršavanje lanca događaja koji se odnosi na izvršni ciklus bloka, nužno je okupiti sve pojedinačne elemente korisničkog sučelja u jednu cjelinu, to jest postaviti krovni element koji će koordinirati njihov rad. Takav element zove se panel.

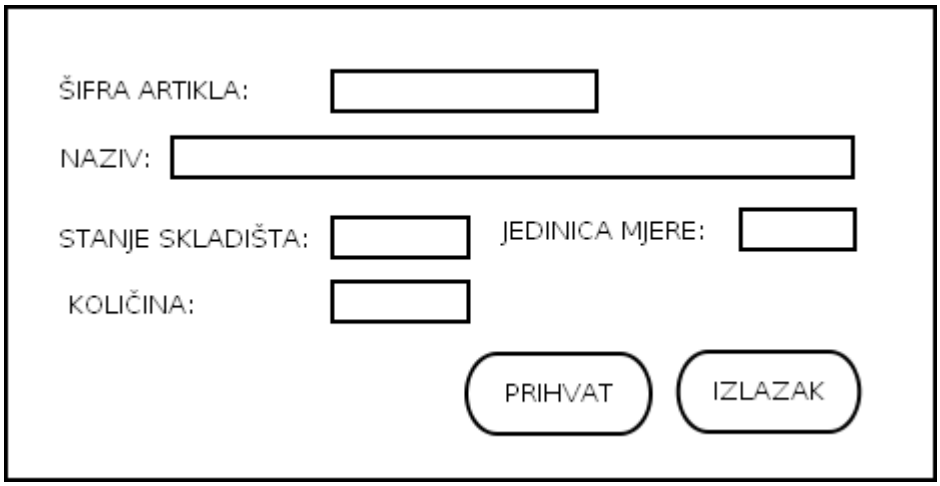
Pojedini elementi korisničkog sučelja raspoređuju se po panelu i tvore događajnu cjelinu. Raspored elemenata se može vršiti na jedan od slijedećih načina:

(1) Ručni raspored

Raspored pojedinih elemenata može se vršiti ručno, u smislu da član razvojnog tima detaljno specificira vizualni raspored i veličinu pojedinih elemenata. Takvo raspoređivanje može se izvoditi direktnim unošenjem koordinata pozicije elemenata, a može se koristiti i neki od alata za vizualno dizajniranje korisničkog sučelja.

Prednost takvog pristupa je puna kontrola nad izgledom korisničkog sučelja. Na taj je način primjerice moguće napraviti panel za unos dokumenta koja vizualno u potpunosti odgovara izgledu dokumenta na papiru. Također, moguće je napraviti ergonomski i vizualno ugodnija korisnička sučelja.

Nedostatak se nalazi u činjenici da je na dizajn potrebno utrošiti dodatno vrijeme što poskupljuje razvoj, a također je potrebno uložiti i dodatni napor za usklađivanje vizualnog doživljaja formi koje rade različiti članovi razvojnog tima jer je čest slučaj da pojedini dijelovi aplikacije različito izgledaju zbog različitosti stilova izrade korisničkog sučelja. Problemi nastaju i kod prilagodbe panela zaslonima različite razlučivosti, te promjeni veličine panela, gdje je potrebno dinamički premještati i mijenjati veličinu pojedinih elemenata kako bi se panel prilagodio svojoj novoj veličini.



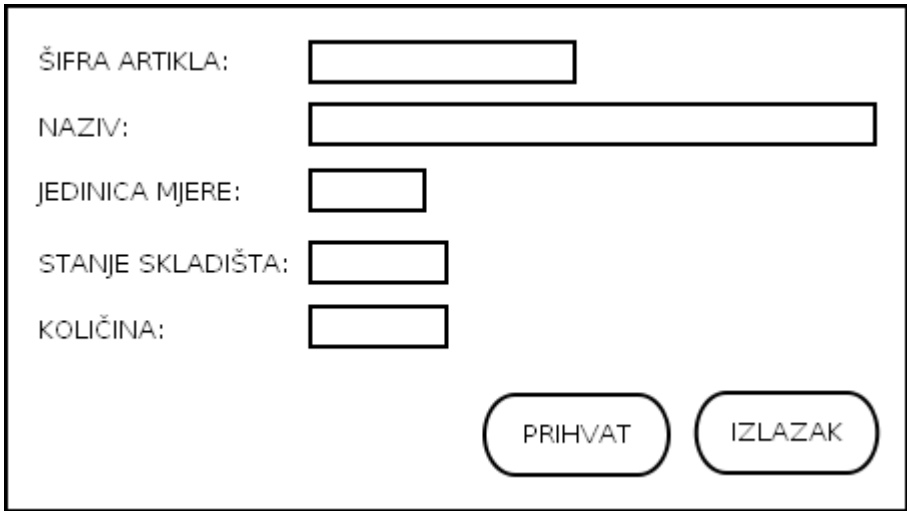
Slika 4.2.1 – Primjer panela nastalog ručnim rasporedom

(2) Automatizirani raspored

Automatizirani raspored elemenata pretpostavlja korištenje nekog od automatiziranih mehanizama, takozvanih upravljača izgledom (eng. layout manager) koji po nekom unaprijed utvđenom algoritmu raspoređuje elemente po panelu, a isto tako se i brine o promjeni pozicije ili veličine elemenata pri promjeni veličine panela. Moguće je koristiti više različitih upravljača sa različitim stilovima raspoređivanja, već prema potrebi pojedinog panela.

Prednost automatiziranog rasporeda leži u uniformnosti korisničkog sučelja cijele aplikacije, te u smanjenju troškova razvoja korisničkog sučelja.

Nedostatak pristupa nalazi se u izostanku pune kontrole nad izgledom korisničkog sučelja, te u činjenici da se može primijetiti da je korisničko sučelje napravljeno automatski, no to je već subjektivni dojam.



4.2.2 – Primjer panela nastalog automatskim rasporedom

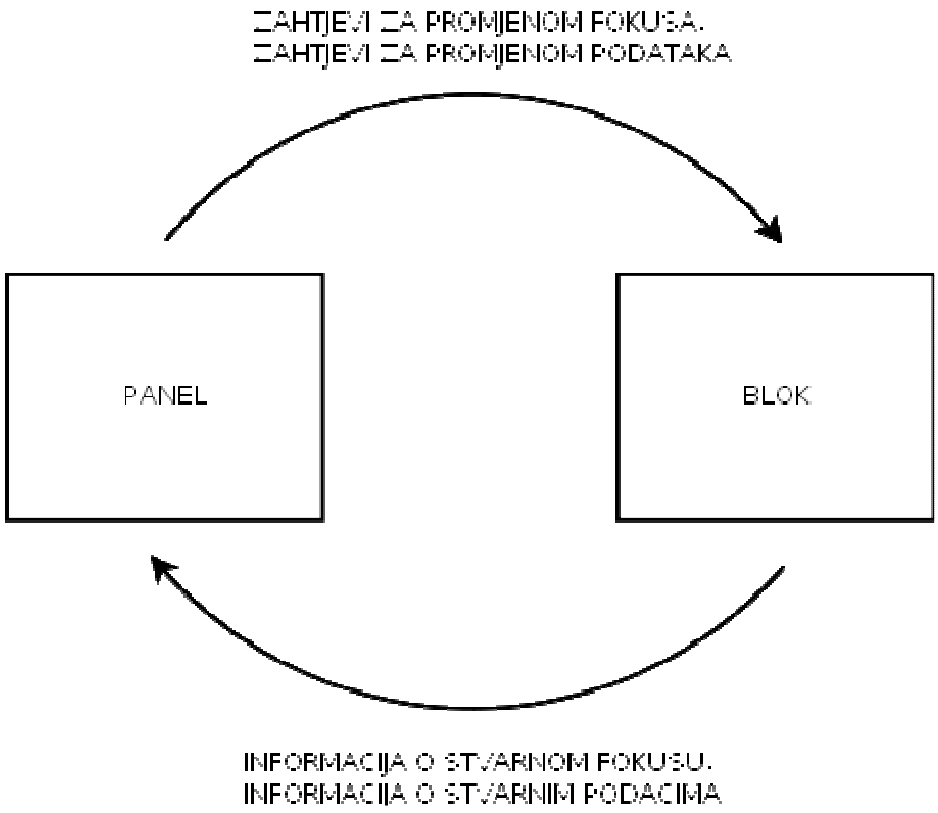
4.3. ODNOS PANELA, BLOKA I FORME

Osim zadaće objedinjavanja pojedinačnih elemenata korisničkog sučelja, panel ima zadaću komunicirati s blokom, usklađivati događaje bloka sa događajima korisničkog sučelja, te uskladiti sadržaj pojedinačnih elemenata korisničkog sučelja sa spremnikom podataka u bloku.

Općenito vrijedi pravilo da svaki blok ima jedan i samo jedan pripadajući panel, i obratno, svaki panel ima jedan i samo jedan pripadajući blok. Pojedinačni elementi korisničkog sučelja na panelu moraju odgovarati elementima varijabli iz bloka, iako sve varijable ne moraju biti na panelu, mogu postojati skrivene varijable koje su dio bloka, ali se ne prikazuju na korisničkom sučelju.

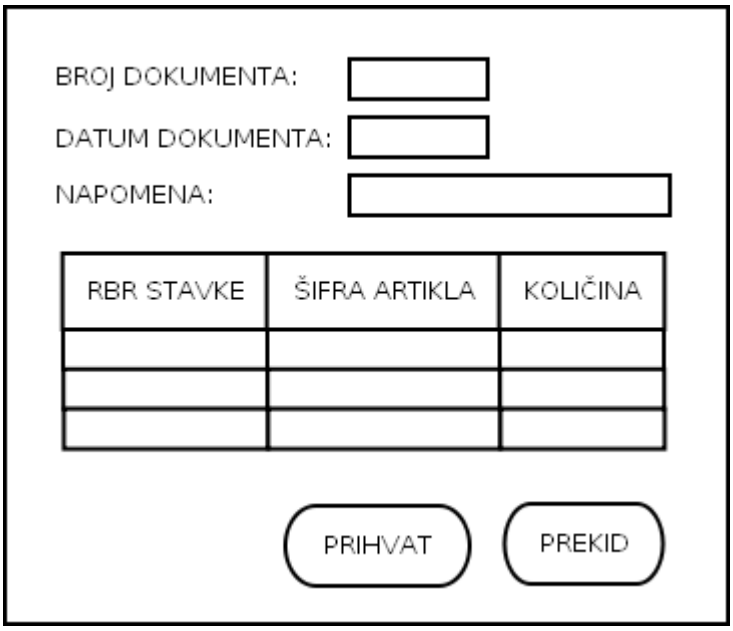
Komunikacija između panela i bloka treba se odvijati preko standardiziranog sučelja (eng. interface) kako bi se paneli jedne programske biblioteke korisničkog sučelja mogli što lakše zamijeniti panelima druge programske biblioteke korisničkog sučelja. Ovo je važno zbog prenosivosti aplikacija na različite sustave (Windows, Mac OS X, Linux, Android, iPhone OS i slično).

Panel preuzima zahtjev korisnika za unosom polja, novim fokusom i slično, te taj zahtjev i podatke prosljeđuje bloku na obradu. Blok obrađuje zahtjev kroz lanac događaja, odnosno izvršni ciklus bloka, te vraća povratnu informaciju o tome gdje je zapravo fokus, te koji je sadržaj elemenata bloka. Može se dogoditi da korisnički zahtjev za promjenom fokusa nije odobren od strane bloka zbog neispravnog unosa, te blok javlja da se fokus ne smije promijeniti. Panel bezuvjetno mora izvršiti naloge bloka.

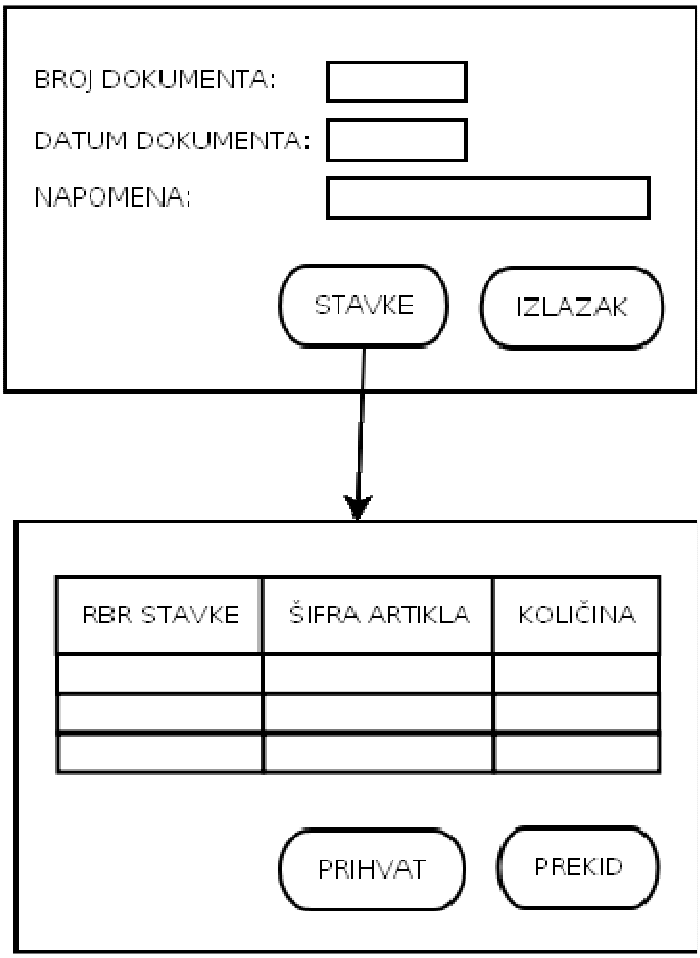


Slika 4.3.1 – Komunikacija panela i bloka

Paneli su, osim blokovima, s druge strane pridruženi i formama, odnosno prozorima grafičkog sučelja. Vizualni identitet formi jako ovisi o konkretnom operativnom sustavu na kojem se izvršavaju, pa neće ovdje biti detaljnije opisivani. Na jednoj formi najčešće se nalazi samo jedan panel, a onda samim tim i blok. Ali ne mora uvijek biti tako. Na jednoj formi se mogu nalaziti dva i više panela (roditelj-dijete odnos), ali onda treba strogo paziti na skokove fokusa među panelima, ti skokovi moraju biti u skladu sa izvršnim ciklusom blokova. Ilustracija ovoga mogao bi biti primjer sa zaglavljem dokumenta i pripadajućim stavkama, gdje postoje dva bloka, jedan za zaglavlje i drugi za stavke, te pripadajuća dva panela. Ukoliko svaki panel ima zasebnu formu, tada bi postojale dvije forme, jedna sa panelom zaglavlja i druga, podređena, sa panelom stavki. No moguće je i oba panela staviti na jednu formu, gdje bi gornju polovicu forme zauzimao panel zaglavlja, a donju polovicu panel stavki.



Slika 4.3.1a – Primjer zaglavlja i stavki s jednom formom



Slika 4.3.1b – Primjer zaglavlja i stavki s dvije forme

4.4. ELEMENTI KORISNIČKOG SUČELJA

Elementi korisničkog sučelja koji se raspoređuju po panelima, moraju odgovarati varijablama pripadajućeg bloka i služe prikazu njihovog sadržaja korisniku i korisničkim zahtjevima za promjenom sadržaja. Postoje dvije osnovne grupe elemenata korisničkog sučelja: jednostavni i složeni.

4.4.1. JEDNOSTAVNI ELEMENTI

Jednostavni elementi korisničkog sučelja su oni elementi koji se odnose samo na jednu varijablu u bloku. To znači da u jednom trenutku pokazuju sadržaj samo jedne varijable ili samo jedan podatak. Jednostavni elementi direktno su vezani na pripadajuće varijable u bloku, pa svaka promjena sadržaja u jednostavnom elementu rezultira zahtjevom za

promjenom sadržaja varijable u bloku. Svaka promjena sadržaja varijable u bloku rezultira promjenom sadržaja jednostavnog elementa na panelu. Sadržaj varijable odnosi se na sadržaje varijabli u trenutnom redku, jer jednostavni elementi ne mogu prikazati sadržaje varijabli iz više redaka podataka.

Programske biblioteke danas imaju veliki broj različitih elemenata koji mogu poslužiti kao jednostavni elementi, no ovdje će biti predstavljena samo dva osnovna.

(1) Unosno polje

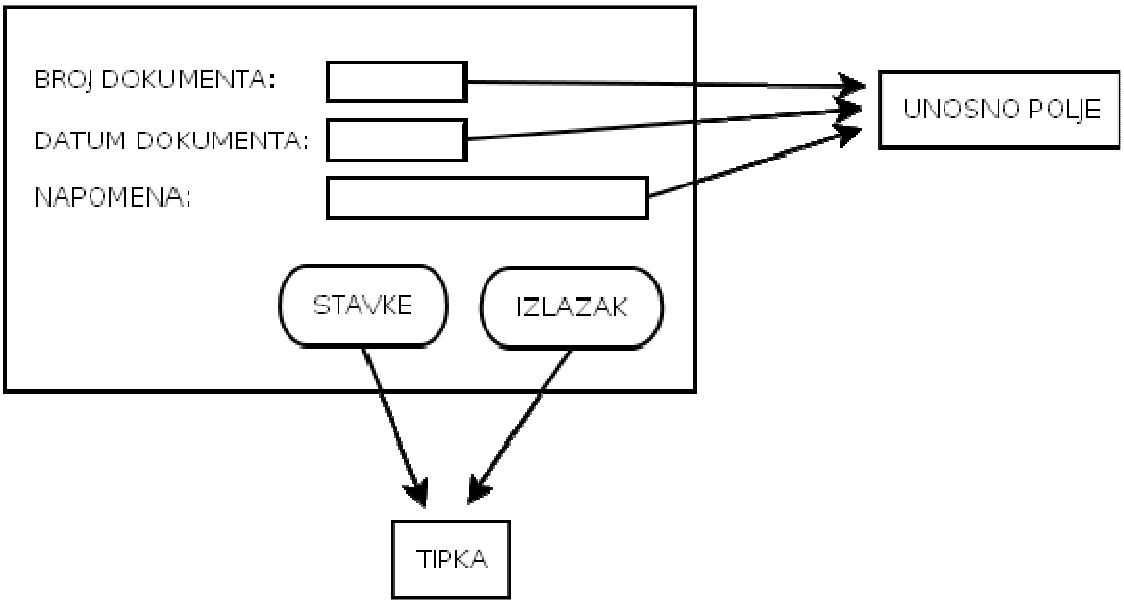
Unosno polje (eng. edit control) zapravo je običan pravokutnik koji može preuzeti fokus i u koji se pokazivačem teksta (eng. cursor) može unijeti neka vrijednost - tekstualna, numerička, datumska ili kakva druga. Unosno polje često se pojavljuje u korisničkim sučeljima.

Uz unosno polje obično stoji i labela koja pojašnjava o kakvom je podatku riječ. Ta se labela može smatrati dijelom elementa unosnog polja, a može se smatrati i samostalnim elementom.

(2) Tipka

Tipka (eng. button) kao element korisničkog sučelja služi za pokretanje neke akcije. Aktiviranjem tipke povezane s varijablom bloka, aktivira se događaj akcije na varijabli. Tipka također predstavlja jedan od najčešće korištenih elemenata korisničkog sučelja.

Ovo su osnovni jednostavni elementi uz koje je moguće prikazati podatke jednog redka i upravljati aplikacijom. Iako korisnička sučelja imaju još brojne takve jednostavne elemente, oni su zapravo samo posebni slučajevi unosnog polja i tipke. U konkretnim implementacijama ovog sustava, mogu se koristiti i drugi jednostavni elementi na panelu.



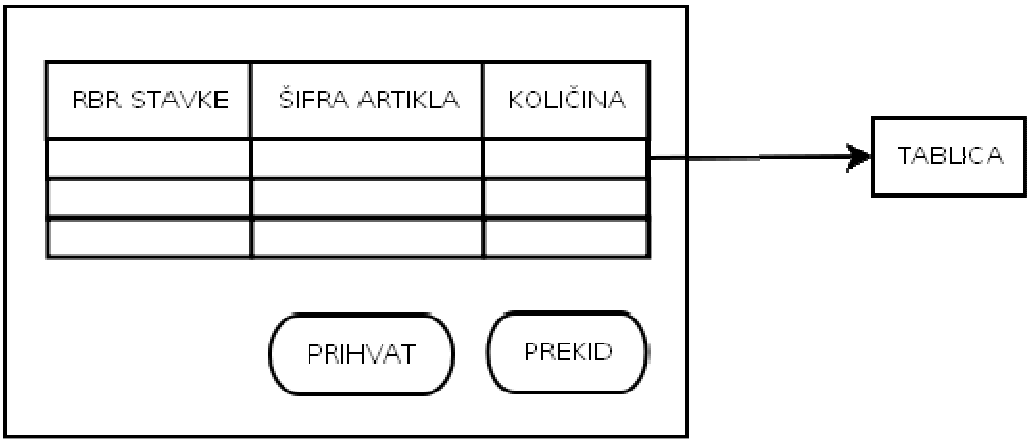
Slika 4.4.1 – Jednostavni elementi

4.4.2. SLOŽENI ELEMENTI

Složeni elementi korisničkog sučelja služe za istovremeni prikaz i unos podataka više varijabli i više redaka istovremeno. Za razliku od jednostavnih elemenata, programske biblioteke korisničkih sučelja ne obiluju ovakvim elementima, a njihova kvaliteta varira. Događajni sustav takvih elemenata također često predstavlja problem, pa im treba pristupiti vrlo oprezno tijekom implementacije. Ovdje će biti predstavljena tri osnovna složena elementa.

(1) Tablica

Tablica je najjednostavniji element iz ove kategorije i predstavlja direktnu i lako razumljivu reprezentaciju podataka u bloku. Kolone tablice odnose se na varijable u bloku, a redovi tablice na redke podataka. Na vrhu tablice je redak sa labelama pojedinih kolona koje pobliže opisuju podatak koji se nalazi u koloni. Ćelije tablica su unosna polja i odnose se na jednu varijablu u jednom redku. Promjenom fokusa unutar tablice pobuđuju se odgovarajući događaji iz lanca događaja i izvršnog ciklusa bloka, jer je promjenom fokusa moguće istovremeno otići u drugu varijablu u drugom redku. Trenutni redak bloka treba biti posebno označen u tablici. Ovo je najčešće korišteni složeni element.



Slika 4.4.2.1 – Primjer tablice

(2) Matrica

Matrica je, u odnosu na tablicu, nešto složeniji element korisničkog sučelja. U matrici i redovi i kolone predstavljaju kombinacije vrijednosti varijabli iz bloka, a pojedinačne ćelije su zapravo agregirani podaci neke treće (numeričke) varijable.

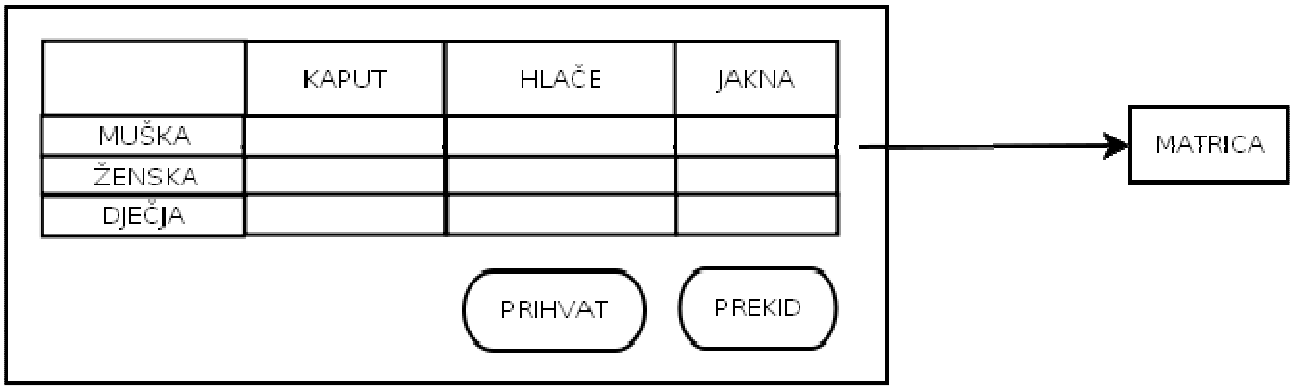
Ako se blok sastoji od tri varijable: županije, proizvođača i prometa, znači da se svaki redak tablice odnosi na promet određenog proizvođača u određenoj županiji. U tom slučaju matrica može biti oblikovana na način da su u kolonama pobrojani pojedini proizvođači, u redcima pojedine županije, a u ćelijama su sumirani prometi odgovarajućih županija i proizvođača.

Agregatne funkcije osim sumiranja mogu biti i prosječna vrijednost, maksimalna vrijednost, minimalna vrijednost, pobrojavanje i slično.

Matrice mogu imati i nekoliko razina podataka u redovima i kolonama (na primjer godine i unutar njih dvanaest mjeseci) posloženih hijerarhijski, što dodatno usložnjava element.

Matrice se najčešće koriste za prikaz podataka (u izvještajnim i OLAP sustavima), ali moguće ih je koristiti i za unos podataka, ali u tom slučaju problem može predstavljati raspisivanje jedne vrijednosti ćelije na varijablu u više pripadajućih redaka jer podatke treba raspodijeliti po nekom ključu. Ovo je posebno slučaj u aplikacijama za planiranje i budžetiranje. Najjednostavnije rješenje jest u obliku bloka konstruiranog u tijeku izvršavanja aplikacije koji

ima tri varijable a u podacima selekciju kombinacija vrijednosti dvije varijable (eng. distinct select) koje se smještaju u kolone i redke matrice, gdje svaka kombinacija ima jedan redak podataka u bloku. Događaj poslije redka u bloku tada treba umjesto jednostavnih SQL naredbi INSERT, UPDATE, DELETE izvršiti složenije razbijanje podatka, možda pozivom podbloka ili procedure u bazi podataka.



Slika 4.4.2.2 – Primjer matrice

(3) Grafički model

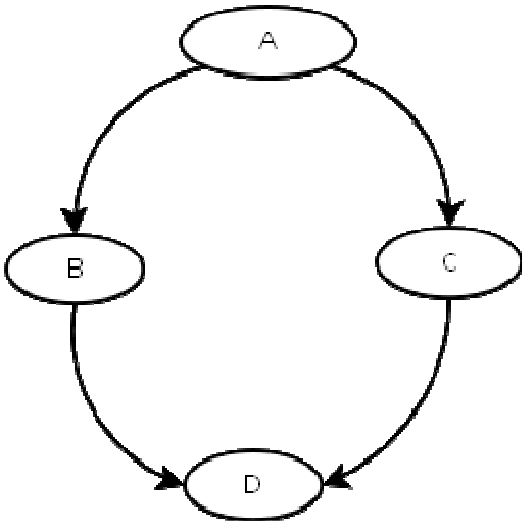
Postoji velika potreba za grafičkim modeliranjem procesa i organizacijskih struktura, posebice u uvjetima multikorisničkog kolaboracijskog pristupa. Element korisničkog sučelja koji bi bio u stanju prikazati i mijenjati takve modele može stoga imati veliku primjenu.

Grafički model kao element korisničkog sučelja temelji se na ideji da se sam model mora nalaziti u bazi podataka. Ako je model definiran preko čvorova i njihovih međusobnih veza, tada se i čvorovi i veze moraju nalaziti u istoj tablici. Element grafičkog sučelja mora biti u stanju interpretirati zapis iz tablice modela, s tim da se u samoj tablici moraju nalaziti svi podaci potrebni za grafički prikaz (čvor ili veza, koordinate, boje, tekstovi i slično).

Upravljanje grafičkim modelom treba biti identično kao i upravljanje elementom tablice. Dodavanje novog čvora ili veze trebalo bi dodavati novi red u tablicu modela, aktivirajući pri tom sve događaje iz lanca događaja i izvršnog ciklusa bloka. Pomicanje čvorova i veza te njihovo brisanje svodi se na izmjenu podataka u redku, te brisanje redka.

Zaključavanjem slogova u tablici modela može se postići efekt zaključavanja pojedinih dijelova modela od strane jednog korisnika, koristeći transakcijski mehanizam baze podataka.

Teško je osmisliti jedan univerzalni grafički model koji bi mogao pokriti sve potrebe, pa je vjerojatniji scenarij u kojem postoji više grafičkih modela za različite namjene, a tablice modela u bazi su im u potpunosti prilagođene.



ID	VRSTA	PARAMETRI	TEKST
1	Čvor	(x1,y1)	A
2	Čvor	(x2,y2)	B
3	Čvor	(x3,y3)	C
4	Čvor	(x4,y4)	D
5	Veza	(1-2)	
6	Veza	(1-3)	
7	Veza	(2-4)	
8	Veza	(3-4)	

Slika 4.4.2.3 – Primjer grafičkog modela i vrijednosti u tablici

4.5. INFRASTRUKTURA KORISNIČKOG SUČELJA

Pod infrastrukturom korisničkog sučelja podrazumijeva se funkcionalnost aplikacije koja je prisutna u svakom trenutku u svakom bloku i predstavlja stalno dostupnu pomoć korisniku. Infrastruktura treba biti integralni dio razvojnog okruženja i njeno uključivanje u aplikaciju treba biti implicitno, odnosno podrazumjevano. Zadaće infrastrukture mogu biti korisničko filtriranje, sortiranje, eksportiranje i importiranje, te ispis podataka, bilježenje korisničkih akcija i drugo. Također, infrastruktura bi trebala biti nadogradiva, na način da razvojni tim ima mogućnost izmjene postojeće funkcionalnosti ili nadogradnje infrastrukture novom funkcionalnošću.

Funkcionalnost infrastrukture treba se aktivirati pritiskom na neku tipku, ikonu ili iz menija otvarajući novi blok. U novootvorenom bloku korisnik specificira dodatnu akciju koja treba biti odrađena.

4.5.1. KORISNIČKO FILTRIRANJE PODATAKA

Velike količine podataka s kojima se suočavaju korisnici aplikacija usmjerenih na bazu podataka stvaraju potrebu za infrastrukturom filtriranja koja pomaže korisnicima izdvojiti potrebne podatke po kriterijima koji se određuju tijekom izvođenja aplikacije u skladu s trenutnim potrebama.

Apstraktni model pruža podršku mogućnosti korisničkog filtriranja sposobnošću dinamičke izmjene bloka kao i korištenjem elemenata uvjeta. Elementi uvjeta mogu biti korisnički ili ugrađeni. Korisnički uvjeti su oni koje zadaje korisnik tijekom korištenja aplikacije, a ugrađeni su uvjeti oni koje je u aplikaciju ugradio razvojni tim i koji su neophodni za ispravan rad aplikacije pa ih korisnik ne može mijenjati.

Pozivom korisničkog filtriranja otvara se forma za unos korisničkih filtera. Ona može biti lista elemenata uvjeta gdje korisnih navodi uvjete i njihove parametre ili tekstualno polje u kojem korisnik navodi uvjete poštujući određenu specificiranu sintaksu. Ovaj unos uvjeta može dolaziti u puno različitih oblika, pa ga ovdje nije nužno detaljno specificirati.

Nakon korisničkog određivanja uvjeta, blok iz kojeg je pozvano filtriranje i na koji se filtriranje odnosi tada treba biti promijenjen na način da se navedeni korisnički uvjeti pretvore u elemente uvjeta i ugrade u blok, te da se SQL elementi ponovo izgrade kako bi SQL

naredbe dobile nove WHERE klauzule, a nakon toga bi se mora izvršiti događaj dohvaćanja podataka bloka, čime bi se stari sadržaj spremnika bloka zamijenio s novim, koji je u skladu s novim korisničkim filterom.

4.5.2. KORISNIČKO SORTIRANJE PODATAKA

Osim filtriranja podataka, postoji potreba i za sortiranjem podataka po pojedinim poljima, bilo da je riječ o silaznom ili uzlaznom redoslijedu. Stavke nekog dokumenta, na primjer, mogu biti sortirane po količini od stavki s najvećim količinama prema stavkama s najmanjim količinama.

Apstraktni model ima mogućnost određivanja elemenata varijabli kao dijelova ORDER BY klauzula, nakon čega je potrebno ponovo izgraditi SQL elemente, te pokrenuti izvođenje događaja dohvaćanja podataka bloka.

Korisnik može uvjete sortiranja unijeti kao listu varijabli pored kojih stoje oznake rastućeg ili opadajućeg redoslijeda i prioriteta unutar klauzule sortiranja.

4.5.3. KORISNIČKO EKSPORTIRANJE PODATAKA

Poduzeća često imaju nekoliko različitih informacijskih sustava, koji ne dijele isti podatkovni prostor, ali moraju međusobno razmjenjivati podatke. Isto tako, nije rijedak slučaj da poduzeće mora dio svojih podataka poslati nekom od vanjskih partnera, bilo povremeno, bilo redovito. Korisnici imaju preferenciju koristiti neke od proračunskih tablica poput Excela ili Open Office Calc-a kako bi mogli dodatno obrađivati i prezentirati podatke, jer im ti programi omogućuju velike mogućnosti izmjene i formatiranja, a vrlo malo kontrole podataka.

Stoga je nužno omogućiti podršku za eksportiranja podataka u različite formate, poput tekstualnih, CSV, XML, XLS i slično. Korisnih pri tome treba odrediti kolone koje se eksportiraju, izlazni format podataka, te ime datoteke koja će biti rezultat eksporta.

Proces eksporta treba biti baziran na čitanju spremnika podataka bloka koji se eksportira, te spremanju tih podataka u ciljnu datoteku u odgovarajućem formatu.

4.5.4. KORISNIČKO IMPORTIRANJE PODATAKA

Razmjena podataka između različitih sustava može rezultirati potrebom za importiranjem podataka. Korisnici mogu sami pripremiti neke podatke u proračunskim tablicama, poput šifarnika partnera ili nekih podataka izračunatih vanjskim pomoćnim programom i takve podatke trebaju importirati u aplikaciju, a mogu dobiti podatke i od vanjskog partnera.

Ugrađena mogućnost importa može riješiti takve potrebe, a realizacija se svodi na proces kojim se simulira ručni korisnički unos podataka umetanjem podataka iz izvora za import, redak po redak, pri čemu podaci moraju proći provjeru koju i inače prolaze prilikom regularnog unosa podataka.

4.5.5. KORISNIČKI ISPIS PODATAKA

U svakodnevnom korištenju aplikacije javlja se potreba za brzim ad-hoc ispisom koji se ne nalazi u skupu standardnih izvještaja ili ispisa. Ugrađena mogućnost ispisa omogućuje ispisivanje takvih povremenih ispisa bez potrebe da se podaci prvo eksportiraju u vanjsku datoteku, a potom ispisuju, ili potrebe da razvojni tim vrši izmjenu aplikacije.

Prilikom ispisa korisnik mora izabrati polja koja se ispisuju, broj kopija, stil ispisa te ciljni printer. Kao i kod eksporta, podaci se mogu dohvatiti direktnim čitanjem spremnika podataka bloka.

4.5.6. BILJEŽENJE KORISNIČKIH AKCIJA

Informacijski sustavi se neprestano mijenjaju, nadograđuju i prilagođavaju novim potrebama. Tijekom životnog ciklusa prolaze kroz brojna stabilna, ali i nestabilna stanja, te se događa da u nekim specifičnim situacijama ne funkcioniraju ispravno, a ponekad se zbog vanjskih utjecaja dogode nepredviđene situacije koje rezultiraju pogreškama.

Traženje uzroka pogrešaka proces je u kojem se razvojni timovi i korisnici često nalaze na suprotnim stranama, te iskazuju međusobno neslaganje oko uzroka koji su rezultirali pogreškama. Da bi se eliminirala učestalost ovakvih sukoba, moguće je napraviti automatski sustav bilježenja korisničkih akcija poput unosa u polja, promjena fokusa, aktiviranja tipaka i slično, a usporedo s time i praćenje reakcija aplikacije. Ovakvo bilježenje može se vršiti na

svakom bloku u aplikaciji, bez posebne intervencije razvojnog time, kao ugrađena mogućnost. Prednosti takvog bilježenja su slijedeće:

- Jednoznačno utvrđivanje mjesta i uzroka greške, čime se eliminira eventualni razlog sukoba među zainteresiranim stranama

- S obzirom da se jasno vidi tijek unosa podataka, postoji mogućnost da se greška može namjerno ponoviti, što je jedan od preduvjeta za uspješno otklanjanje grešaka

- Analizom automatskih bilješki mogu se utvrditi korisničke navike prilikom unosa podataka, što može biti korisno za daljnje unaprjeđenje aplikacije

4.5.6. OSTALO

Osim navedenih ugrađenih mogućnosti, moguće su i druge, poput automatskog dokumentiranja aplikacije, SQL konzola koje bi bile u istoj transakciji s blokom koji se trenutno izvršava, pregleda stanja svih elemenata u bloku i slično.

5. REPOZITORIJ

Razvoj aplikacijskih sustava često se vrši na način da se izvorni kod aplikacije podijeli u manje cjeline koje se čuvaju u odvojenim datotekama, pri čemu je izvorni kod napisan u tekstualnom obliku, poštujući sintaksu ciljnog programskog jezika. U slučaju timskog razvoja, izvorni kod može se pohraniti na CVS serveru gdje je moguć višekorisnički razvoj aplikacije. Izvorni kod se po potrebi pretvara u izvršni (najčešće u obliku izvršne datoteke) te se distribuira krajnjim korisnicima ili postavlja na određeno mjesto na produkcijskom serveru. Implementacija apstraktnog modela može podržati i takav klasičan način razvoja.

Relacijska baza podataka, sa svojim transakcijskim sustavom, zaključavanjem podataka, mogućnošću postavljanja upita i ažuriranja podataka, predstavlja izuzetno efikasan način spremanja velike količine podataka. S obzirom na aspekt timskog razvoja aplikacije, ali i potrebom za istodobnim pristupom aplikaciji velikog broja korisnika, znajući da relacijska baza u tom smislu prednjači nad datotečnim sustavom, poželjno je da se izvorni i izvršni kod čuvaju u relacijskoj bazi.

Apstraktni model temelji se na elementima koji se mogu jednostavno modelirati kao tablice u relacijskoj bazi, nema potrebe za razlikovanjem izvornog i izvršnog koda, već se oni ujedinjuju u jedan kod. Takve rezultirajuće tablice moguće je organizirati u repozitorij, te napraviti posebnu aplikaciju usmjerenu na baze podataka, koja služi za unos i pregled takvih tablica, odnosno za razvoj aplikacija. U relacijskoj bazi podataka moguće je smjestiti repozitorij s listama elemenata apstraktnog sustava koji bi bili smješteni u tablicama relacijske baze podataka.

U slijedećem tekstu razrađen je jedan od mogućih sustava repozitorija, što ne znači da je jedini moguć. Naprotiv, ideja repozitorija znači da se takav sustav može proširivati dodatnim podacima koji su potrebni razvojnim timovima.

5.1. LISTA APLIKACIJA

Repozitorij mora imati mogućnost istovremene pohrane više od jedne aplikacije, što znači da mora postojati lista aplikacija, u kojoj svaki od redaka predstavlja jednu aplikaciju. Redcima aplikacije pridružene su liste globalnih varijabli.

5.2. LISTA TIPOVA PODATAKA

Razvoj aplikacija treba početi od definicije tipova podataka koji će se koristiti kroz cijelu aplikaciju. Tip podatka šifre partnera, na primjer, može se definirati kao numerički podatak od 7 mjesta. Svaki blok koji bude koristio neku varijablu s tipom podatka šifre partnera imat će u toj varijabli numerički podatak od 7 mjesta definiran tipom.

Princip repozitorija ima pozitivan efekt u situaciji naknadnih promjena podataka. U primjeru sa tipom podatka šifre partnera to znači da se tip podatka može promijeniti sa 7 na 8 mjesta. Nakon te promjene sva mjesta u aplikaciji gdje se spominje tip podatka šifre partnera bit će proširen sa 7 na 8 mjesta, ali ne automatikom nego kod stvarnog kreiranja bloka u memoriji.

Od podataka koji su bitni za definiciju tipova podataka treba izdvojiti domenu (numerička, znakovna, datumska itd), preciznost i veličinu i format prikaza.

5.3. LISTA KONEKCIJA

Svaka tablica unutar aplikacije mora pripadati nekoj od konekcija. Stoga su konekcije definirane u sustavu repozitorija, a tablice se na njih referenciraju putem stranog ključa.

Podaci koji su bitni za definiciju konekcije su korisničko ime, zaporka i poslužitelj. Mogući su i drugi podaci ukoliko to ciljna baza podataka zahtijeva.

5.4. LISTA TABLICA

Lista tablica služi za pohranu podataka koji su potrebni za detaljniju specifikaciju tablica, gdje svaki redak predstavlja jednu tablicu i sadrži bitne podatke poput konekcije, imena tablice u bazi podataka i slično.

Svaki redak liste tablica sadrži i specifikaciju polja iz tablice u bazi u obliku liste, pri čemu su za svako polje definirani podaci o imenu u bazi i tipu podatka.

Jednom kad su definirani konekcija i ime tablice, moguće je pročitati definiciju tablice direktno iz baze podataka i napuniti listu polja što ukida potrebu za ručnim unošenjem tih podataka.

5.5. LISTA BLOKOVA

Aplikacija mora imati definiranu listu korijenskih blokova koji su povezani sa hijerarhijskom strukturom podređenih blokova. Podređene blokove je moguće prikazivati u listi, ali ih je primjerenije prikazivati u strukturi stabla. Svaki redak u listi blokova osim svog imena, mora imati i sve druge bitne atribute.

5.6. LISTA TABLICA U BLOKU

Tablice koje se koriste u bloku moraju biti pobrojane u posebnoj veznoj listi, referencirajući se na listu tablica na nivou aplikacije. Razlog za to jest činjenica da jedna tablica u bazi može imati više uloga unitar bloka, te se u ovoj listi može pojavljivati i više puta. Od podataka u listi nužno je specificirati da li se radi o središnjoj ili vezanoj tablici, te mora imati pridodijeljen alias.

5.7. LISTA VARIJABLI

Lista varijabli omogućuje definiranje svih varijabli koje pripadaju nekom bloku, pri čemu svaka varijabla mora biti definirana sa svim svojim bitnim podacima te mora imati definirano:

(1) Ime varijablce

(2) Tip varijable (parametar, globalna varijabla, virtualna varijabla redka ili bloka, bazno polje)

(3) Ako je bazno polje onda mora imati poveznicu sa listom tablice bloku i odgovarajućeg polja u listi polja odgovarajuće tablice u repozitoriju

(4) Ako je globalna varijabla, ime globalne varijable na koju se odnosi

(5) Tip podatka

(6) Da li sudjeluje u SELECT, INSERT, UPDATE i DELETE SQL naredbama

(7) Da li varijabla sudjeluje u GROUP BY i ORDER BY klauzulama

(8) Doseg varijable (metoda na koju se odnosi)

Varijable mogu imati i druge dodatne podatke, na primjer atribute koji su potrebni za specifikaciju korisničkog sučelja i slično.

Varijable mogu imati i listu pripadajućih rekalkulacija i to s aspekta ciljne varijable (rekalkulacije koje mijenjaju varijablu), i sa aspekta izvorne varijable (promjena varijable koja uzrokuje rekalkulaciju neke druge varijable).

5.8. LISTA SQL ELEMENATA

SQL elementi bloka moraju biti pobrojani u listi SQL elemenata gdje su navedeni njihovi bitni atributi poput podvrste SQL elementa, te specifični atributi koji pripadaju podvrsti.

5.9. LISTA IZRAZA

Elementi izraza bloka su pobrojani u listi izraza što omogućuje da jedan izraz može biti korišten na više mjesta unutar bloka. Kod definiranja lista ostalih elemenata u bloku, mogu se koristiti izrazi iz ove liste, ali se mogu koristiti i konstante.

5.10. LISTA UVJETA

Svaki blok ima definiranu listu uvjeta u kojoj svaki redak predstavlja jedan element uvjeta, te u tom redku moraju biti specificirani svi podaci nužni za kreiranje elementa. U listu uvjeta ne ulaze korisnički uvjeti jer se oni kreiraju tijekom izvođenja aplikacije i privremenog su karaktera, već samo uvjeti definirani od strane razvojnog tima.

5.11. LISTA VEZA

Elementi veza u bloku definirani su u listi veza koja se po potrebi može referencirati na ostale liste (na primjer listu uvjeta ili listu tablica u bloku), te mora sadržavati atribut o vrsti veze.

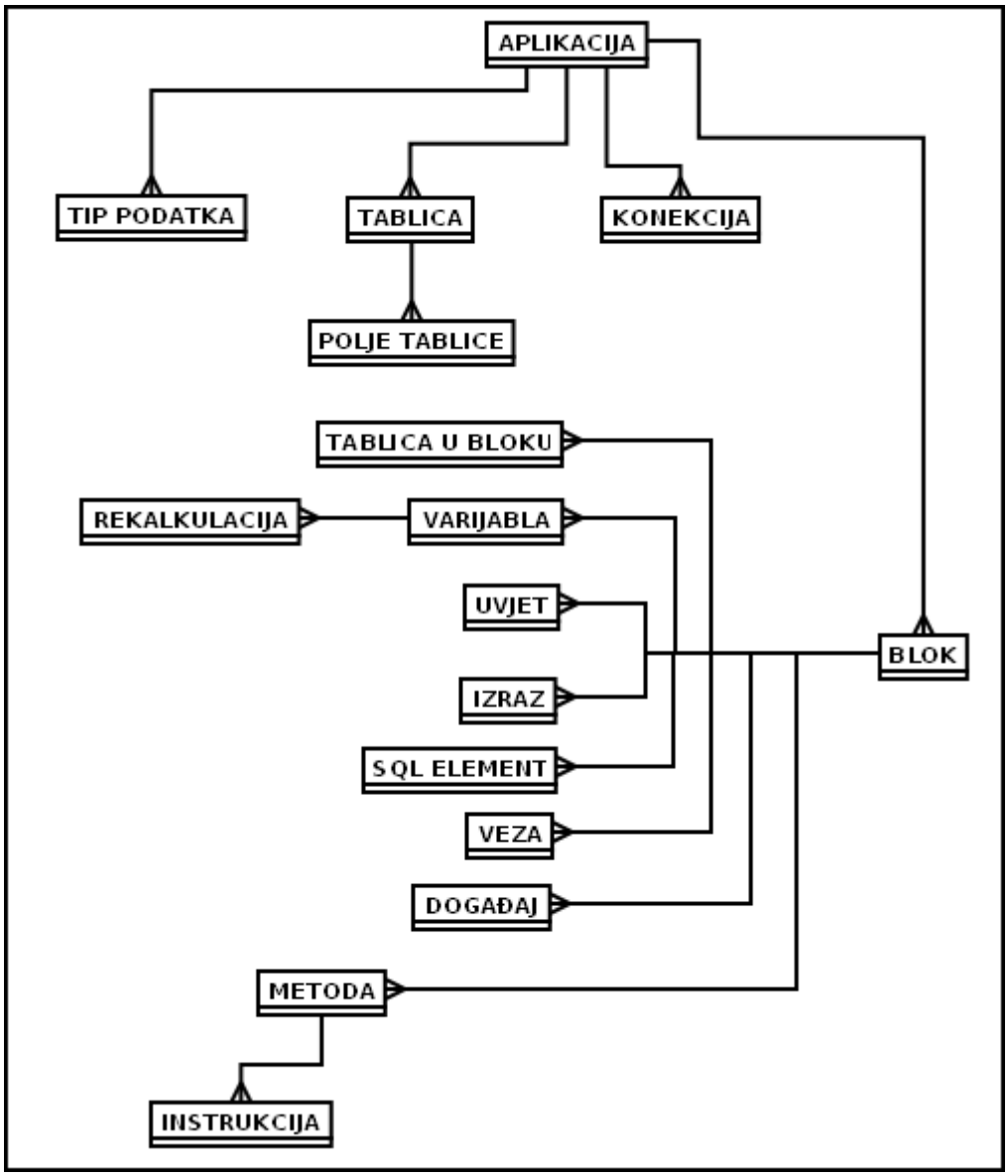
5.12. LISTA DOGAĐAJA

U listi događaja moraju biti pobrojani samo oni događaji bloka koji stvarno imaju vezanu neku od metoda sa programskom logikom. U listi događaja, osim tipa događaja i ostalih podataka koje pobliže opisuju događaj, mora biti vidljivo koja se metoda poziva prilikom aktiviranja događaja.

5.13. LISTA METODA

Lista metoda može biti povezana sa varijablama koje pripadaju metodi po dosegu varijable. Na taj način postiže se veća preglednost podataka.

Svaka metoda iz liste metoda ima pripadajuću listu instrukcija kojom se definira njena programska logika.



Slika 5.1 – ER model repozitorija

5.14. KORIŠTENJE REPOZITORIJA

Blokovi iz repozitorija kreiraju se dinamički prilikom poziva bloka. Slično kao kod dinamičkog kreiranja bloka, kod poziva bloka element aplikacije mora imati podatak o mjestu gdje je smješten repozitorij, iz njega izvući potrebne informacije iz odgovarajućih lista, te na temelju tih informacija kreirati blok i izvršiti ga. Ovaj mehanizam mora biti sastavni dio implementacije sustava koji se odvija automatzirano, bez posebne programske logike definirane od strane razvojnog sustava.

Timski razvoj aplikacije nameće ista transakcijska pravila članovima tima kao i običnim korisnicima aplikacija usmjerenih na baze podataka, jer i sam sustav repozitorija mora biti realiziran na taj način. Repozitorij se može proširiti dodatnim podacima koji odgovaraju razvojnim metodologijama tima (na primjer odgovornost pojedinih članova tima za određene blokove). Voditelj tima može odrediti pravila kojima može ograničiti pristup dijelova aplikacije nekim od članova tima, odnosno definirati podjelu prava i uloga pojedinim članovima.

Repozitorij ne mora nužno biti smješten u relacijskoj bazi. On može biti sadržan u u XML dokumentima, na CVS poslužitelju, definiran kao web servis i slično. Pri tome struktura repozitorija može biti jednaka, ali tada se gube transakcijske prednosti razvoja.

6. PROGRAMSKI PRIMJER

S obzirom na kompleksnost predloženog sustava za razvoj aplikacija usmjerenih na baze podataka, napravljen je programski primjer ograničenog dosega iz kojeg je vidljiv način na koji sustav funkcionira.

Za potrebe programskog primjera razvijena je osnovna implementacija svih elemenata apstraktnog modela, sa generatorom SQL-a za bazu podataka, te modulom za korisničko sučelje.

6.1. TEHNOLOŠKO OKRUŽENJE

Osnovna implementacija elemenata apstraktnog sučelja razvijena je u programskom jeziku Python i napravljena je tako da može funkcionirati višeplatformski, odnosno na svim platformama za koje je razvijen Python (Linux/Unix, Mac OS X, MS Windows, Java, .NET...). Kao baza podataka korišten je SQLite koji se standardno nalazi u Python distribuciji i obično se koristi kao ugrađena baza podataka (eng. embedded database) i za koju je napisan modul za generiranje SQL-ova iz definicije elemenata bloka, te komunikacijski modul. Modul za korisničko sučelje realiziran je wxPython bibliotekom koja je također višeplatformska. Ovi moduli predstavljaju osnovu implementacije sustava za razvoj aplikacija usmjerenih na bazu i korišteni su za realizaciju programskog primjera.

Programski primjer napisan je u Pythonu, koji koristeći prethodno spomenute module kreira u memoriji sve elemente potrebne za realizaciju zadane funkcionalnosti primjera.

6.1.1. PROGRAMSKI JEZIK PYTHON

Programski jezik Python je skriptni interpreterski programski jezik opće namjene primjenjiv za pisanje aplikativnih i sistemskih programskih rješenja. Često ga uspoređuju sa slijedećim programskim jezicima: Tcl, Perl, Ruby, Scheme i Java.

Odlikuju ga slijedeće osobine (Python Project, 2012):

- čiste i čitljiva sintaksa,

- snažne introspektivne mogućnosti

- objektna orjentiranost

- puna modularnost koja podržava hijerarhijske odnose među modulima

- obrada grešaka temeljena na događajima

- dinamični tipovi podataka

- proširiva standardna biblioteka modula

- ogromna količina modula pisana od strane korisnika

- jednostavna proširivost modulima pisanim u jezicima C, C++ ili Java

- jednostavna ugradivost u aplikacije gdje može poslužiti kao ugrađeni skriptni jezik

Autor Pythona, Guido van Rossum, započeo je njegov razvoj u kasnim osamdesetim godinama prošlog stoljeća sudjelujući na Amoeba projektu na CWI (Centrum Wiskunde & Informatica) institutu u Nizozemskoj, jednim od vodećih instituta te vrste u Europi. U prosincu 1989. godine bila je završena prva implementacijska verzija programskog jezika Python.

Guido van Rossum je prije toga sudjelovao u razvoju programskog jezika ABC, pa je taj programski jezik poslužio kao temelj za razvoj Pythona ali i prilika da ispravi brojne nedostatke koje je imao jezik ABC. Jedna od najvećih zamjerki odnosila se na otežanu proširivost jezika ABC novim modulima, pa je stoga tome posvetio veliku pažnju i danas je proširivost jedna od značajnijih osobina Pythona (Venners, 2003; Foord, 2009, 3-11).

Označavanje bloka koda indentacijom jest značajna osobina po kojoj se Python razlikuje od velike većine programskih jezika. Ostali programski jezici označavaju blok koda početnom i završnom ključnom riječi, npr. "begin" i "end" u Pascalu ili "{" i "}" u C/C++, dok se u Pythonu blok koda označava istovjetnom indentacijom (razmakom od početka linije) svih linija koda koje pripadaju bloku (Jones, 2002, 1-2; Mertz, 2003, 1-8; Downey, 2009, 1-6).

Usporedbe radi, u programskom jeziku C blok od dvije linije koje ispisuju riječi "Jedan" i "Dva" koji se nalazi u funkciji piše se:

```
int funkcija()
{
```

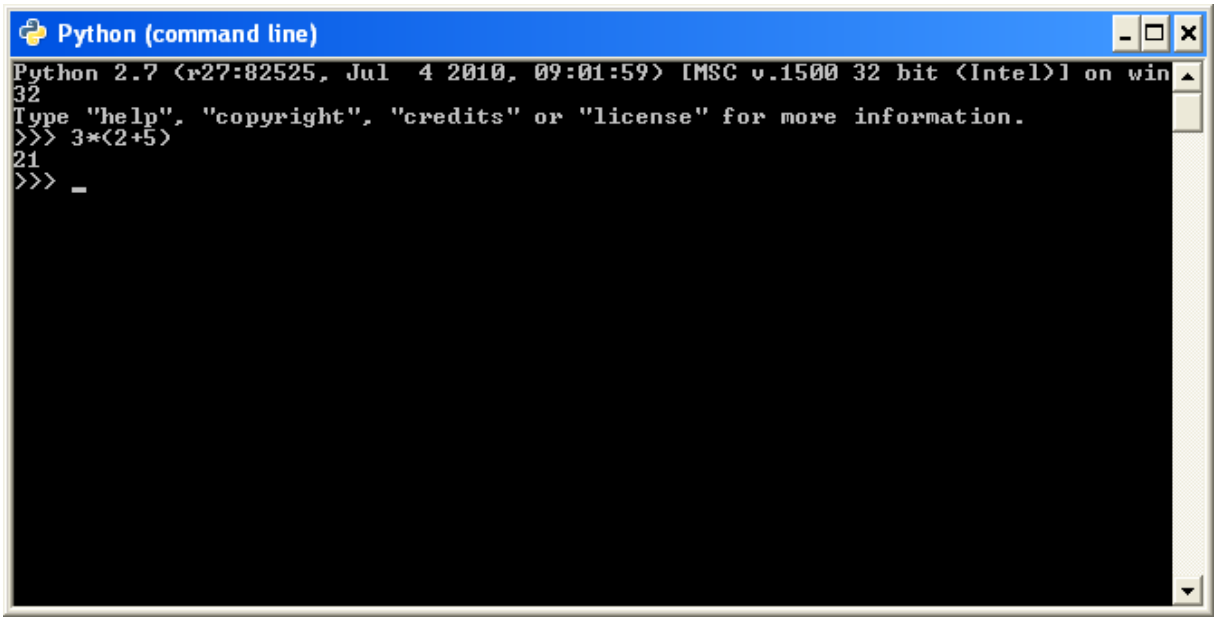
```
printf("Jedan\n");
printf("Dva");
}
```

Dakle blok od dvije naredbe za ispis omeđen je dvjema zagradama, dok se u Pythonu isti kod piše na slijedeći način:

```
def funkcija():
	print "Jedan"
	print "Dva"
```

Ova osobina čini stil pisanja koda indentacijom, ne samo mogućnošću koja kod čini čitljivijim, već to postaje sintaktička obveza, što za posljedicu ima veliku preglednost i čitljivost programskog koda. S druge strane, ova osobina je obično najveća prepreka početnicima koji prelaze na Python s nekog drugog programskog jezika (Raymond, 2000).

Python ima već ugrađenu podršku za brojne strukture podataka od kojih su najvažnije liste i asocijativni nizovi koji se mogu koristiti na krajnje efikasan način i koji uvelike pojednostavljuju pisanje kompleksnih programa, a snažna orjentiranost Pythona na mogućnosti proširenja omogućuju jednostavno dodavanje novih struktura.



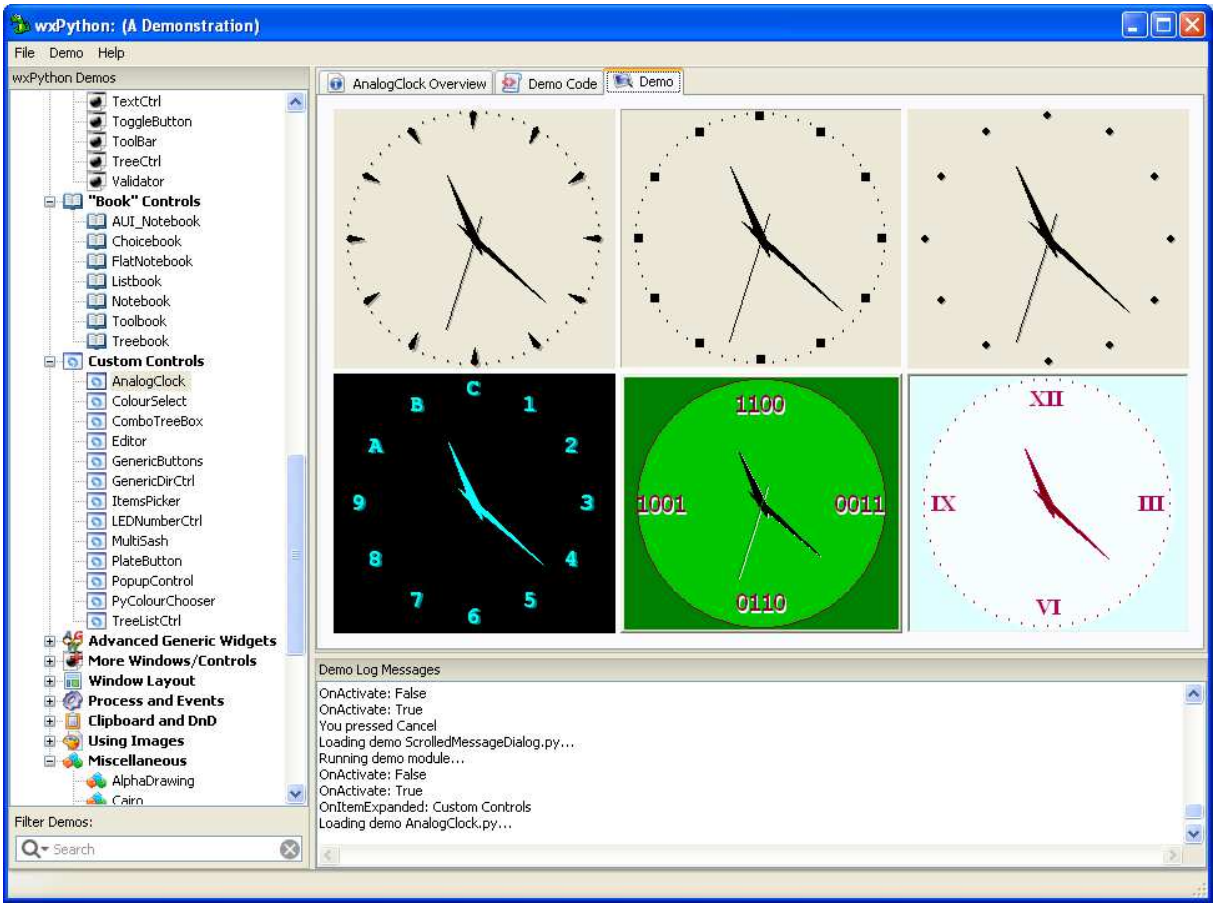
Slika 6.1.1 – Python interaktivna konzola

Programski jezik Python ima interaktivni mod koji omogućuje interaktivno testiranje modula, ali i korištenje pythona za potrebe administriranja operativnog sustava ili kao običan kalkulator (Eckel, 2001, 11-16; Ziade, 2008, 10-20).

Danas je Python jedan od programskih jezika tzv. glavne struje i koriste ga razvojni timovi širom svijeta za potrebe razvoja od malih usputnih skripti do velikih korporativnih aplikacija. Python se distribuira pod licencom otvorenog koda.

6.1.2. PROGRAMSKI MODUL WXPYTHON I WXWIDGETS BIBLIOTEKA

wxPython je programski modul koji je namijenjen izradi grafičkog korisničkog sučelja u programskom jeziku Python. Riječ je o multiplatformskom modulu koji se može izvršavati na svim značajnijim operativnim sustavima (Windows, Linux, Unix, OS X) (wxPython Project, 2012a; Rappin, 2006, 3-7).



Slika 6.1.2 – wxPython demo aplikacija

wxPython je, u tehnološkom smislu, modul koji obuhvaća wxWidgets biblioteku koristeći infrastrukturu za proširenje programskog jezika Python. wxWidgets biblioteka pisana je u programskom jeziku C++ (wxWidgets Project, 2012b; Precord, 2010, 7).

wxPython omogućuje rad sa svim osnovnim elementima grafičkog korisničkog sučelja poput formi, unosnih polja, tipki, padajućih menija, dijaloga i tablica, ali omogućuje rad i sa brojnim drugim elementima, poput satova, dijagrama, grafičkih animacija itd. Na taj način moguće je kreirati zaista bogato korisničko sučelje.

wxPython i wxWidgets su distribuirani pod licencom otvorenog koda.

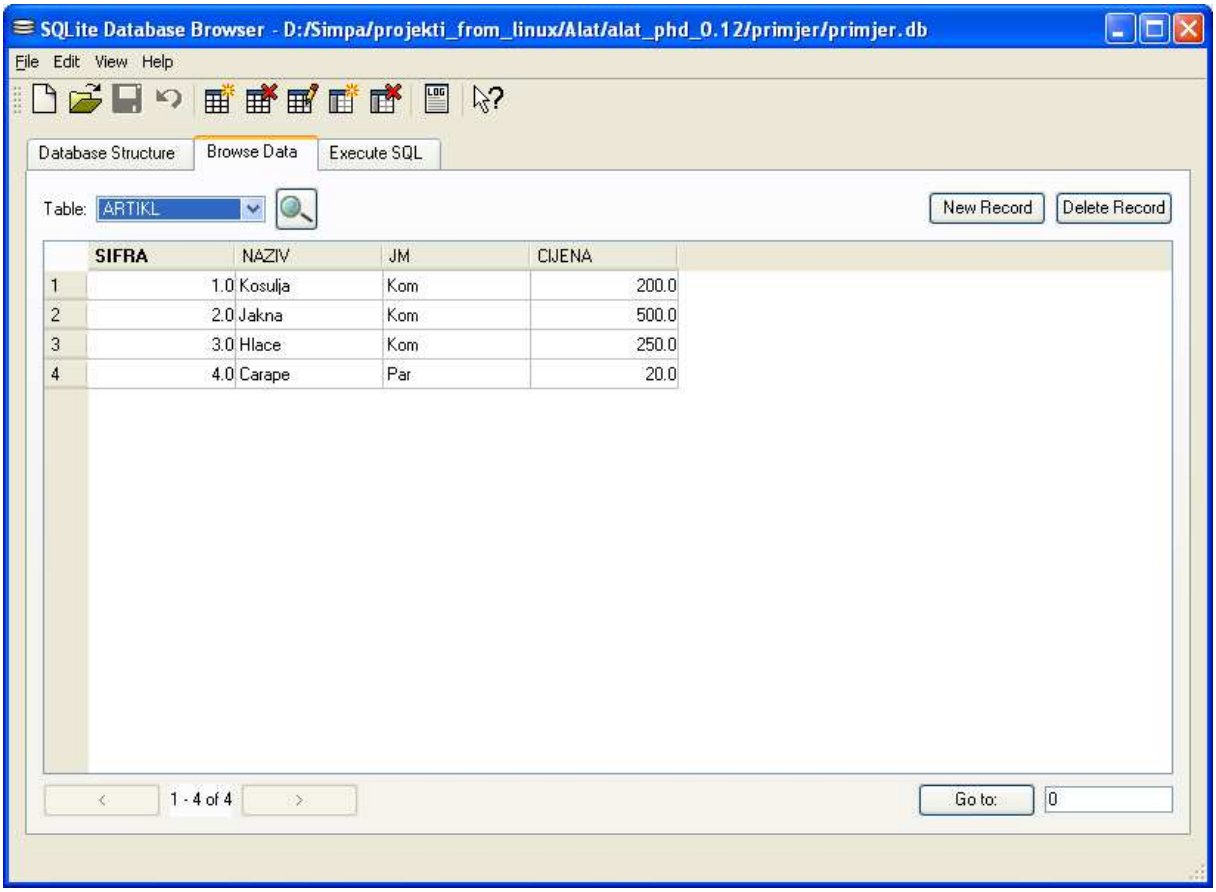
6.1.3. SQLITE BAZA PODATAKA

SQLite je relacijska baza podataka i progamska biblioteka koja implementira samostojeći transakcijski SQL mehanizam (SQLite Project, 2012a). SQLite se ne instalira na poslužitelje već na lokalni stroj i nije mu potrebno nikakvo podaševanje. Uglavnom se koristi kao

ugrađena baza pa tako se koristi u Pythonu, ali i na drugim mjestima (SQLite Project, 2012b; Kreibich, 2010, 1-6; Allen, 2010, 1-17; Wei, 2012, 27-40; Lans, 2009, 3-22) poput:

- iPhone i Android mobilnim uređajima
- Firefox, Google Chrome i Apple Safari internetskim preglednicima
- Adobe aplikacijama poput Adobe Readera
- Airbus zrakoplovima
- McAfee antivirusnom softveru
- Skype klijentskom softveru

Zbog svoje jednostavnosti i malog zauzeća prostora koristi se na brojnim mjestima gdje je potrebno zapisati podatke u nekom kompleksnijem formatu u datoteku na disku. SQLite također omogućuje i pohranu podataka u radnoj memoriji računala čime se postižu velike brzine rada. Ima ugrađene transakcijske mehanizme, a omogućuje kreiranje baznih objekata koji se pojavljuju u vodećim relacijskim bazama podataka poput pogleda, okidača (eng. trigger) te pohranjenih procedura (eng. stored procedure).



Slika 6.1.3 – SQLite preglednik baze podataka

6.2. IMPLEMENTACIJA APSTRAKTNOG MODELA

Implementacija apstraktnog modela koja je izgrađena za potrebe programskog primjera je djelomična ali dovoljna za uporabu. Implementacija je izgrađena već spomenutim tehnologijama (Python, SQLite), a klase su podijeljene u module.

6.2.1. MODUL "__init__.py"

Modul "__init__.py" je inicijalizacijski modul koji iz ostalih modula importira klase koje odgovaraju pojedinim elementima apstraktnog modela.

6.2.2. MODUL "_base_block.py"

Modul "_base_block.py" sadrži slijedeće klase:

(1) klasa "Block" odgovara elementu bloka
(2) klasa "Application" odgovara elementu aplikacije
(3) pomoćna klasa "BlockWrapper" omogućuje referenciranje varijabli bloka u izrazima.

6.2.3. MODUL "_base_block_buffer.py"

Modul "_base_block_buffer.py" sadrži slijedeće klase:

(1) klasa "Buffer" odgovara elementu spremnika podataka
(2) klasa "MethodCall" odgovara pozivu metode u elementu spremnika poziva metoda

6.2.4. MODUL "_base_code.py"

Modul "_base_code.py" sadrži slijedeće klase:

(1) klasa "Event" je krovna klasa koja odgovara elementu događaja
(2) klasa "EventBlockDescriptor" odgovara događaju opisa bloka
(3) klasa "EventBlockFetch" odgovara događaju dohvata na bloku
(4) klasa "EventRecordFetch" odgovara događaju dohvata na redku
(5) klasa "EventBlockBefore" odgovara događaju prije bloka
(6) klasa "EventBlockAfter" odgovara događaju poslije bloka
(7) klasa "EventRecordBefore" odgovara događaju prije redka
(8) klasa "EventRecordAfter" odgovara događaju poslije redka
(9) klasa "EventFieldBefore" odgovara događaju prije varijable
(10) klasa "EventFieldAfter" odgovara događaju poslije varijable
(11) klasa "EventFieldAction" odgovara događaju akcije na varijabli
(12) klasa "EventExternal" odgovara vanjskom događaju
(13) klasa "EventFieldChange" odgovara događaju promjene vrijednosti varijable
(14) klasa "EventError" odgovara događaju greške
(15) klasa "EventBlockExit" odgovara događaju izlaska iz bloka
(16) klasa "Method" odgovara elementu metode
(17) klasa "_Instruction" odgovara elementu instrukcije
(18) klasa "UpdateInstr" odgovara instrukciji pridruživanja
(19) klasa "EvaluateInstr" odgovara instrukciji evaluacije

(20) klasa "DumpInstr" odgovara instrukciji komentiranja
(21) klasa "CallExtInstr" odgovara instrukciji vanjskog poziva
(22) klasa "IfInstr" odgovara instrukciji "ako"
(23) klasa "ElseInstr" odgovara instrukciji "inače"
(24) klasa "WhileInstr" odgovara instrukciji iteracije
(25) klasa "ExecSQLStatementInstr" odgovara SQL instrukciji
(26) klasa "CallInstr" odgovara instrukciji poziva bloka
(27) klasa "SubCallInstr" odgovara instrukciji poziva podbloka
(28) klasa "CommitInstr" odgovara commit instrukciji
(29) klasa "RollbackInstr" odgovara rollback instrukciji
(30) klasa "ReturnInstr" odgovara povratnoj instrukciji

6.2.5. MODUL "_base_condition.py"

Modul "_base_condition.py" sadrži slijedeće klase:

(1) klasa "Operator" je pomoćna klasa koja pobliže opisuje operatore koji se koriste u elementu uvjeta
(2) klasa "Condition" je krovna klasa elementa uvjeta
(3) klasa "OpCond" odgovara uvjetu s operacijom
(4) klasa "ClauseCond" odgovara uvjetu klauzule
(5) klasa "ExprCond" odgovara uvjetu s izrazom

6.2.6. MODUL "_base_datatype.py"

Modul "_base_datatype.py" sadrži slijedeće klase:

(1) klasa "Type" je krovna klasa elementa tipa podatka
(2) klasa "GenericType" odgovara generičkom tipu podatka
(3) klasa "StringType" odgovara znakovnom tipu podatka
(4) klasa "NumberType" odgovara numeričkom tipu podatka
(5) klasa "BooleanType" odgovara logičkom tipu podatka
(6) klasa "DateTimeType" odgovara datumsko-vremenskom tipu podatka

6.2.7. MODUL "_base_expression.py"

Modul "_base_expression.py" sadrži slijedeće klase:

(1) klasa "Expression" je krovna klasa elementa izraza
(2) klasa "ConstExpr" je klasa koja opisuje element izraza s konstantom
(3) klasa "EvalExpr" je klasa koja opisuje element izraza sa znakovnom definicijom
(4) klasa "LambdaExpr" je klasa koja opisuje element izraza sa funkcijom Pythona

6.2.8. MODUL "_base_field.py"

Modul "_base_field.py" sadrži slijedeće klase:

(1) klasa "Field" je krovna klasa elementa varijable
(2) klasa "DbField" odgovara baznom polju
(3) klasa "RowField" odgovara virtualnoj varijabli redka
(4) klasa "BlockField" odgovara virtualnoj varijabli bloka
(5) klasa "GlobalField" odgovara globalnoj varijabli
(6) klasa "IOParameter" odgovara ulazno-izlaznom parametru
(7) klasa "InputParameter" odgovara ulaznom parametru
(8) klasa "OutputParameter" odgovara izlaznom parametru
(9) klasa "Recalc" odgovara elementu rekalkulacije

6.2.9. MODUL "_base_join.py"

Modul "_base_join.py" sadrži slijedeće klase:

(1) klasa "Join" je krovna klasa elementa veze
(2) klasa "InnerJoin" odgovara unutarnjoj vezi
(3) klasa "LeftOuterJoin" odgovara lijevoj vanjskoj vezi

6.2.10. MODUL "_base_sqlstatement.py"

Modul "_base_sqlstatement.py"

(1) klasa "DirectSQL" je pomoćna klasa koja opisuje direktni SQL u SQL elementima
(2) klasa "SQLExpr" je pomoćna klasa koja opisuje SQL izraze u SQL elementima
(3) klasa "SQLStatement" je krovna klasa SQL elementa

(4) klasa "SQLSelect" odgovara SQL selekciji
(5) klasa "SQLSelectRefetch" odgovara SQL selekciji sloga
(6) klasa "SQLLock" odgovara SQL zaključavanju
(7) klasa "SQLLockRefetch" odgovara SQL zaključavanju sloga
(8) klasa "SQLInsertRow" odgovara SQL dodavanju sloga
(9) klasa "SQLUpdateRow" odgovara SQL izmjeni sloga
(10) klasa "SQLDeleteRow" odgovara SQL brisanju sloga

6.2.11. MODUL "_base_table.py"

Modul "_base_table.py" sadrži slijedeće klase:

(1) klasa "Table" je krovna klasa elementa tablice
(2) klasa "MainTable" odgovara središnjoj tablici
(3) klasa "LinkTable" odgovara vezanoj tablici

6.2.12. MODUL "common.py"

Modul "common.py" sadrži zajedničke klase i funkcije implementacije apstraktnog modela koje obavljaju pomoćne poslove u drugim modulima.

6.2.13. MODUL "db.py"

Modul "db.py" je krovni modul osnovnih klasa za komunikaciju s bazom podataka i generiranje SQL naredbi. Sadrži slijedeće klase:

(1) klasa "Connection" odgovara elementu konekcije
(2) klasa "SQLGenerator" odgovara elementu SQL generatora

6.2.14. MODUL "db_sqlite3.py"

Modul "db_sqlite3.py" nasljeđuje klase modula "db.py" i ugrađuje specifičnosti SQLite baze podataka.

6.2.15. MODUL "lib.py"

Modul "lib.py" je modul koji sadrži funkcije koje se mogu koristiti u elementima izraza.

6.3. IMPLEMENTACIJA KORISNIČKOG SUČELJA

Implementacija korisničkog sučelja izgrađenog za potrebe programskog primjera izgrađena je u wxPython tehnologiji.

6.3.1. MODUL "__init__.py"

Modul "__init__.py" je inicijalizacijski modul koji iz ostalih modula importira klase koje odgovaraju pojedinim elementima korisničkog sučelja.

6.3.2. MODUL "_common.py"

Modul "_common.py" sadrži temeljne generičke klase elemenata korisničkog sučelja. Te klase se ne koriste direktno, već kao osnova za proces nasljeđivanja. Sadrži slijedeće klase:

(1) klasa "Widget" je roditeljska klasa svih elemenata na panelu
(2) klasa "Grid" je roditeljska klasa elementa tablice
(3) klasa "_GridCellEditor" je klasa za unos podataka u ćeliju
(4) klasa "_GridData" je klasa za dinamički dohvat podataka

6.3.3. MODUL "_containers.py"

Modul "_containers.py" sadrži slijedeće klase:

(1) klasa "Application" je korijenska klasa podsustava korisničkog sučelja
(2) klasa "AuiMDIParentFrame" je klasa za vrstu forme
(3) klasa "Container" je roditeljska klasa za panel
(4) klasa "AuiMDIChildFrame" je klasa za vrstu forme
(5) klasa "Frame" je klasa za vrstu forme

6.3.4. MODUL "_interface.py"

Modul "_interface.py" sadrži samo klasu "Interface" koja služi kao posrednik između klasa implementacije apstraktnog modela i klasa implementacije korisničkog sučelja

6.3.5. MODUL "_panel.py"

Modul "_panel.py" sadrži slijedeće klase:

(1) klasa "_Panel" je roditeljska klasa fiksnog i dinamičkog panela
(2) klasa "ScrolledPanel" odgovara dinamičkom elementu panela
(3) klasa "Panel" odgovara fiksnom elementu panela

6.3.6. MODUL "_sizer.py"

Modul "_sizer.py" sadrži klase za dinamički raspored elemenata po panelu i to:

(1) klasa "Sizer" je klasa za mrežni raspored elemenata
(2) klasa "_BoxSizer" je roditeljska klasa za linijski raspored elemenata
(3) klasa "VSizer" je klasa za vertikalni linijski raspored elemenata
(4) klasa "HSizer" je klasa za horizontalni linijski raspored elemenata

6.3.7. MODUL "_table.py"

Modul "_table.py" sadrži slijedeće klase:

(1) klasa "Table" odgovara elementu tablice korisničkog sučelja
(2) klasa "_TableData" je klasa za dohvat podataka koja komunicira sa elementom spremnika podataka implementacije apstraktnog modela

6.3.8. MODUL "_widgets.py"

Modul "_widgets.py" sadrži slijedeće klase:

(1) klasa "Label" je klasa za ispis teksta na elementu panela
(2) klasa "Button" odgovara elementu tipke
(3) klasa "Edit" odgovara elementu unosnog polja

6.4. OPIS FUNKCIONALNOSTI

Kao programski primjer uzet je sustav sa unosom dokumenata, stavki dokumenata i pripadajućih artikala, zajedno sa praćenjem stanja skladišta. Ovo je vrlo pojednostavljeni primjer i upravo zbog toga nije primjenjiv u realnim situacijama, ali može poslužiti kao ilustracija. Primjer se sastoji od tri bloka.

Prvi blok je blok dokumenata koji služi za unos zaglavlja dokumenata koji se unosi i formi tablice koja ima tri polja: ID dokumenta, datum i napomenu. ID dokumenta je brojač koji se

dodjeljuje automatizmom, dok se datum i napomena mogu slobodno korisnički unositi. Iz bloka dokumenata moguće je pozvati blok artikala, blok stavki dokumenta ili izaći iz aplikacije.

Drugi blok odnosi se na stavke dokumenta, ali ne stavke svih dokumenata, već samo onog dokumenta iz kojeg je blok stavki pozvan. Unos je podijeljen na tablicu i listu polja. U tablici se nalaze polja ID stavke, ID dokumenta, šifra artikla, ulaz i izlaz. ID stavke je brojač stavke koji sustav sam dodjeljuje automatizmom, kao i ID dokumenta koji predstavlja poveznicu prema pripadajućem dokumentu. Šifra artikla je polje koje označava artikl koji se koristi u stavci i mora se nalaziti u šifarniku artikala. Sustav mora upozoriti korisnika koji želi koristiti nepostojeću šifru artikla i ne smije dozvoliti unos takve stavke. Konačno, polja ulaz i izlaz predstavljaju količine ulaza odnosno izlaza u jedinici mjere navedenoj u šifarniku artikla. U listi polja nalaze se podaci o artiklu (naziv, jedinica mjere i cijena) koje su dohvaćene iz šifarnika artikla, te polje stanje koje je dobijeno sumiranjem svih ulaza i izlaza koji se odnose na traženi artikl. Korisnik u ovom bloku smije unositi samo šifru artikla, ulaz i izlaz, dok ostale podatke ne može mijenjati ručno. Iz ovog bloka moguće je vratiti se na prethodni blok dokumenata ili ući u blok artikala.

Treći blok, blok artikala, služi za unos šifarnika artikala u formi tabličnog unosa. Polja koja se nalaze u šifraniku su: šifra artikla, naziv artikla, jedinica mjere i cijena. Sva polja su slobodna za unos, s tim da šifra artikla mora biti jedinstvena, ali se može slobodno odrediti. Iz ovog bloka moguće je vratiti se u prethodni iz kojeg je pozvan.

6.5. OPIS MODELA BAZE PODATAKA

Za potrebe realizacije modela korištene su tri tablice i jedan pogled. Tablice su: DOKUMENT, STAVKA i ARTIKL, a pogled je SKLADISTE.

Slijedi opis tablica:

(a) Tablica DOKUMENT

Rbr	Polje	Tip podatka	Ključ?	Napomena
1	ID	Numerički	Da, primarni	ID dokumenta
2	DATUM	Datumski	Ne	Datum dokumenta
3	NAPOMENA	Znakovni	Ne	Napomena dokumenta

(b) Tablica STAVKA

Rbr	Polje	Tip podatka	Ključ?	Napomena
1	ID	Numerički	Da, primarni	ID stavke
2	ID_DOKUMENTA	Numerički	Da, strani	ID dokumenta
3	SIFRA_ARTIKLA	Numerički	Da, strani	Šifra artikla
4	ULAZ	Numerički	Ne	Količina ulaza
5	IZLAZ	Numerički	Ne	Količina izlaza

(c) Tablica ARTIKL

Rbr	Polje	Tip podatka	Ključ?	Napomena
1	SIFRA	Numerički	Da, primarni	Šifra artikla
2	NAZIV	Znakovni	Ne	Naziv artikla
3	JM	Znakovni	Ne	Jedinica mjere
4	CIJENA	Numerički	Ne	Cijena artikla

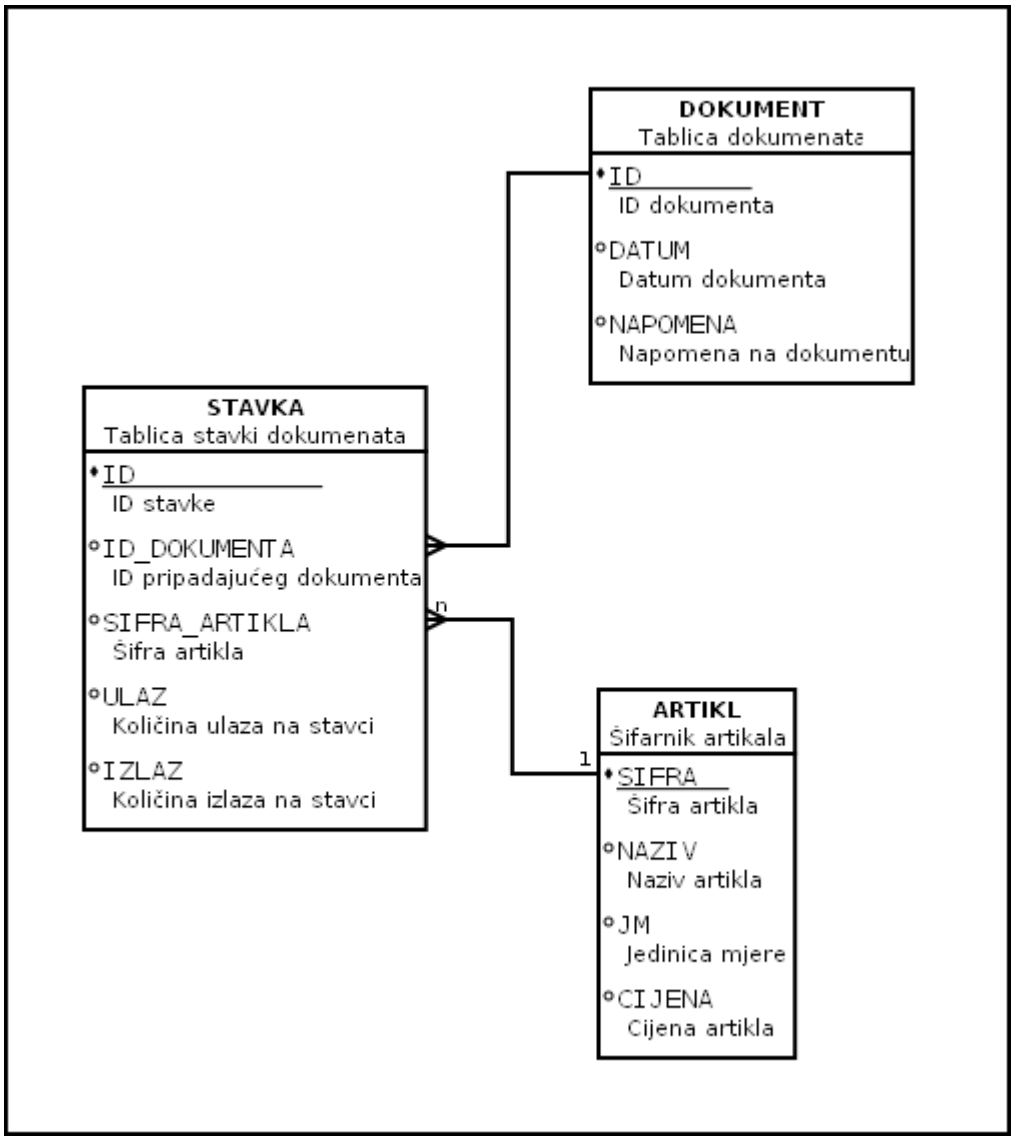
(d) Pogled SKLADISTE

Rbr	Polje	Tip podatka	Napomena
1	SIFRA_ARTIKLA	Numerički	Šifra artikla
2	STANJE	Numerički	Stanje

Slijedi SQL kod koji kreira gore navedene objekte u ciljnoj SQLite3 bazi podataka. Nisu navođeni tipovi podataka jer SQLite3 to ne zahtijeva.

```
CREATE TABLE DOKUMENT (
    ID PRIMARY KEY,
```

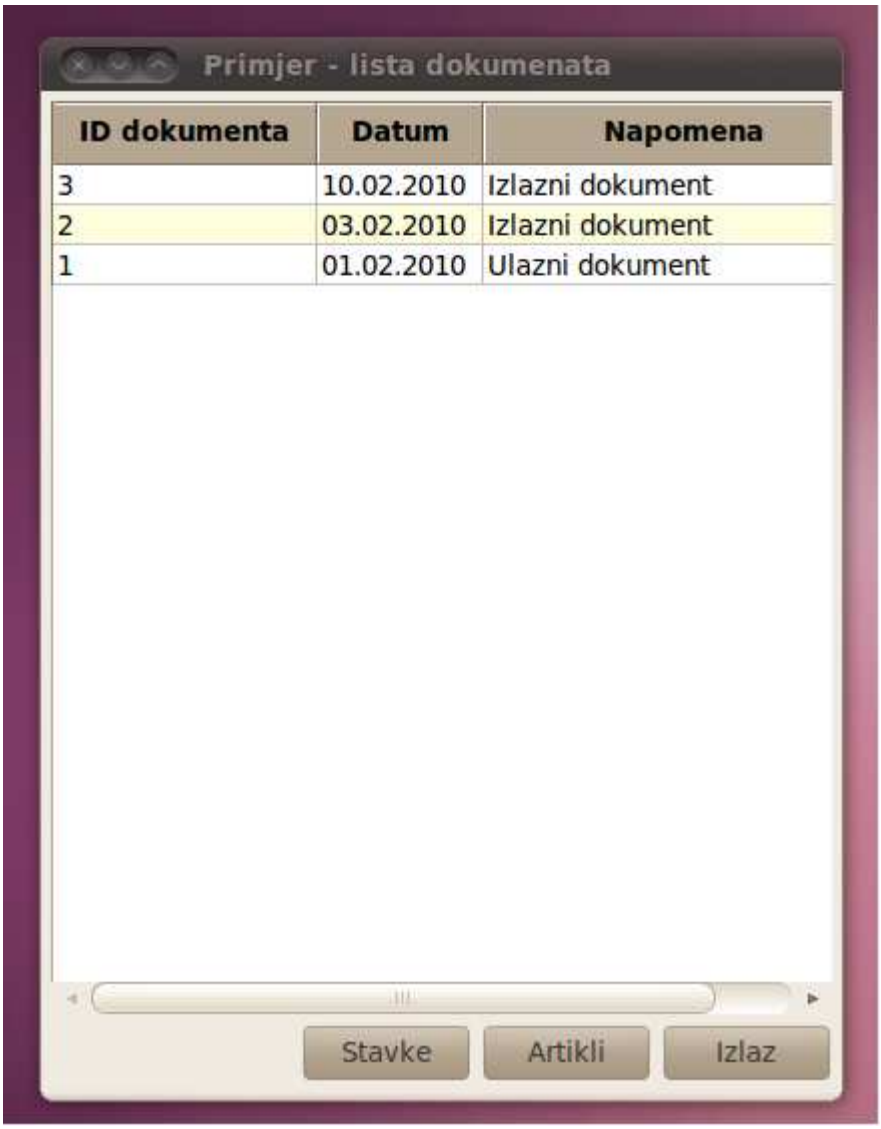
```
        DATUM,
        NAPOMENA);

CREATE TABLE STAVKA (
        ID PRIMARY KEY,
        ID_DOKUMENTA,
        SIFRA_ARTIKLA,
        ULAZ DEFAULT 0,
        IZLAZ DEFAULT 0);

CREATE TABLE ARTIKL (
        SIFRA PRIMARY KEY,
        NAZIV,
        JM,
        CIJENA);

CREATE VIEW SKLADISTE AS
        SELECT
                SIFRA_ARTIKLA,
                SUM(ULAZ-IZLAZ) AS STANJE
        FROM STAVKA
        GROUP BY SIFRA_ARTIKLA;
```



Slika 6.4 – ER dijagram modela podataka

6.6. BLOK DOKUMENATA

Slijedi specifikacija elemenata bloka dokumenata. Nije naveden originalni kod, već je specifikacija izložena u opisnom obliku koji je razumljiviji.



Slika 6.6 – Korisničko sučelje bloka dokumenata

6.6.1. VIRTUALNE VARIJABLE BLOKA

(a) CGUI = globalna varijabla, generička

(b) CAPP = globalna varijabla, generička

(c) LIB = globalna varijabla, generička

(d) CFRAME = varijabla bloka, generička

(e) CPANEL = varijabla bloka, generička

(f) B_STAVKE = varijabla bloka, numerička, tipka "Stavke" u korisničkom sučelju

(g) B_ARTIKLI = varijabla bloka, numerička, tipka "Artikli" u korisničkom sučelju

(h) B_IZLAZ = varijabla bloka, numerička, tipka "Izlaz" u korisničkom sučelju

Riječ je o varijablama na nivou bloka koje služe za potrebe definiranja i upravljanja korisničkim sučeljem (a) - (e). Varijable (f) (g) i (h) su tipke "Stavke", "Artikli" i "Izlaz" na korisničkom sučelju i služe tome da bi mogle preuzeti događaj akcije na varijabli.

6.6.2. DEFINICIJA TABLICA

(a) D = Glavna tablica DOKUMENT

(b) L = Vezana tablica DOKUMENT

U bloku dokumenata glavnu ulogu vode dva elemenata tablica, koje se zapravo odnose na istu tablicu u bazi podataka. Glavna tablica D služi za prikaz i unos podataka, a vezana tablica L dohvaća najviši zapisani ID u tablici dokumenata, kako bi sustav mogao generirati slijedeći.

6.6.3. DEFINICIJA POLJA U TABLICAMA

(a) D_ID = Polje ID iz tablice D, numeričkog tipa

- dio je SELECT i INSERT SQL naredbi
- dio je identifikatora redka
- dio je ORDER BY klauzule i to u opadajućem poretku
- ne smije se modificirati
- vidi se u tablici korisničkog sučelja pod nazivom "ID dokumenta"

(b) D_DATUM = Polje DATUM iz tablice D, datumskog tipa

- dio je SELECT, INSERT i UPDATE SQL naredbi
- vidi se u tablici korisničkog sučelja pod nazivom "DATUM"

(c) D_NAPOMENA = Polje NAPOMENA tablice D, znakovnog tipa

- dio je SELECT, INSERT i UPDATE SQL naredbi

- vidi se u tablici korisničkog sučelja pod nazivom "NAPOMENA"

(d) L_ID = Polje ID tablice L, numeričkog tipa

- dio je SELECT SQL naredbe

- ne smije se mijenjati

- ima agregatnu funkciju MAX

Gore navedeni elementi predstavljaju polja iz tablice koja će biti korištena u bloku. Polje D_ID predstavlja identifikator redka u bloku i ne smije se modificirati, nego samo dohvaćati SELECT naredbom i zapisivati INSERT naredbom u slučaju novog redka. Polja D_DATUM i D_NAPOMENA su polja slobodnog unosa i na njima nema nikakvog ograničenja. Polje L_ID služi za generiranje SELECT MAX(ID) FROM DOKUMENT naredbu kojom se dohvaća najveći ID dokumenta, kako bi sustav mogao izračunati slijedeći.

6.6.4. SQL ELEMENTI

(a) STM_D = SQL SELECT element tablice D, za početno dohvaćanje

(b) STM_D_UPDATE = SQL UPDATE element tablice D

(c) STM_D_DELETE = SQL DELETE element tablice D

(d) STM_D_INSERT = SQL INSERT element tablice D

(e) STM_L = SQL SELECT element za dohvaćanje sloga vezane tablice

Ovdje su definirani SQL elementi nužni za rad s tablicama D i L, pri čemu se tablici D dohvaćaju vrijednosti, ali se i umeću, modificiraju i brišu, dok se kod tablice L samo dohvaćaju.

6.6.5. DOGAĐAJ OPISA BLOKA

Događaj opisa bloka definira korisničko sučelje bloka i to pozivom vanjskih metoda modula korisničkog sučelja, preko poveznica u varijablama CGUI, CAPP, CFRAME i CPANEL, dok

se globalnoj varijabli LIB pridružuje biblioteka funkcija koja se koristi u bloku. S obzirom da modul korisničkog sučelja koji je razvijen za ovaj primjer nije detaljno specificiran, neće biti specificiran ni događaj opisa bloka.

6.6.6. DOGAĐAJ DOHVATA NA BLOKU

(a) SQL instrukcija SQL elementa STM_D, nivo 0

Ova SQL instrukcija dohvaća sve slogove tablice D u opadajućem redoslijedu i zapisuje ih u spremnik podataka. Događaj se odvija na početku izvršavanja bloka.

6.6.7. DOGAĐAJ POSLIJE REDKA

(a) Instrukcija evaluacije izraza "func_write()", nivo 0

Događaj poslije redka poziva se prilikom izlaska iz redka i predstavlja mjesto gdje se zapisuju promjene na slogu u bazu podataka. U ovom bloku za zapisivanje koristi još i mjesto prije poziva bloka stavaka, pa su instrukcije zapisivanja smještene u metodu FUNC_WRITE tako da se ne moraju ponavljati na dva mjesta.

6.6.8. METODA "FUNC_WRITE"

(a) SQL instrukcija STM_D_DELETE, nivo 0, ako je potrebno brisanje

(b) SQL instrukcija STM_D_UPDATE, nivo 0, ako treba modificirati tablicu D

(c) Instrukcija AKO, nivo 0, ako je potrebno umetanje podataka

(d) SQL instrukcija STM_L, nivo 1

(e) Instrukcija pridruživanja varijabli D_ID vrijednosti izraza "L_ID+1", nivo 1

(f) SQL instrukcija STM_D_INSERT, nivo 1

(g) COMMIT instrukcija, nivo 0

(h) Povratna instrukcija, nivo 0, vraća vrijednost logičko "da"

Ova metoda zapisuje promjene podataka u redku u bazu podataka. Napravljena je iz razloga što je ovu operaciju potrebno napraviti na dva mjesta; događaju nakon redka, te prije poziva bloka stavaka. Instrukcija (a) briše redak ako je redak označen za brisanje. Instrukcija (b)

ažurira sadržaj redka u bazi ukoliko je došlo do promjene podataka u redku. Situacija oko umetanja podataka je nešto složenija, jer je potrebno izvršiti više operacija. Stoga je kreirana AKO instrukcija (c) koja ispituje da li je potrebno umetnuti podatke u bazu, i ako jest izvršavaju se instrukcije nivoa 1 - (d), (e) i (f). Instrukcija (d) dohvaća trenutni najviši ID dokumenta, a instrukcija (e) ažurira D_ID polje sa slijedećom vrijednosću (najviši ID uvećan za 1). Konačno, instrukcija (f) umeće redak u bazu podataka. Instrukcija (g) izvršava COMMIT naredbu, a instrukcija (h) vraća logičko "da".

6.6.9. DOGAĐAJ AKCIJE NA VARIJABLI "B_STAVKE"

(a) Instrukcija poziva bloka, koja izvršava poziv bloka stavaka

- kao uvjet, navodi se poziv metode FUNC_WRITE

- kao parametar poziva navodi se varijabla D_ID

Instrukcija (a) poziva blok stavaka po imenu. Izraz uvjeta (u ovom slučaju poziv funkcije FUNC_WRITE) bit će izvršen prije poziva bloka stavaka, i to samo u slučaju da je FUNC_WRITE metoda vratila logičko "da". U slučaju da upis u bazu nije uspio poziv bloka stavaka se neće izvršiti. Poziv metode FUNC_WRITE je važan jer osigurava da je dokument zapisan i da mu je dodijeljen ID koji se predaje kao parametar, pa se nove stavke mogu pridodijeliti tom ID-ju dokumenta.

6.6.10. DOGAĐAJ AKCIJE NA VARIJABLI "B_ARTIKLI"

(a) Instrukcija poziva bloka, koja izvršava poziv bloka artikala, nivo 0

Kao i prethodna metoda, i ova metoda se temelji na instrukciji poziva blokova, s tom razlikom što se ovdje ne poziva metoda FUNC_WRITE, jer nema potrebe za zapisivanjem dokumenta.

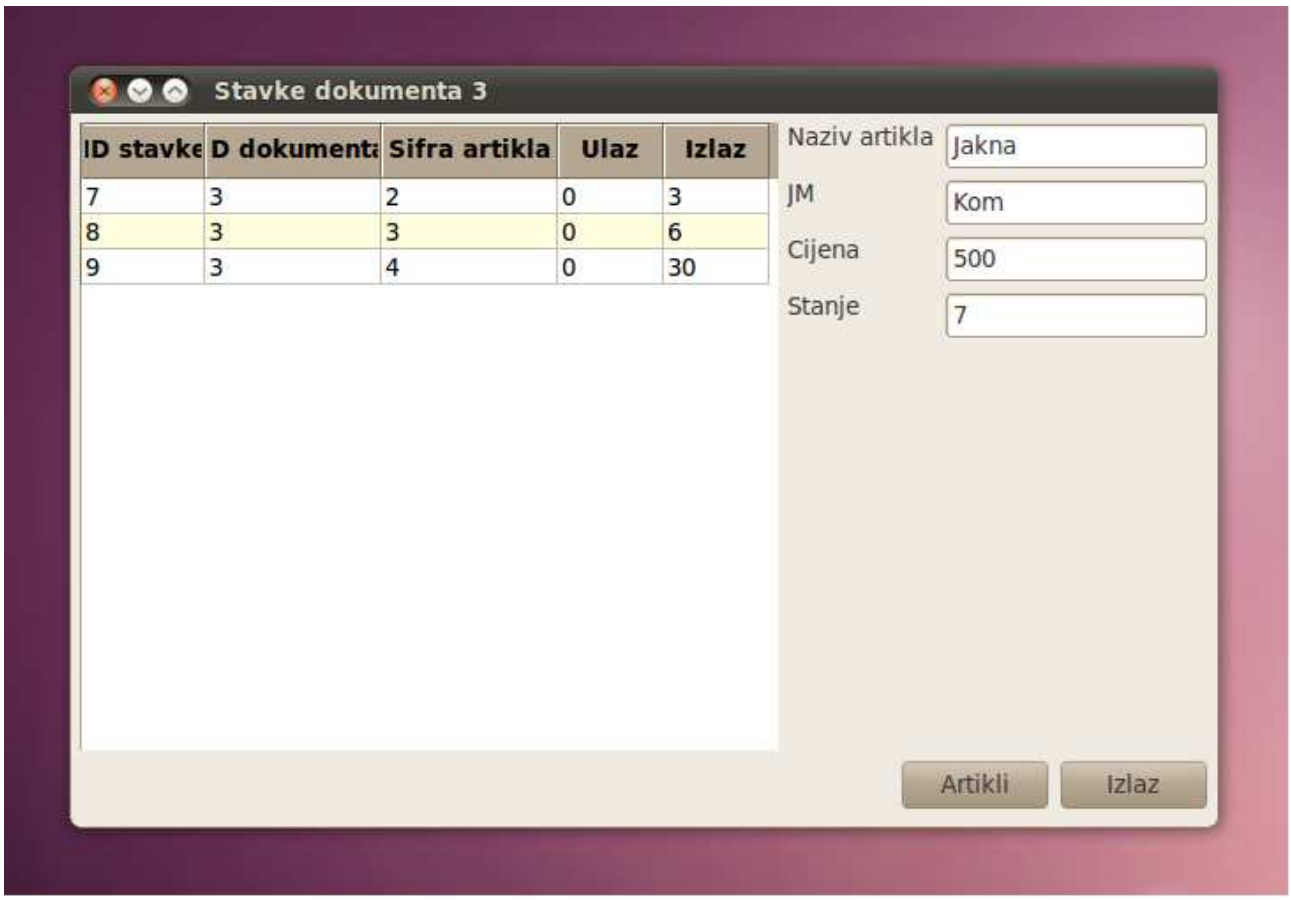
6.6.11. DOGAĐAJ AKCIJE NA VARIJABLI "B_IZLAZ"

(a) Instrukcija vanjskog poziva metode korisničkog sučelja, koja izvršava zatvaranje unosne forme, nivo 0

Ova instrukcija poziva metodu zatvaranja unosne forme biblioteke korisničkog sučelja, čime se ujedno i završava rad aplikacije.

6.7. BLOK STAVAKA

Blok stavaka poziva se iz bloka dokumenata i služi za unos stavaka izabranog dokumenta. Blok stavaka ima ulazni parametar kojim dobija informaciju o kojem se dokumentu radi, te ima dvije tipke na formi: "Artikli" kojim poziva blok artikala i "Izlaz" kojim zatvara unosna forma i vrši povratak na blok dokumenata. Slijedi specifikacija elemenata bloka.



Slika 6.7 – Korisničko sučelje bloka stavaka

6.7.1. VIRTUALNE VARIJABLE BLOKA

(a) CGUI = globalna varijabla, generička

(b) CAPP = globalna varijabla, generička

(c) LIB = globalna varijabla, generička

(d) CFRAME = varijabla bloka, generička

(e) CPANEL = varijabla bloka, generička

(f) P_DOK_ID = ulazni parametar bloka, numerička varijabla

(g) B_ARTIKLI = varijabla bloka, numerička, tipka "Artikli" u korisničkom sučelju

(h) B_IZLAZ = varijabla bloka, numerička, tipka "Izlaz" u korisničkom sučelju

Kao i u prethodnom bloku, varijable (a), (b), (c), (d) i (e) služe za potrebe korisničkog sučelja. Ulazni parametar (f) P_DOK_ID preuzima i čuva ID dokumenta iz poziva prethodnog bloka. Varijable (g) i (h) služe za preuzimanje odgovarajućih događaja akcije na varijabli.

6.7.2. DEFINICIJA TABLICA

(a) S = Glavna tablica STAVKA

(b) A = Vezana tablica ARTIKL

(c) L = Vezana tablica STAVKA

(d) SK = Vezana tablica, pogled SKLADISTE

Glavna tablica ovog bloka je S, odnosno STAVKA. Vezana tablica L služi za dohvaćanje najvećeg zapisanog ID-ja stavki, kako bi se automatikom mogao pridodijeliti novi. Vezana tablica A služi za dohvat podataka o artiklu koji je naveden u stavci, a vezana tablica SK, koja predstavlja pogled SKLADISTE, služi za dohvat informacije o trenutnom stanju artikla na skladištu sumirajući ulaze i izlaze tog artikla na svim stavkama svih dokumenata.

6.7.3. DEFINICIJA POLJA U TABLICAMA

(a) S_ID = Polje ID iz tablice S, numeričkog tipa

- dio je SELECT i INSERT SQL naredbi
- dio je identifikatora redka
- dio je ORDER BY klauzule

- ne smije se modificirati

- vidi se u tablici korisničkog sučelja pod nazivom "ID stavke"

(b) S_ID_DOKUMENTA = Polje ID_DOKUMENTA iz tablice S, numeričkog tipa

- dio je SELECT, INSERT i UPDATE SQL naredbi

- ne smije se modificirati

- vidi se u tablici korisničkog sučelja pod nazivom "ID dokumenta"

(c) S_SIFRA_ARTIKLA = Polje SIFRA_ARTIKLA iz tablice S, numeričkog tipa

- dio je SELECT, INSERT i UPDATE SQL naredbi

- vidi se u tablici korisničkog sučelja pod nazivom "Šifra artikla"

(d) S_ULAZ = Polje ULAZ iz tablice S

- dio je SELECT, INSERT i UPDATE SQL naredbi

- vidi se u tablici korisničkog sučelja pod nazivom "Ulaz"

(e) S_IZLAZ = Polje IZLAZ iz tablice S

- dio je SELECT, INSERT i UPDATE SQL naredbi

- vidi se u tablici korisničkog sučelja pod nazivom "Izlaz"

(f) L_ID = Polje ID tablice L, numeričkog tipa

- dio je SELECT SQL naredbe

- ne smije se mijenjati

- ima agregatnu funkciju MAX

(g) A_SIFRA = Polje SIFRA iz tablice A, numeričkog tipa

- dio je SELECT SQL naredbe

- ne smije se mijenjati

(h) A_NAZIV = Polje NAZIV iz tablice A, znakovnog tipa

- dio je SELECT SQL naredbe
- ne smije se mijenjati
- vidi se na korisničkom sučelju kao polje pod nazivom "Naziv artikla"

(i) A_JM = Polje JM iz tablice A, znakovnog tipa

- dio je SELECT SQL naredbe
- ne smije se mijenjati
- vidi se na korisničkom sučelju kao polje pod nazivom "JM"

(j) A_CIJENA = Polje CIJENA iz tablice A, numeričkog tipa

- dio je SELECT SQL naredbe
- ne smije se mijenjati
- vidi se na korisničkom sučelju kao polje pod nazivom "CIJENA"

(k) SK_SIFRA_ARTIKLA = Polje SIFRA_ARTIKLA iz tablice SK, numeričkog tipa

- dio je SELECT SQL naredbe
- ne smije se mijenjati

(l) SK_STANJE = Polje STANJE iz tablice SK, numeričkog tipa

- dio je SELECT SQL naredbe
- ne smije se mijenjati
- vidi se na korisničkom sučelju kao polje pod nazivom "STANJE"

Nabrojani elementi su polja iz tablica koje se koriste u bloku stavaka. Polja (a), (b), (c), (d) i (e) polja su tablice stavaka. Polje (a) je primarni ključ tablice, a polje (b) strani ključ prema tablici dokumenata iz prethodnog bloga. Polje (f) služi za dohvat maksimalnog ID-ja stavke kako bi se automatikom mogao izračunati slijedeći ID. Polja (g), (h), (i) i (j) polja su iz tablice

artikala te služe kako bi se mogli prikazati podaci o artiklu sa stavke. Polja (k) i (l) služe za dohvat i prikaz stanja skladišta iz pogleda SKLADISTE.

6.7.4. DEFINICIJE ELEMENATA UVJETA

(a) C1 = Uvjet s operacijom "=" sa varijablom S_ID_DOKUMENTA i izrazom "P_DOK_ID"

(b) C2 = Uvjet s operacijom "=" sa varijablom A_SIFRA i izrazom "S_SIFRA_ARTIKLA"

(c) C3 = Uvjet s operacijom "=" sa varijablom SK_SIFRA_ARTIKLA i izrazom "S_SIFRA_ARTIKLA"

Tablice S, A i SK su međusobno povezane, a potrebno je i filtriranje slogova tablice S kako bi bile vidljive samo stavke zadanog dokumenta iz bloka dokumenata. Uvjet C1 služi filtriranju tablice stavaka na način da u SELECT SQL element tablice S ugrađuje klauzulu "WHERE ID_DOKUMENTA = :X" gdje je :X vrijednost parametra P_DOK_ID. Uvjet C2 služi povezivanju tablice artikala sa tablicom stavaka, i u SELECT SQL element tablice A ugrađuje klauzulu "WHERE SIFRA = :Y" gdje je :Y vrijednost polja SIFRA_ARTIKLA iz tablice S. Slično tome, uvjet C3 povezuje tablicu stavaka sa pogledom SKLADISTE.

6.7.5. DEFINICIJA REKALKULACIJA

(a) R1 = Element rekalkulacije varijable S_ID_DOKUMENTA s inicijalizacijskim izrazom "P_DOK_ID"

Element rekalkulacije (a) na varijabli S_ID_DOKUMENATA služi inicijalizaciji vrijednosti varijable prilikom kreiranja novog redka. S obzirom na to da korisnik ne smije modificirati vrijednost varijable, ova rekalkulacija osigurava da svaka nova stavka pripada odgovarajućem dokumentu.

6.7.6. DEFINICIJA SQL ELEMENATA

(a) STM_S = SQL SELECT element tablice S, za početno dohvaćanje

(b) STM_S_UPDATE = SQL UPDATE element tablice S

(c) STM_S_DELETE = SQL DELETE element tablice S

(d) STM_S_INSERT = SQL INSERT element tablice S

(e) STM_L = SQL SELECT element za dohvaćanje sloga vezane tablice L

(f) STM_A = SQL SELECT element za dohvaćanje sloga vezane tablice A

(g) STM_SK = SQL SELECT element za dohvaćanje sloga vezane tablice SK

Pobrojani SQL elementi služe za rad s tablicama S, L, A i SK, pri čemu se tablici S dohvaćaju vrijednosti, ali i dodaju, modificiraju i brišu, dok se kod tablica L, A i SK samo dohvaćaju podaci, ali se na njima ne vrše nikakve izmjene.

6.7.7. DEFINICIJA METODE "LINK_REFRESH"

(a) SQL instrukcija STM_A, nivo 0

(b) SQL instrukcija STM_SK, nivo 0

Metoda LINK_REFRESH sastoji se od dvije instrukcije (a) i (b) koje dohvaćaju podatke tablica A i SK na trenunom slogu. Ovaj dohvat podataka se vrši na nekoliko mjesta u bloku, pa je zbog toga izdvojen u posebnu metodu.

6.7.8. DEFINICIJA DOGAĐAJA DOHVATA NA BLOKU

(a) SQL instrukcija STM_S, nivo 0

Događaj dohvata na bloku pokreće instrukciju za izvršavanje SQL elementa STM_S koji dohvaća slogove tablice S, uzimajući pri tom u obzir parametar P_DOK_ID i uvjet C1, zbog čega dohvaća samo stavke zadanog dokumenta. Izvršava se jednom za cijeli blok.

6.7.9. DEFINICIJA DOGAĐAJA DOHVATA NA REDKU

(a) Instrukcija evaluacije izraza "link_refresh()", nivo 0

Za svaki dohvaćeni redak izvršava se instrukcija evaluacije koja poziva metodu LINK_REFRESH, koja dohvaća podatke tablica A i SK za svaki redak posebno.

6.7.10. DEFINICIJA DOGAĐAJA POSLIJE VARIJABLE "S_SIFRA_ARTIKLA"

(a) Instrukcija evaluacije izraza "link_refresh()", nivo 0

(b) Instrukcija AKO s uvjetom da S_SIFRA_ARTIKLA nije prazna i da je S_SIFRA_ARTIKLA različita od A_SIFRA, nivo 0

(c) Poziv vanjske metode korisničkog sučelja s porukom "Ne postoji taj artikl!", nivo 1

(d) Povratna instrukcija koja vraća logičku vrijednost "ne", nivo 1

Ovaj događaj izvršava se nakon što korisnik unese šifru artikla na stavci (polje S_SIFRA_ARTIKLA). Instrukcija (a) najprije poziva metodu LINK_REFRESH dohvaćajući podatke tablica A i SK u skladu s novounesenom šifrom artikla. Instrukcija (b) provjerava da li je šifra artikla unešena, ali nije ispravno povezana s artiklom tablice A. Ukoliko je to istina, unešena je neispravna šifra artikla i o tome je potrebno obavijestiti korisnika. Instrukcije (c) i (d) obavljaju tu zadaću, gdje instrukcija (c) prikazuje poruku, a instrukcija (d) označava da je došlo do greške i da fokus ne smije napustiti polje S_SIFRA_ARTIKLA.

6.7.11. DEFINICIJA DOGAĐAJA PRIJE REDKA

(a) Instrukcija evaluacije izraza "link_refresh()", nivo 0

Prije ulaska u redak izvršava se instrukcija evaluacije koja poziva metodu LINK_REFRESH, koja dohvaća podatke tablica A i SK.

6.7.12. DEFINICIJA DOGAĐAJA POSLIJE REDKA

(a) Instrukcija pridruživanja koja varijabli S_ULAZ pridružuje vrijednost 0 ako S_ULAZ nije unešen, nivo 0

(b) Instrukcija pridruživanja koja varijabli S_IZLAZ pridružuje vrijednost 0 ako S_IZLAZ nije unešen, nivo 0

(c) SQL instrukcija STM_S_DELETE, nivo 0, ako je slog označen za brisanje

(d) SQL instrukcija STM_S_UPDATE, nivo 0, ako je slog tablice S označen za modificiranje

(e) Instrukcija AKO s uvjetom da je slog označen za umetanje, nivo 0

(f) SQL Instrukcija STM_L, nivo 1

(g) Instrukcija pridruživanja koja varijabli S_ID pridružuje vrijednost izraza "L_ID+1", nivo 1

(h) SQL instrukcija STM_S_INSERT, nivo 1

(i) COMMIT instrukcija, nivo 0

Događaj se izvršava prilikom izlaska iz redka i služi zapisivanju promjena podataka u redku u bazu podataka. Instrukcije (a) i (b) osiguravaju da u varijablama S_ULAZ i S_IZLAZ sigurno postoji broj kako bi pogled SKLADISTE mogao ispravno funkcionirati. Instrukcija (c) briše redak ako je označen za brisanje, dok instrukcija (d) modificira vrijednost redka u bazi ako je označen za promjenu. Instrukcija (e) provjerava da li je redak potrebno umetnuti u bazu. Ako je odgovor potvrdan izvršavaju se instrukcije (f), (g) i (h) od kojih (f) dohvaća maksimalni ID stavke, instrukcija (g) taj maksimalni broj uvećava za 1 i tu vrijednost pridružuje varijabli S_ID, a instrukcija (h) izvršava SQL element STM_S_INSERT. Instrukcija (i) izvršava COMMIT naredbu baze podataka.

6.7.13. DEFINICIJA DOGAĐAJA AKCIJE NA VARIJABLI "B_ARTIKLI"

(a) Instrukcija poziva bloka, koja izvršava poziv bloka artikala, nivo 0

(b) Instrukcija evaluacije koja poziva metodu LINK_REFRESH, nivo 0

Instrukcija (a) poziva izvršenje bloka artikala, a nakon što se kontrola vrati u blok stavaka, izvršava se instrukcija (b) koja poziva LINK_REFRESH metodu, koja dohvaća podatke tablica A i SK. Ovo dohvaćanje se vrši zbog potencijalne mogućnosti da je na artiklu došlo do promjene podataka, na primjer naziva artikla, pa bi se ta promjena trebala odmah vidjeti i na stavci dokumenta.

6.7.14. DEFINICIJA DOGAĐAJA AKCIJE NA VARIJABLI "B_IZLAZ"

(a) Instrukcija vanjskog poziva metode korisničkog sučelja, koja izvršava zatvaranje unosne forme, nivo 0

Instrukcija (a) poziva metodu zatvaranja unosne forme biblioteke korisničkog sučelja, čime se kontrola predaje prethodnoj formi, odnosno bloku dokumenata.

6.8. BLOK ARTIKALA

Blok artikala služi za pregled i unos šifarnika artikla. Može se pozvati iz bloka dokumenata ili iz bloka stavaka, a nema ulazne parametre. Korisničko sučelje sastoji se od tablice s podacima o artiklima i tipke za izlaz. Slijedi specifikacija elemenata bloka.

Slika 6.8 – Korisničko sučelje bloka artikala

6.8.1. VIRTUALNE VARIJABLE BLOKA

(a) CGUI = globalna varijabla, generička

(b) CAPP = globalna varijabla, generička

(c) LIB = globalna varijabla, generička

(d) CFRAME = varijabla bloka, generička

(e) CPANEL = varijabla bloka, generička

(f) B_IZLAZ = varijabla bloka, numerička, tipka "Izlaz" u korisničkom sučelju

Varijable (a), (b), (c), (d) i (e) služe za potrebe korisničkog sučelja. Varijabla (f) služi za preuzimanje odgovarajućeg događaja akcije na varijabli.

6.8.2. DEFINICIJA TABLICA

(a) A = Glavna tablica ARTIKL

Bloku artikala nalazi se samo definicija tablice ARTIKL. Nema potrebe za drugim tablicama s obzirom na to da se ne dohvaćaju podaci iz drugih tablica, a šifra artikla se ne dodjeljuje automatikom nego korisničkim unosom.

6.8.3. DEFINICIJA POLJA U TABLICAMA

(a) A_SIFRA = Polje SIFRA iz tablice A, numeričkog tipa

- dio je SELECT i INSERT SQL naredbi
- dio je identifikatora redka
- dio je ORDER BY klauzule
- smije se modificirati samo ako je novi redak
- vidi se u tablici korisničkog sučelja pod nazivom "Šifra artikla"

(b) A_NAZIV = Polje NAZIV iz tablice A, znakovnog tipa

- dio je SELECT, UPDATE i INSERT SQL naredbi
- vidi se u tablici korisničkog sučelja pod nazivom "Naziv"

(c) A_JM = Polje JM iz tablice A, znakovnog tipa

- dio je SELECT, UPDATE i INSERT SQL naredbi
- vidi se u tablici korisničkog sučelja pod nazivom "JM"

(d) A_CIJENA = Polje CIJENA iz tablice A, numeričkog tipa

- dio je SELECT, UPDATE i INSERT SQL naredbi
- vidi se u tablici korisničkog sučelja pod nazivom "Cijena"

U bloku su definirana sva polja tablice ARTIKL. Polje (a) šifra artikla je primarni ključ, ali ga korisnik smije slobodno unijeti, ali samo dok je redak još nije zapisan u bazu. Polja (b), (c) i (d) mogu se slobodno unositi bez ikakvih ograničenja.

6.8.4. DEFINICIJA SQL ELEMENATA

(a) STM_A = SQL SELECT element tablice A, za početno dohvaćanje slogova

(b) STM_A_UPDATE = SQL UPDATE element tablice A

(c) STM_A_DELETE = SQL DELETE element tablice A

(d) STM_A_INSERT = SQL INSERT element tablice A

Glavna tablica ima SQL elemente za početno dohvaćanje slogova (a), za modificiranje sloga (b), za brisanje sloga (c) i konačno, za umetanje sloga (d) u bazu podataka.

6.8.5. DEFINICIJA DOGAĐAJA DOHVATA NA BLOKU

(a) SQL instrukcija STM_A, nivo 0

Poziv SQL instrukcije STM_A ima za posljedicu dohvat svih slogova tablice A, te njihovo zapisivanje u spremnik bloka.

6.8.6. DEFINICIJA DOGAĐAJA POSLIJE REDKA

(a) SQL instrukcija STM_A_DELETE, nivo 0, ako je slog označen za brisanje

(b) SQL instrukcija STM_A_UPDATE, nivo 0, ako je slog označen za modificiranje

(c) SQL instrukcija STM_A_INSERT, nivo 0, ako je slog označen za umetanje

(d) COMMIT instrukcija, nivo 0

Događaj poslije redka služi za zapisivanje promjena u bazu podataka. Instrukcija (a) odnosi se na brisanje sloga ako je potrebno, instrukcija (b) na modificiranje, a instrukcija (c) na umetanje sloga u bazu. Instrukcija (d) izvršava COMMIT naredbu baze podataka.

6.8.7. DEFINICIJA DOGAĐAJA AKCIJE NA VARIJABLI "B_IZLAZ"

(a) Instrukcija vanjskog poziva metode korisničkog sučelja, koja izvršava zatvaranje unosne forme, nivo 0

Ova instrukcija poziva metodu zatvaranja unosne forme biblioteke korisničkog sučelja, a programska kontrola se predaje bloku koji je pozvao blok artikala.

7. ZAKLJUČAK

Problematika korištenja relacijskih baza podataka, bilo da je riječ o modeliranju baza ili o internoj funkcionalnosti, pokrivena je brojnim znanstvenim istraživanjima. Isto tako, dobro je pokrivena problematika upravljanja projektima i razvojnim timovima. Problematika kvalitete sustava za razvoj aplikacija usmjerenih na baze podataka, s druge strane, gotovo da ne postoji. Ovaj rad predstavlja tek jedan korak u tom smjeru.

U poglavlju 2. dan je pregled specifičnosti korištenja i razvoja aplikacija usmjerenih na baze podataka, te osobina koje bi sustavi za razvoj takvih aplikacija morali imati. Napravljen je izbor nekoliko značajnih i relevantnih sustava za razvoj aplikacija usmjerenih na baze koji se koriste u praksi, te napravljena ocjena njihove kvalitete s obzirom na željene osobine (prilog 1). Pregled specifičnosti i osobina ostavlja prostora za daljnja unaprjeđenja, a popis postojećih sustava za razvoj aplikacija usmjerenih na baze može biti i značajno dulji i kompletniji. Za to bi bilo potrebno provesti jedno posebno opsežno istraživanje koje bi rezultiralo zasebnim znanstvenim radom.

U ovom radu je dan prijedlog novog sustava za razvoj aplikacija usmjerenih na baze podataka u obliku detaljne apstraktnog modela elemenata iz kojih se može kreirati konkretna implementacija sustava, odnosno konkretan razvojni sustav.

Ovisno o dubini implementacije, on ima potencijal zadovoljiti sve željene osobine navedene u poglavlju 2.

Tablica 7 - Kratka usporedba željenih osobina razvojnog sustava i predloženog sustava

RB	ŽELJENA OSOBINA	KOMENTAR
1.	Dinamičko kreiranje korisničkog sučelja i logike aplikacije	S obzirom na to da se cijeli apstraktni model temelji na objektno orjentiranom pristupu moguće je definirati nove blokove i pripadajuće elemente u toku izvršavanja aplikacije
2.	Nezavisnost korisničkog sučelja od logike aplikacije	Elementi programske logike odvojeni su od korisničkog sučelja, pa se samim tim mogu i slobodno izmjenjivati
3.	Izmjenjivi repozitorij elemenata aplikacije	U poglavlju 5 opisan je sustav repozitorija kojim se mogu smjestiti definicije svih elemenata (izvorni kod) u relacijsku bazu podataka. Naravno, moguće je i drugačije zapisati repozitorij (XML, tekstualne datoteke i slično)
4.	Korisnički upiti i obrade nad repozitorijem	S obzirom na to da je repozitorij zapisan u relacijskoj bazi i potpuno otvoren, moguće mu je pristupiti SQL-om, što znači izvršiti korisničke upite i obrade.
5.	Prilagođenost događajnog sustava logici unosa podataka	Događajni sustav se ne temelji na korisničkom sučelju već na logici unosa, pa tako na primjer, postoje elementi događaja prije ulaska u redak i nakon izlaska iz redka i drugi.
6.	Kontrola procesa unosa podataka	Kontrola ulaska i izlaska je potpuna, jer metode događaja imaju povratne vrijednosti koje određuju da li su uvjeti zadovoljeni ili nisu, što opet utječe na fokus korisničkog sučelja.
7.	Fleksibilna distribucija aplikacije	Ovo ovisi o konkretnoj implementaciji sustava, ali potencijalno je moguće napraviti razne modele distribucije.
8.	Otvorenost sustava za razvoj aplikacija usmjerenih na bazu podataka	Transparentnost apstraknog modela je potpuna, a o implementaciji ovisi da li će takva i ostati. Implementacijom je moguće kreirati kako otvoreni, tako i zatvoreni sustav.
9.	Ujednačena infrastruktura	Ova infrastruktura također ovisi o konkretnoj implementaciji, ali nema razloga da ne bude provedena, s obzirom na to da je podržana elementima apstraktnog modela.
10.	Mogućnost pristupa sql-u na niskom nivou	SQL element apstraktnog modela omogućuje pisanje direktnog SQL-a.

Ovakav apstraktni model, iako zadovoljava sve željene osobine, ipak ne predstavlja apsolutni kraj istraživanja, te ga je moguće dodatno unaprijediti. Smjer budućih istraživanja treba tražiti

u odvajanju elemenata apstraktnog modela u dvije kategorije: klijentsku i serversku. Ako govorimo o web serverima ili aplikacijskim serverima, tada bi neki elementi mogli egzistirati na klijentskoj strani u obliku recimo javascript koda, dok bi neki drugi elementi mogli postojati na serveru, te bi njihovo izvršavanje moglo biti sinkronizirano. Drugi smjer budućih istraživanja može se tražiti u povećanoj fleksibilnosti elemenata metode i instrukcije. Nema razloga da se elementi instrukcije ne prošire, primjerice, predikatnom logikom ili nekim drugim naprednim oblikom pisanja programske logike, ili čak kodom pisanim nekim od klasičnih programskih jezika, čime bi se nesumnjivo u nekim situacijama dobilo na efikasnosti izvršavanja aplikacije. Apstraktni model, zasigurno, predstavlja dobru osnovu za daljnja istraživanja.

8. LITERATURA

1. Agarwal, V., "Beginning C# 5.0 Databases", Apress, 2012.
2. Ahmed, R., "Create Rapid Web Applications Using Oracle Application Express", Riaz Ahmed, 2011.
3. Allen, G., Owens, M., "The Definitive Guide to SQLite", Apress, 2010.
4. Allen, C., Creary, C., Chatwin, S., "Introduction to Relational Databases", McGraw-Hill Osborne Media, 2003.
5. Andrews, J., "Co-verification of Hardware and Software for ARM SoC Design (Embedded Technology)", Newnes, 2004.
6. Azarmsa, R., "Educational Computing: Principles and Applications", Educational Technology Publications, 1991.
7. Banks, R., "Visual Studio 2012 Cookbook", Packt Publishing, 2012.
8. Baum, D., "Moving from Punch Cards to Object Orientation: The Evolution of Application Development", InfoWorld magazine vol. 14 no. 36, InfoWorld Media Group, 1992.
9. Beck, K., "Extreme Programming Explained: Embrace Change", Addison-Wesley Professional, 1999.
10. Bench-Capon, T., Soda, G., Tjoa, A. M., "Database and Expert Systems Applications: 10th International Conference, DEXA'99, Florence, Italy, August 30 - September 3, 1999, Proceedings (Lecture Notes in Computer Science)", Springer, 1999.
11. Bidgoli, H., "The Internet Encyclopedia, Volume 3", Wiley, 2003.
12. Birrell,N.D.,Ould,M.A.,"A Practical Handbook for Software Development",Cambridge University Press, 1988.
13. Brooks, F. P., "The Mythical Man-Month". Addison-Wesley Publishing Company, 1995.

14. Brooks, F. P., "No Silver Bullet — Essence and Accident in Software Engineering". Proceedings of the IFIP Tenth World Computing Conference,1986.

15. Cimolini, P., Cannell, K., "Agile Oracle Application Express", Apress, 2012.

16. Committee on the Past and Present Contexts for the Use of Ada in the Department of Defense, National Research Council, "Ada and Beyond: Software Policies for the Department of Defense", National Academies Press, 1997.

17. Connolly, T. M., Begg, C. E., "Database Systems: A Practical Approach to Design, Implementation and Management", Addison Wesley, 2004.

18. Darwin, I. F., "Java Cookbook", O'Reilly Media, 2004.

19. Date, C. J., "SQL and Relational Theory: How to Write Accurate SQL Code", O'Reilly Media Inc., 2009.

20. Date, C. J., "An Introduction to Database Systems", Sixth Edition, Assison-Wesley Publishing Company, 1995.

21. Del Sole, A., "Microsoft Visual Studio LightSwitch Unleashed", Pearson Education, 2012.

22. Desbiens, F., Moskovits, P., Weckerle, P., "Oracle WebCenter 11g Handbook: Build Rich, Customizable Enterprise 2.0 Applications", Oracle Press, 2009.

23. Downey, A., "Python for Software Design: How To Think Like a Computer Scientist", Cambridge University Press, 2009.

24. Eckel, B., "Thinking in Python: Design Patterns and Problem-Solving Techniques", MindView, 2001.

25. Farrell, J., "An Object-Oriented Approach to Programming Logic and Design", Cengage Learning, 2012.

26. Foord, M., Muirhead, C., "IronPython in Action", Manning Publications, 2009.

27. Fowler, M., "Patterns of Enterprise Application Architecture", Addison-Wesley, 2003.

28. Fox, T., Scott, J., Spendolini, S., "Pro Oracle Application Express 4", Apress, 2011.

29. Gault, D., Cannell, K., Cimolini, P., D'Souza, M., Hilaite, T., "Beginning Oracle Application Express 4", Apress, 2011.

30. Goodson, J., Steward, R. A., "The Data Access Handbook: Achieving Optimal Database Application Performance and Scalability", Prentice Hall, 2009.

31. Gousset, M., Keller, B., Woodward, M., "Professional Application Lifecycle Management with Visual Studio 2012", Jon Wiley and Sons, 2012.

32. Greenwald, R., "Beginning Oracle Application Express", Wrox, 2008.

33. Halvorson, M., "Microsoft Visual Basic 2010 Step by Step", Microsoft Press, 2010.

34. Hammond, M., Robinson, A., "Python Programming on WIN32: Help for Windows Programmers", O'Reilly Media, 2000.

35. Haralabidis, N., "Oracle JDeveloper 11gR2 Cookbook", Packt Publishing, 2012.

36. Hentzen, W., "MySQL Client-Server Applications with Visual FoxPro", Hentzenwerke Publishing, 2007.

37. Hubbard, J. R., "Schaum's outline of theory and problems of programming with C++", McGraw-Hill, 2000.

38. ISRD Group, "Introduction to Database Management Systems", McGraw-Hill Companies, 2006.

39. Jalote, P., "An Integrated Approach to Software Engineering (Texts in Computer Science)", Springer; 3rd edition, 2005.

40. Jones, C., Drake, F., "Python and XML", O'Reilly, 2002.

41. Kan, S. H.,"Metrics and Models in Software Quality Engineering", Second Edition, Addison-Wesley Professional, 2002.

42. Kann, C. W., "Creating Components: Object Oriented, Concurrent, and Distributed Computing in Java", Auerbach Publications, 2004.

43. Kedar, S., "Database Management Systems", Technical Publications Pune, 2008.

44. Kernighan, B., Ritchie, D., "Programming language C", Prentice Hall, 1988.

45. Koletzke, P., Mills, D., "Oracle JDeveloper 10g for Forms & PL/SQL Developers: A Guide to Web Development with Oracle ADF", Mc Graw Hill, 2006.

46. Kreibich, J., "Using SQLite", O'Reilly, 2010.

47. Lans, R., "The SQL Guide to SQLite", Lulu, 2009.

48. Liang, Y. D., "Introduction to Java Programming, Comprehensive", Prentice Hall, 2010.

49. Liberty, J., MacDonald, B., "Learning C# 3.0", O'Reilly Media, 2008.

50. Lobel, L., Brust, A., "Programming Microsoft SQL Server 2012", Microsoft Press, 2012.

51. Mace, S., "Vendors Take on Market Leaders", InfoWorld magazine vol. 8 no. 18, InfoWorld Media Group, 1986.

52. Magic Software, "Magic xpa Application Platform: The Smarter Way to Build and Deploy Business Applications", Magic Software 2012. http://www.magicsoftware.com/magic-xpa-application-platform#Overview (7.10.2012).

53. Mertz, D., "Text Processing in Python", Addison-Wesley, 2003.

54. Microsoft Corporation, "What's new in Visual Studio 2012", Microsoft Corporation, 2012. http://www.microsoft.com/visualstudio/eng/whats-new (7.10.2012).

55. Mills, D., Koletzke, P., Roy-Faderman, A., "Oracle JDeveloper 11g Handbook: A Guide to Fusion Web Development", Mc Graw Hill, 2009.

56. Morton, K., deHaan, L., Gorman, T., Jargensen, I., Fink D., "Beginning Oracle SQL", Apress, 2009.

57. O'Brien, J., Marakas, G., "Introduction to Information Systems", McGraw-Hill, 2010.

58. O'Donnell, T.J., "Design and Use of Relational Databases in Chemistry", CRC Press, 2008.

59. O'Neil, P., O'Neil, E., "Database: Principles, Programming, and Performance", Morgan Kaufmann, 2000.

60. Olsen, D. R., “Developing User Interfaces”, Morgan Kaufmann Publishers Inc., 1998.

61. Oracle, "Oracle JDeveloper and Oracle ADF Technical Resources", Oracle JDeveloper 2012. http://www.oracle.com/technetwork/developer-tools/jdev/learnmore/index.html (7.10.2012).

62. Oracle, "Oracle Application Express End User's Guide", Oracle, 2012 (2012b). http://docs.oracle.com/cd/E23903_01/doc/doc.41/e26811/toc.htm (7.10.2012).

63. Oualline, S., "Practical C++ Programming", O'Reilly Media, 2003.

64. Pandey, H. M., "Design Analysis and Algorithms", University Science Press, 2008.

65. Parsons, J. J., Oja, D., "New Perspectives on Computer Concepts 2010: Comprehensive", Course Technology, 2008.

66. Pfleeger, S. L., Atlee J, M. "Software Engineering: Theory and Practice (4th Edition)", Prentice Hall, 2009.

67. Ponniah, P., “Data Warehousing Fundamentals For IT Professionals”, John Wiley & Sons, 2010.

68. Precord, C., "wxPython 2.8 Application Development Cookbook", Packt Publishing, 2010.

69. Price, J., "Oracle Database 11g SQL (Osborne ORACLE Press Series)", McGraw-Hill Osborne Media, 2007.

70. Python Project., "About", Python Project, 2012. http://www.python.org/about/ (22.09.2012).

71. Rappin, N., Dunn, R., "wxPython in Action", Manning Publications, 2006.

72. Raymond, E. S., "Why Python?", Linux Journal, 30.04.2000. http://www.linuxjournal.com/article/3882 (22.09.2012).

73. Raymond, E., "The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary", O'Reilly, 2001.

74. Rob, P., Coronel, C., "Database Systems: Design, Implementation, and Management", Thomson Course Technology, 2009.

75. Qin, Z., Xing, J., Zheng X., "Software Arhitecture", Springer, 2008.

76. Schmidt, J. W., Stogny, A. A., "Next Generation Information System Technology: First International East/West Data Base Workshop, Kiev, USSR, October 9-12, 1990. Procceedings (Lecture Notes in Computer Science)", Springer, 1991.

77. Schuppenhauer, H., "The Sorcerer's Apprentice V10: 1. Lessons", Cabrita Software, 2007.

78. Scott, M. L., "Programming Language Pragmatics". Morgan Kaufmann, 2009.

79. Selby, R. W., "Software Engineering: Barry W. Boehm's Lifetime Contributions to Software Development, Management, and Research (Practitioners)", Wiley-IEEE Computer Society Pr, 2007.

80. Sharma, P., "Software Engineering", APH Publishing Corporation, 2004.

81. Shroff, G., "Enterprise Cloud Computing: Technology, Architecture, Applications", Cambridge University Press, 2010.

82. Smart, J., Hock, K., Csomor, S., “Cross-Platform Gui Programming With WxWidgets”, Pearson Education Inc., 2006.

83. Sommerville, I.,"Software Engineering", Fifth Edition. Addison-Wesley Publishing Company, 1995.

84. SQLite Project, "About SQLite", SQLite Project, 2012 (2012a). http://www.sqlite.org/about.html (22.09.2012).

85. SQLite Project, "Well-Known Users of SQLite", SQLite Project, 2012 (2012b) http://www.sqlite.org/famous.html (22.09.2012).

86. Stair, R., Reynolds, G., "Principles of Information Systems", Course Technology, 2009.

87. Stephens, R., "Visual Basic 2012 Programmer's Reference", Jon Wiley and Sons, 2012.

88. Svenk, G., "Object-Oriented Programming: Using C++ for Engineering and Technology", Delmar Learning, 2003.

89. Tipton, H. F., Krause, M., "Information Security Management Handbook, Volume 2", Auerbach Publications, 2007.

90. Tkalac, S., "Relacijski model podataka", Društvo za razvoj informacijske pismenosti, 1993.

91. Troelsen, A., "Pro C# 5.0 and the .NET 4.5 Framework", Apress, 2012.

92. Tymann, P., Reynolds, C., "Schaum's Outline of Principles of Computer Science", McGraw-Hill, 2008.

93. Venners, B., "The Making of Python: A Conversation with Guido van Rossum", Artima Inc., 2003. http://www.artima.com/intv/python.html (22.09.2012).

94. Vohra, D., "Processing XML documents with Oracle JDeveloper 11g", Packt Publishing, 2009.

95. Vohra, D., "Ajax in Oracle JDeveloper", Springer, 2008.

96. Wei, J., "Android Database Programming", Packt Publishing, 2012.

97. Wilson, C., "User Experience Re-Mastered: Your Guide to Getting the Right Design", Elsevier Inc., 2010.

98. Wohlin, C., Runeson, P., Höst, M., "Experimentation in Software Engineering: An Introduction (International Series in Software Engineering)", Springer, 1st edition, 1999.

99. wxWidgets Project, "About the wxWidgets Project", wxWidgets Project, 2012 (2012a). http://wxwidgets.org/about/ (22.09.2012).

100. wxPython Project, "What is wxPython?", wxPython Project, 2012 (2012b). http://www.wxpython.org/what.php (22.09.2012).

101. Yeung, A. K. W., Hall, G. B., "Spatial Database Systems: Design, Implementation and Project Management (GeoJournal Library)", Springer, 2007.

102. Zehoo, E., "Oracle Application Express 4 Recipes", Apress, 2011.

103. Ziade, T., "Expert Python Programming", Packt Publishing, 2008.

PRILOG 1: IZVORNI KOD IMPLEMENTACIJE APSTRAKTNOG MODELA

MODUL:__init__.py

```
import lib

from _base_datatype import generic,string,number,boolean,datetime
from _base_expression import ConstExpr,EvalExpr,LambdaExpr
from _base_field import Field, DbField, RowField, BlockField, GlobalField, \
    IOParameter, InputParameter, OutputParameter, Recalc
from _base_table import MainTable, LinkTable
from _base_condition import OPERATORS, \
    op_eq,op_lt,op_le,op_gt,op_ge,op_ne,op_isnull,op_in,op_between,op_like, \
    OpCond,ClauseCond,ExprCond
from _base_block import Block, Application
from _base_sqlstatement import SQLExpr, DirectSQL,\
                    SQLSelect, SQLSelectRefetch,\
                    SQLLock, SQLLockRefetch, \
                    SQLInsertRow, SQLUpdateRow,SQLDeleteRow
from _base_join import InnerJoin, LeftOuterJoin
from _base_code import Method, UpdateInstr, EvaluateInstr, DumpInstr, CallExtInstr, \
                IfInstr, ElseInstr, WhileInstr, \
                ExecSQLStatementInstr, CallInstr, \
                CommitInstr, RollbackInstr, ReturnInstr, \
                EventBlockBefore, EventBlockAfter, \
                EventRecordBefore, EventRecordAfter, \
                EventFieldBefore, EventFieldAfter, EventFieldAction, \
                EventExternal, EventBlockDescriptor,\
                EventBlockFetch,EventRecordFetch, \
                EventFieldChange, EventError, EventBlockExit
```

MODUL: _base_block.py

```
import common

from _base_block_buffer import Buffer
from _base_field import Field
from _base_expression import Expression
```

```
import sys, traceback

class Block:
   ## KLASA Blok
   def __init__(self,app,name,parent=None,**kw):
      self.application = app
      ## ATR Aplikacija
      self.name = name
      ## ATR Ime
      self.parent = parent
      ## ATR Roditelj
      self.parentdict = {}
      ## ATR Roditelji_po_imenu
      if parent is not None:
         parent.childlist.append(self)
         for x in parent.parentdict.keys():
            self.parentdict[x] = parent.parentdict[x]
         self.parentdict[parent.name] = parent
      self.childlist = []
      ## ATR Djeca
      self.fieldablelist = []
      ## ATR ???
      self.buffer = Buffer(self)
      ## ATR Spremnik_podataka
      self.tablelist = []
      ## ATR Lista_tablica
      self.tablealiasdict = {}
      ## ATR Tablice_po_nadimku
      self.maintable = None
      ## ATR Sredisnja_tablica
      self.fieldlist = []
      ## ATR Lista varijabli
      self.fielddict = {}  # field name dictionary. key = (scope,name). def. Scope=None
      ## ATR Varijable_po_imenu
      self.firstfield = None # First field to jump in row
      ## ATR Prva_varijabla
      self.tablefieldlist = {None:[]} # key(dbtable),list of fields,key = None virtuals
      ## ATR Polja_tablice
      self.expressionlist = []
      ## ATR Lista izraza
      self.conditionlist = []
      ## ATR Lista uvjeta
```

```
    self.sqlstatementlist = []
    ## ATR Lista SQL elemenata
    self.joinlist = []
    ## ATR Lista_veza
    self.tablejoindict = {}
    ## ATR Veze_tablice
    self.righttablejoindict = {}
    ## ATR Vanjske_veze_tablice
    self.methodlist = []
    ## ATR Lista_metoda
    self.methoddict = {}
    ## ATR Metode_po_imenu
    self.eventlist = []
    ## ATR Lista_dogadjaja
    self.eventdict = {}
    ## ATR Dogadjaji_po_kljucu_registracije
    self.block_before_executed = False # is block_before event executed?
    ## ATR Dogadjaj_blok_prije_izvrsen
    self.rowref = common.ROWREF_NO_RECORD
    ## ATR Trenutni_identifikator_redka
    self.fieldref = None
    ## ATR Trenutni_identifikator_varijable
    self.focus_list_diff = []
    self.focus_list_all = []
    self.delete_row_request = None # if field then, requested at field
    ## ATR Zahtjev_za_brisanje_redka
    self.after_return_value = None
    ## ATR Vrijednost_dogadjaja_poslije
    self.debug = False
    self.prompt = False # if step is paused and prompted
    self.stepcounter = 1
    self.msgcounter = 0
    self.running = True # if False block should stop
    ## ATR Blok_aktivan
    self.recalclist = []
    ## ATR Lista_rekalkulacija
def __str__(self):
    return '<Block %s>'%self.name
def __repr__(self):
    return self.__str__()
def output(self,message):
    if self.debug:
```

```
        self.msgcounter += 1
        print "%s %d:"%(self,self.msgcounter),message
def show_prompt(self,instruction):
    # Step pause
    if self.prompt:
        print "%s Step %d:"%(self,self.stepcounter)
        print "  Instruction %s executed."%instruction
        while True:
            a = raw_input('Enter to continue> ')
            if len(a)==0: break
            try:
                print self.evaluate(a)
            except:
                print "Error!"
def close_statements(self):
    for sqlstatement in self.sqlstatementlist:
        sqlstatement.close()
def isfield(self,object):
    return object in self.fieldlist
def isexpr(self,object):
    return object in self.exprlist
def rebuild(self):
    ## MET Izgradi
    try:
        for s in self.sqlstatementlist:
            s.rebuild()
        self.buffer.rebuild()
    except:
        print "ERROR REBUILD."
def blockdict(self,rowref):
    return BlockDict(self,rowref)
def findevent(self,*key):
    ## MET Nadji_dogadjaj
    return self.eventdict.get(key,None)
def evaluate(self,string_expression):
    return eval(string_expression,{},self.blockdict(self.rowref))
def findfield(self,fieldname):
    # returns field object
    return self.fielddict.get((self.buffer.get_scope(),fieldname),\
        self.fielddict.get((None,fieldname),None))
def getvalue(self,object):
    # evaluate expression or field or constant within block context
```

```
    if isinstance(object,Field) or isinstance(object,Expression):
        return object.get()
    return object
def fetchdata(self):
    ## MET Dohvati_podatke
    self.running = True
    # clear db data
    self.buffer.clear_db_data()
    # call fetch block event
    evt = self.findevent(common.EVT_BLOCK_FETCH)
    if evt is not None:
        evt.call()
    # for each row call fetch row event
    evt = self.findevent(common.EVT_RECORD_FETCH)
    if evt is not None:
        for rowref in self.buffer.rowreforder:
            self.rowref = rowref
            evt.call()
    # make first rowref
    if len(self.buffer.rowreforder)>0:
        self.rowref = self.buffer.rowreforder[0]
def callevent(self,eventtype,*arglist,**argdict):
    ## MET Pozovi_dogadjaj
    evt = self.findevent(eventtype)
    if evt is not None:
        return evt.call(*arglist,**argdict)
def start(self,*arglist,**argdict):
    ## MET Start
    self.output('Start block %s'%str(self.name))
    self.output('Param list: %s'%str(arglist))
    self.output('Param dict: %s'%str(argdict))
    # args to var
    i = 0
    for p in self.fieldlist:
        if p.param_in:
            if len(arglist)>i:
                p.set(self.getvalue(arglist[i]))
            if argdict.has_key(p.name):
                p.set(self.getvalue(argdict[p.name]))
            self.output('Parameter %s value = %s'%(p.name,str(p.get())))
            i = i + 1
    # process init events for block varijables
```

```
    for field in self.fieldlist:
        # init if field is rowfield or dbfield
        if not field.rowbuffer and field.dbtable is None:
            self.buffer.recalc_init(None,field)
    # execute fetch data
    self.fetchdata()
    # if has block descriptor event execute it
    e = self.findevent(common.EVT_BLOCK_DESCRIPTOR)
    if e is not None:
        self.output('Block descriptor exists')
        e.call()
    else:
        self.output('Block descriptor does not exists')
        # calling default block handler
        pass
    retval = None
    # Call of exit event
    e = self.findevent(common.EVT_BLOCK_EXIT)
    if e is not None:
        self.output('Block exit exists')
        retval = e.call()
    self.output('End of block "%s" execution'%str(self.name))
    self.output('Block return value is "%s"'%str(retval))
    return retval
def call(self,blockname,*l,**kw):
    ## MET Pozovi_blok
    # Start another block by name
    return self.application.call(blockname,*l,**kw)
def subcall(self,childblockname,*l,**kw):
    ## MET Pozovi_podredjeni_blok
    # Start child block by name
    pass
def rowref2rownum(self,rowref):
    ## MET Identifikator_redka_u_redni_broj_redka
    if len(self.buffer.rowreforder)==0:
        return 0
    return self.buffer.rowreforder.index(rowref)
def rownum2rowref(self,rownum):
    ## MET Redni_broj_redka_u_identifikator_redka
    if len(self.buffer.rowreforder)==0:
        return common.ROWREF_NO_RECORD
    elif rownum+1>len(self.buffer.rowreforder):
```

```
            return common.ROWREF_NO_RECORD
        return self.buffer.rowreforder[rownum]
    def get_pos(self):
        ### M Dohvati_poziciju
        return self.rowref2rownum(self.rowref),self.fieldref
    def get_ref(self):
        ### M Dohvati_identifikator
        return self.rowref,self.fieldref
    def set_pos(self,rownum,fieldref):
        ### M Postavi_poziciju
        self.rowref = self.rownum2rowref(rownum)
        self.fieldref = fieldref
    def process_action(self,fieldref):
        ### M Obradi_akciju
        self.output('FIELD ACTION EVENT %s'%str(fieldref))
        e = self.findevent(common.EVT_FIELD_ACTION,fieldref)
        if e is not None:
            e.call()
    def process_cancel(self,rowref,fieldref):
        ### M Obradi_odustajanje
        self.output('CANCEL AT %s'%str(fieldref))
        if rowref <> common.ROWREF_NO_RECORD:
            self.buffer.restore_field_values(rowref)
    def process_delete(self,rowref,fieldref):
        ### M Obradi_brisanje
        print "BLOCK: DELETE ROW AT",rowref,fieldref
        if rowref <> common.ROWREF_NO_RECORD:
            index = self.buffer.rowreforder.index(rowref)
            self.call_field_after()
            self.delete_row_request = fieldref
            self.call_record_after()
            # if deleted change rowref
            self.change_deleted_rowref(index)
            self.delete_row_request = None
    def change_deleted_rowref(self,index):
        ### M Promijeni_izbrisani_identifikator
        if self.rowref not in self.buffer.rowreforder:
            # determine new rowref if deleted
            if len(self.buffer.rowreforder)>0:
                if len(self.buffer.rowreforder)<=index:
                    self.rowref = self.buffer.rowreforder[-1]
                else:
```

```
            self.rowref = self.buffer.rowreforder[index]
         self.fieldref = self.get_first_fieldref()
      else:
         self.rowref = common.ROWREF_NO_RECORD
def get_first_fieldref(self):
   ### M Dohvati_prvu_varijablu
   if self.firstfield is None:
      return self.fieldlist[0]
   return self.firstfield
def process_insert(self,oldrowref,oldfieldref):
   ### M Obradi_umetanje
   print "BLOCK: INSERT ROW AT",oldrowref,oldfieldref
   if oldrowref == common.ROWREF_NO_RECORD:
      self.rowref = common.ROWREF_NEW_RECORD
      self.fieldref = self.get_first_fieldref()
      self.buffer.create_row(self.rowref,pos=0)
      newrowref,newfieldref = self.rowref,self.fieldref
      # process init events
      for field in self.fieldlist:
         # init if field is rowfield or dbfield
         if field.rowbuffer or field.dbtable is not None:
            self.buffer.recalc_init(newrowref,field)
      # before events
      self.call_record_before(newrowref)
      self.call_field_before(newfieldref)
   else:
      index = self.buffer.rowreforder.index(oldrowref)
      # after events
      self.call_field_after()
      self.call_record_after()
      self.change_deleted_rowref(index)
      # Insert new row if current rowref is not new record
      if self.rowref <> common.ROWREF_NEW_RECORD:
         self.rowref = common.ROWREF_NEW_RECORD
         self.fieldref = self.get_first_fieldref()
         self.buffer.create_row(self.rowref,pos=index+1)
         newrowref,newfieldref = self.rowref,self.fieldref
         # process init events
         for field in self.fieldlist:
            # init if field is rowfield or dbfield
            if field.rowbuffer or field.dbtable is not None:
               self.buffer.recalc_init(newrowref,field)
```

```
            # before events
            self.call_record_before(newrowref)
            self.call_field_before(newfieldref)
    def process_focus(self,rowref,fieldref):
        ### M Obradi_fokus
        print "BLOCK: PROCESS FOCUS OLD =",rowref,fieldref
        # focus is always (rowref,fieldref)
        oldfocus = (self.rowref,self.fieldref)
        newfocus = (rowref,fieldref)
        if self.block_before_executed is False:
            self.focus_enter(newfocus)
        else:
            self.focus_list_all = self.focus_list_all[-9:] + [oldfocus]
            if len(self.focus_list_diff)>0 and self.focus_list_diff[-1]<>oldfocus:
                self.focus_list_diff = self.focus_list_diff[-9:] + [oldfocus]
            if oldfocus <> newfocus:
                self.focus_move(oldfocus,newfocus)
    def show_focus(self,focus):
        if focus is None:
            return 'N/A'
        return str(focus)
    def get_prev_rowref(self):
        ### M Dohvati_prethodni_identifikator
        index = self.buffer.rowreforder.index(self.rowref)
        if index > 0:
            return self.buffer.rowreforder[index-1]
        return self.rowref
    def get_next_rowref(self):
        ### M Dohvati_slijedeci_identifikator
        index = self.buffer.rowreforder.index(self.rowref)
        if index+1 < len(self.buffer.rowreforder):
            return self.buffer.rowreforder[index+1]
        return self.rowref
    def focus_enter(self,newfocus):
        ### M Fokus_ulazak
        self.after_return_value = None
        newrowref,newfieldref = newfocus
        self.output("FOCUS ENTER TO '%s'"%self.show_focus(newfocus))
        self.output("BLOCK_BEFORE_EXECUTED %s"%str(self.block_before_executed))
        self.call_block_before()
        self.call_record_before(newrowref)
        self.call_field_before(newfieldref)
```

```
    def focus_move(self,oldfocus,newfocus):
        ### M Fokus_promjena
        self.after_return_value = None
        oldrowref,oldfieldref = oldfocus
        newrowref,newfieldref = newfocus
        self.output("FOCUS MOVE FROM '%s' TO '%s'"%(self.show_focus(oldfocus),self.show_focus(newfocus)))
        if oldrowref<>newrowref:
            self.call_field_after()
            if self.after_return_value is not False: self.call_record_after()
            if self.after_return_value is not False: self.call_record_before(newrowref)
            if self.after_return_value is not False: self.call_field_before(newfieldref)
        else:
            if oldfieldref<>newfieldref:
                self.call_field_after()
                if self.after_return_value is not False: self.call_field_before(newfieldref)
    def focus_leave(self,oldfocus):
        ### Fokus_izlazak
        self.after_return_value = None
        rowref,fieldref = oldfocus
        self.output("FOCUS LEAVE FROM '%s'"%self.show_focus(oldfocus))
        self.call_field_after()
        if self.after_return_value is not False: self.call_record_after()
        if self.after_return_value is not False: self.call_block_after()
    def call_block_before(self):
        ### M Pozovi_dogadjaj_prije_bloka
        self.output('BLOCK BEFORE EVENT')
        e = self.findevent(common.EVT_BLOCK_BEFORE)
        if e is not None:
            e.call()
        self.block_before_executed = True
    def call_block_after(self):
        ### M Pozovi_dogadjaj_poslije_bloka
        self.output('BLOCK AFTER EVENT')
        e = self.findevent(common.EVT_BLOCK_AFTER)
        if e is not None:
            self.after_return_value = e.call()
    def call_record_before(self,newrowref):
        ### M Pozovi_dogadjaj_prije_redka
        if newrowref in self.buffer.rowreforder:
            self.rowref = newrowref
            self.output('RECORD BEFORE EVENT %s'%str(self.rowref))
```

```
        e = self.findevent(common.EVT_RECORD_BEFORE)
        if e is not None:
            e.call()
    # backup field values
    self.buffer.backup_field_values()
def call_record_after(self):
    ### M Pozovi_dogadjaj_poslije_redka
    if self.rowref in self.buffer.rowreforder:
        self.output('RECORD AFTER EVENT %s'%str(self.rowref))
        e = self.findevent(common.EVT_RECORD_AFTER)
        if e is not None:
            self.after_return_value = e.call()
            # if record after returns false exit
            if self.after_return_value is False:
                return
        # if record still NEW - delete
        if self.rowref == common.ROWREF_NEW_RECORD:
            self.buffer.delete_row(self.rowref)
            print "DELETING EMPTY RECORD",self.rowref
def call_field_before(self,newfieldref):
    ### M Pozovi_dogadjaj_prije_varijable
    self.fieldref = newfieldref
    self.output('FIELD BEFORE EVENT %s'%str(self.fieldref))
    e = self.findevent(common.EVT_FIELD_BEFORE,self.fieldref)
    if e is not None:
        e.call()
def call_field_after(self):
    ### M Pozovi_dogadjaj_poslije_varijable
    self.output('FIELD AFTER EVENT %s'%str(self.fieldref))
    e = self.findevent(common.EVT_FIELD_AFTER,self.fieldref)
    if e is not None:
        self.after_return_value = e.call()
def handle_exception(self):
    # False - stop method execution
    # True - continue method execution
    # sys.exit(1) - exits system immediatelly
    _type,_value,_traceback = sys.exc_info()
    self.output("BLOCK ERROR %s"%str((_type,_value,_traceback)))
    # Prints error traceback
    traceback.print_exc()
    # Execute error event if exists
    e = self.findevent(common.EVT_ERROR)
```

```
            if e is not None:
                return_value = e.call(_type.__name__,_value)
                if return_value is None:
                    sys.exit(1)
                return return_value
            sys.exit(1)
        def close(self):
            ### M Zatvori_blok
            print "CLOSE BEGIN"
            self.focus_leave((self.rowref,self.fieldref))
            # if no error then stop running
            if self.after_return_value is not False:
                self.running = False
                print "STOP RUNNING"
            print "CLOSE END"
            print "After Return Value:",self.after_return_value
            print "Running:",self.running
        def should_insert(self):
            ### M Treba_umetnuti
            mc = self.buffer.get_curr_call()
            return mc.should_insert
        def should_delete(self):
            ### M Treba_obrisati
            mc = self.buffer.get_curr_call()
            return mc.should_delete
        def should_update(self,alias=None):
            ### M Treba_azurirati
            mc = self.buffer.get_curr_call()
            if alias is None:
                return len(mc.should_update)>0
            table = self.tablealiasdict.get(alias,None)
            if table is None:
                return False
            return table in mc.should_update

    class BlockDict:
        def __init__(self,block,rowref):
            self.block = block
            self.rowref = rowref
        def __getitem__(self,name):
            scope = self.block.buffer.get_scope()
            # simulate mapping sequence for local parameter in eval
```

```
        if name == self.block.name:
            return BlockWrapper(self.block,rowref=self.rowref)
        elif name == 'self':
            return self.block
        elif name == 'global':
            return self.block
        elif (scope,name) in self.block.fielddict.keys(): # field name in curr scope
            return self.block.fielddict[scope,name].get(rowref=self.rowref)
        elif (None,name) in self.block.fielddict.keys(): # field name in block scope
            return self.block.fielddict[None,name].get(rowref=self.rowref)
        elif name in self.block.methoddict.keys():
            return self.block.methoddict[name]
        elif name in self.block.parentdict.keys():
            return BlockWrapper(self.block.parentdict[name])
        elif __builtins__.has_key(name):
            return __builtins__[name]
        raise common.AlatUnknownItemError(name)


class BlockWrapper:
    # Block wrapper class for expression evaluation
    def __init__(self,block,**kw):
        self.__block__ = block
        self.__rowref__ = kw.get('rowref',block.rowref)
    def __str__(self):
        return str(self.__block__)
    def __getattr__(self,name):
        return self.__block__.fielddict[None,name].get(rowref=self.__rowref__)

class Application:
    ## KLASA: Aplikacija
    def __init__(self):
        self.globaldict = {}
        ## ATRIBUT: Globalne_varijable
    def set(self,globalname,value):
        ## METODA: Postavi
        self.globaldict[globalname]=value
    def get(self,globalname):
        ## METODA: Dohvati
        return self.globaldict.get(globalname,None)
    def load(self,blockname):
        ## METODA: Ucitaj
```

```
        # should be overriden
        return None
    def call(self,blockname,*l,**kw):
        ## METODA: Pozovi
        block = self.load(blockname)
        block.rebuild()
        return block.start(*l,**kw)
```

MODUL: _base_block_buffer.py

```
import common

class Buffer:
    ## KLASA: Spremnik_bloka
    def __init__(self,block):
        self.block = block
        self.initialize()
    def initialize(self):
        self.clear_db_data()
        self.vblockbuffer = {} # dict[field] = value  - block virtual fields
        ## ATRIBUT: Varijable_bloka
        self.vrowbuffer = {}    # dict[field][rowref] = value  - row virtual fields
        ## ATRIBUT: Varijable_sloga
        self.callbuffer = [] # list[call generation] *NEW*
        ## ATRIBUT: Lokalni_pozivi
    def clear_db_data(self):
        self.fetchedbuffer = {}
        ## ATRIBUT: Dohvaceni_bazni_podaci
        self.changedbuffer = {}
        ## ATRIBUT: Promijenjeni_bazni_podaci
        self.recalcedbuffer = {}
        ## ATRIBUT: Rekalkulirani_bazni_podaci
        self.rowreforder = []
        ## ATRIBUT: Redoslijed_slogova
    def backup_field_values(self):
        # After RECORD BEFORE
        for field in self.block.fieldlist:
            if field.scope is None:
                field.cancelvalue = self.lastvalue = field.get()
    def restore_field_values(self,rowref):
        # delete changed
        del self.changedbuffer[rowref]
```

```
        self.changedbuffer[rowref]={}
        # delete recalculated
        del self.recalcedbuffer[rowref]
        self.recalcedbuffer[rowref]={}
        # restore values
        for field in self.block.fieldlist:
            if field.scope is None:
                if field.globalname is None:
                    #field.set(field.lastvalue)
                    pass
                if field.dbtable is None: # virtual field
                    if field.rowbuffer and not field.is_parameter(): # row virtual field
                        self.vrowbuffer[field][rowref] = field.cancelvalue
                    else: # block virtual field
                        self.vblockbuffer[field] = field.cancelvalue
                else: # dbfield
                    if field.get() is not field.cancelvalue:
                        self.recalcedbuffer[rowref][field]=field.cancelvalue
    def rebuild(self):
        ## METODA: Izgradi
        self.initialize()
        for field in self.block.fieldlist:
            if field.globalname is not None: # Reference to global variable
                pass
            elif field.scope is not None: # method parameter/virtual
                pass
            elif field.dbtable is None: # virtual field
                if field.rowbuffer and not field.is_parameter(): # row virtual field
                    self.vrowbuffer[field] = {}
                else: # block virtual field
                    self.vblockbuffer[field] = None
            else: # dbfield
                pass
        # Save fieldable objects
        for object in self.block.fieldablelist:
            object.field.set(object)
    def delchanged(self,rowref,field):
        ## METODA: Izbrisi_promjenu
        if self.ischanged(rowref,field):
            del self.changedbuffer[rowref][field]
    def ischanged(self,rowref,field):
        ## METODA: Je_li_promijenjeno_polje
```

```
    return self.changedbuffer.has_key(rowref) and self.changedbuffer[rowref].has_key(field)
def delrecalced(self,rowref,field):
    ## METODA: Izbrisi_rekalkulaciju
    if self.isrecalced(rowref,field):
        del self.recalcedbuffer[rowref][field]
def isrecalced(self,rowref,field):
    ## METODA: Je_li_rekalkulirano_polje
    return self.recalcedbuffer.has_key(rowref) and self.recalcedbuffer[rowref].has_key(field)
def get(self,rowref,field,fetched=False):
    ## METODA: Dohvati
        if field.dbtable is None: # virtual field
            if field.globalname is not None: # Reference to global variable
                return self.block.application.get(field.globalname)
            elif field.scope is not None: # method parameter/virtual
                methodcall = self.get_curr_call()
                return methodcall.vardict[field]
                # OLD: return self.get_stack_vardict()[field]
            elif field.rowbuffer and not field.is_parameter(): # row virtual field
                if rowref is not common.ROWREF_NO_RECORD:
                    return self.vrowbuffer.get(field,{}).get(rowref,None)
            else: # block virtual field
                return self.vblockbuffer.get(field,None)
        elif rowref is not common.ROWREF_NO_RECORD: # dbfield
            if fetched:
                # return only fetched value
                return self.fetchedbuffer.get(rowref,{}).get(field,None)
            else:
                value = self.changedbuffer.get(rowref,{}).get(field,None)
                if value is None:
                    value = self.recalcedbuffer.get(rowref,{}).get(field,None)
                if value is None:
                    value = self.fetchedbuffer.get(rowref,{}).get(field,None)
                return value
        return None
def set(self,rowref,field,value,fetched=False):
    ## METODA: Postavi
        if field.dbtable is None: # virtual field
            if field.globalname is not None: # Reference to global variable
                self.block.application.set(field.globalname,value)
            elif field.scope is not None: # method parameter/virtual
                methodcall = self.block.buffer.get_curr_call()
                methodcall.vardict[field] = value
```

```
            # OLD: self.get_stack_vardict()[field]=value
        elif field.rowbuffer and not field.is_parameter(): # row virtual field
            if rowref is not common.ROWREF_NO_RECORD:
                self.vrowbuffer[field][rowref] = value
        else: # block virtual field
            self.vblockbuffer[field] = value
        # call recalculation loop triggered by this field
        self.recalc_loop(rowref,field)
    elif rowref is not common.ROWREF_NO_RECORD: # dbfield
        if fetched:
            self.fetchedbuffer[rowref][field]=value
        else:
            self.changedbuffer[rowref][field]=value
            self.delrecalced(rowref,field)
            # call recalculation loop triggered by this field
            self.recalc_loop(rowref,field)
def writeset(self,rowref,field):
    ## METODA: Zapisi_promjenu
    if field.dbtable is None: # virtual field
        return
    elif rowref is not common.ROWREF_NO_RECORD: # dbfield
        if self.isrecalced(rowref,field):
            self.fetchedbuffer[rowref][field]=self.recalcedbuffer[rowref][field]
            del self.recalcedbuffer[rowref][field]
        if self.ischanged(rowref,field):
            self.fetchedbuffer[rowref][field]=self.changedbuffer[rowref][field]
            del self.changedbuffer[rowref][field]
def create_row(self,rowref,pos=None):
    ## METODA: Kreiraj_redak
    if not rowref in self.rowreforder:
        self.fetchedbuffer[rowref] = {}
        self.recalcedbuffer[rowref] = {}
        self.changedbuffer[rowref] = {}
        if pos is None:
            self.rowreforder.append(rowref)
        else:
            self.rowreforder.insert(pos,rowref)
def delete_row(self,rowref):
    ## METODA: Izbrisi_redak
    print "BUFFER DELETE ROW",rowref
    if rowref in self.fetchedbuffer:
        del self.fetchedbuffer[rowref]
```

```
    if rowref in self.recalcedbuffer:
        del self.recalcedbuffer[rowref]
    if rowref in self.changedbuffer:
        del self.changedbuffer[rowref]
    if rowref in self.rowreforder:
        del self.rowreforder[self.rowreforder.index(rowref)]
def create_rowref(self,rowreflist,fieldvaluedict):
    ## METODA: Kreiraj_identifikator_redka
    rowref = []
    for rowreffield in rowreflist:
        rowref.append(fieldvaluedict.get(rowreffield,None))
    return tuple(rowref)
def change_rowref(self,oldrowref,newrowref):
    ## METODA: Promijeni_identifikator_redka
    # fetched buffer
    self.fetchedbuffer[newrowref]=self.fetchedbuffer[oldrowref]
    del self.fetchedbuffer[oldrowref]
    # recalced buffer
    self.recalcedbuffer[newrowref]=self.recalcedbuffer[oldrowref]
    del self.recalcedbuffer[oldrowref]
    # changed buffer
    self.changedbuffer[newrowref]=self.changedbuffer[oldrowref]
    del self.changedbuffer[oldrowref]
    # row virtuals
    for field in self.vrowbuffer.keys():
        if self.vrowbuffer[field].has_key(oldrowref):
            self.vrowbuffer[field][newrowref]=self.vrowbuffer[field][oldrowref]
            del self.vrowbuffer[field][oldrowref]
    # roworder
    self.rowreforder[self.rowreforder.index(oldrowref)]=newrowref
# RECALC:
def recalc_init(self,rowref,targetfield):
    ## METODA: Rekalkuliraj_init
    if targetfield.recalc is not None:
        targetfield.recalc.process_init(rowref)
def recalc_loop(self,rowref,sourcefield):
    ## METODA: Proces_rekalkulacije
    print "BEGIN RECALC LOOP"
    inlist = [sourcefield]
    outlist = []
    changedlist = []
    while len(inlist)>0:
```

```
        # Get source field
        sf = inlist.pop(0)
        print "  Processing change of",sf
        # process target field
        for trecalc in sf.recalclist:
            tf = trecalc.targetfield
            if tf not in inlist and tf not in outlist:
                inlist.append(tf)
                changedlist.append(tf)
                print "    Changing",tf
                trecalc.process(rowref)
        outlist.append(sf)
    print "END RECALC LOOP"
    return changedlist
def set_recalc(self,rowref,field,value):
    ## METODA: Postavi_rekalkulaciju
    if field.dbtable is None: # virtual field
        if field.globalname is not None: # Reference to global variable
            self.block.application.set(field.globalname,value)
        elif field.scope is not None: # method parameter/virtual
            methodcall = self.block.buffer.get_curr_call()
            methodcall.vardict[field] = value
            # OLD: self.get_stack_vardict()[field]=value
        elif field.rowbuffer and not field.is_parameter(): # row virtual field
            if rowref is not common.ROWREF_NO_RECORD:
                self.vrowbuffer[field][rowref] = value
        else: # block virtual field
            self.vblockbuffer[field] = value
    elif rowref is not common.ROWREF_NO_RECORD: # dbfield
        self.recalcedbuffer[rowref][field]=value
        self.delchanged(rowref,field)
# CALL BUFFER:
def new_call(self,method,arglist,argdict):
    ## METODA: Novi_poziv
    methodcall = MethodCall(method,arglist,argdict)
    self.callbuffer.append(methodcall)
    return methodcall
def exit_curr_call(self):
    ## METODA: Zavrsi_trenutni_poziv
    self.callbuffer.pop()
def get_curr_call(self):
    ## METODA: Dohvati_trenutni_poziv
```

```
        return self.callbuffer[-1]
    def get_scope(self):
        ## METODA: Dohvati_doseg
        if len(self.callbuffer)>0:
            return self.get_curr_call().method
        return None
    def is_callbuffer_empty(self):
        ## METODA: Je_li_spremnik_poziva_prazan
        return len(self.callbuffer)==0

class MethodCall:
    ## KLASA: Spremnik_poziva_metode
    def __init__(self,method,arglist,argdict):
        ## METODA: Konstruktor
        self.method = method    # method
        ## A Metoda
        self.arglist = arglist  # call arg list
        ## A Lista_argumenata
        self.argdict = argdict  # call arg dict
        ## A Niz_argumenata
        self.vardict = {}       # local variable dict
        ## A Lokalne_varijable
        for param in method.paramlist: # method params to local var dict
            self.vardict[param]=None
        for virtual in method.virtuallist: # method virtuals to local var dict
            self.vardict[virtual]=None
        self.curr_instr = None  # current instruction
        ## A Trenutna_instrukcija
        self.level_instr = {}     # instruction of instr level
        ## A Instrukcija_nivoa
        self.level_tmpdata = {}      # temp data for instr level (used by instruction)
        ## A Podaci_nivoa
        self.return_value = None  # return value
        ## A Povratna_vrijednost
        self.should_eval()
    def should_eval(self):
        ## METODA: Evaluacija_potrebe_zapisivanja_sloga
        self.should_insert = False
        ## A Treba_insertirati
        self.should_delete = False
        ## A Treba_izbrisati
        self.should_update = []   # list of tables for update
```

```
        ## A Treba_azurirati
        if self.method.should_eval:
            block = self.method.block
            if block.delete_row_request is not None:
                self.should_delete = True
            elif block.rowref is common.ROWREF_NEW_RECORD:
                if block.maintable is not None and self.table_changed(block,block.maintable):
                    self.should_insert = True
            else:
                for table in block.tablelist:
                    if self.table_changed(block,table):
                        self.should_update.append(table)
    def table_changed(self,block,table):
        ## METODA: Je_li_promjenjena_tablica
        for field in block.tablefieldlist[table]:
            if block.buffer.ischanged(block.rowref,field):
                return True
        return False
    def level_set(self,level,instr,tmpdata=None):
        ## METODA: Postavi_nivo_instrukcije
        self.level_instr[level] = instr
        self.level_tmpdata[level] = tmpdata
    def level_get(self,level):
        ## METODA: Dohvati_nivo_instrukcije
        return self.level_instr[level],self.level_tmpdata[level]
```

MODUL: _base_code.py

```
import common

import sys, traceback

class Event:
    ## E Dogadjaj
    def __init__(self,block,should_eval=False):
        self.type = None
        ## A Tip
        self.block = block
        ## A Blok
        self.key = None
        ## A Kljuc
        self.method = None
```

```
        ## A Metoda
        self.method = Method(block,None,event=self,should_eval=should_eval)
    def register(self,*params):
        ## M Registriraj
        self.key = tuple([self.type]+list(params))
        self.block.eventlist.append(self)
        self.block.eventdict[self.key]=self
    def call(self,*l,**kw):
        ## Pozovi
        if self.method is not None:
            return self.method.call(*l,**kw)
    def __str__(self):
        return '<Event %s>'%str(self.key)
    def __repr__(self):
        return self.__str__()

class EventBlockDescriptor(Event):
    ## E Dogadjaj opisa bloka
    ## EVT BLOK_OPIS
    def __init__(self,block,**kw):
        Event.__init__(self,block)
        self.type = common.EVT_BLOCK_DESCRIPTOR
        self.register()

class EventBlockFetch(Event):
    ## E Dogadjaj dohvata na bloku
    ## EVT BLOK_DOHVACANJE
    def __init__(self,block,**kw):
        Event.__init__(self,block)
        self.type = common.EVT_BLOCK_FETCH
        self.register()

class EventRecordFetch(Event):
    ## E Dogadjaj dohvata na redku
    ## EVT SLOG_DOHVACANJE
    def __init__(self,block,**kw):
        Event.__init__(self,block)
        self.type = common.EVT_RECORD_FETCH
        self.register()

class EventBlockBefore(Event):
    ## E Dogadjaj prije bloka
```

```
    ## EVT BLOK_PRIJE
    def __init__(self,block,**kw):
        Event.__init__(self,block)
        self.type = common.EVT_BLOCK_BEFORE
        self.register()

class EventBlockAfter(Event):
    ## E Dogadjaj poslije bloka
    ## EVT BLOK_POSLIJE
    def __init__(self,block,**kw):
        Event.__init__(self,block)
        self.type = common.EVT_BLOCK_AFTER
        self.register()

class EventRecordBefore(Event):
    ## E Dogadjaj prije redka
    ## EVT SLOG_PRIJE
    def __init__(self,block,**kw):
        Event.__init__(self,block)
        self.type = common.EVT_RECORD_BEFORE
        self.register()

class EventRecordAfter(Event):
    ## E Dogadjaj poslije redka
    ## EVT SLOG_POSLIJE
    def __init__(self,block,**kw):
        Event.__init__(self,block,should_eval=True)
        self.type = common.EVT_RECORD_AFTER
        self.register()

class EventFieldBefore(Event):
    ## E Dogadjaj prije varijable
    ## EVT VARIJABLA_PRIJE
    def __init__(self,block,field,**kw):
        Event.__init__(self,block)
        self.type = common.EVT_FIELD_BEFORE
        self.field = field
        self.register(field)

class EventFieldAfter(Event):
    ## E Dogadjaj poslije varijable
    ## EVT VARIJABLA_POSLIJE
```

```
    def __init__(self,block,field,**kw):
        Event.__init__(self,block)
        self.type = common.EVT_FIELD_AFTER
        self.field = field
        self.register(field)

class EventFieldAction(Event):
    ## E Dogadjaj akcije na varijabli
    ## EVT VARIJABLA_AKCIJA
    def __init__(self,block,field,**kw):
        Event.__init__(self,block)
        self.type = common.EVT_FIELD_ACTION
        self.field = field
        self.register(field)

class EventExternal(Event):
    ## E Vanjski dogadjaj
    ## EVT VANJSKI
    def __init__(self,block,modulefield,eventname,*params,**kw):
        Event.__init__(self,block)
        self.type = common.EVT_EXTERNAL
        self.modulefield = modulefield
        self.eventname = eventname
        self.register(modulefield,eventname,*params)


class EventFieldChange(Event):
    ## E Dogadjaj promjene vrijednosti varijable
    ## EVT VARIJABLA_PROMJENA
    def __init__(self,block,field,**kw):
        Event.__init__(self,block)
        # method has two parameters: lastvalue and place
        self.type = common.EVT_CHANGE
        self.field = field
        self.register(field)

class EventError(Event):
    ## E Dogadjaj greske
    ## EVT GRESKA
    def __init__(self,block,**kw):
        Event.__init__(self,block)
        # method has two parameters: errorname and value
```

```
        self.type = common.EVT_ERROR
        self.register()

class EventBlockExit(Event):
    ## E Dogadjaj izlaska iz bloka
    ## EVT BLOK_IZLAZ
    def __init__(self,block,**kw):
        Event.__init__(self,block)
        self.type = common.EVT_BLOCK_EXIT
        self.register()

class Method:
    ## E Metoda
    def __init__(self,block,name,**kw):
        self.name = name
        ## A Ime
        self.block = block
        ## A Blok
        block.methodlist.append(self)
        if name is not None:
            block.methoddict[name]=self
        self.instrlist = []
        ## A Lista_instrukcija
        self.paramlist = []
        ## A Lista_parametara
        self.virtuallist = []
        ## A Lista_virtualnih_varijabli
        self.event = kw.get('event',None)
        ## A Dogadjaj
        self.should_eval = kw.get('should_eval',False)
        ## A Treba_li_evaluirati
        if self.event is not None and self.event.method is None:
            self.event.method = self
    def add(self,instr):
        self.instrlist.append(instr)
        instr.index = len(self.instrlist)-1
    def args2vars(self):
        ## M Argumenti_u_varijable
        i = 0
        methodcall = self.block.buffer.get_curr_call()
        arglist = methodcall.arglist
        argdict = methodcall.argdict
```

```
        print "Args:",arglist,argdict
        for p in self.paramlist:
            if p.param_in:
                if len(arglist)>i:
                    p.set(self.block.getvalue(arglist[i]))
                if argdict.has_key(p.name):
                    p.set(self.block.getvalue(argdict[p.name]))
            i = i + 1
    def vars2args(self):
        ## M Varijable_u_argumente
        # local vars and params to args - output
        i = 0
        methodcall = self.block.buffer.get_curr_call()
        arglist = methodcall.arglist
        argdict = methodcall.argdict
        for p in self.paramlist:
            if p.param_out:
                if len(arglist)>i:
                    arglist[i].set(p.get())
                if argdict.has_key(p.name):
                    argdict[p.name].set(p.get())
            i = i + 1
    def start(self,*arglist,**argdict):
        ## M Start
        self.block.output('Start method %s'%str(self))
        methodcall = self.block.buffer.new_call(self,arglist,argdict)
        instr = None
        if len(self.instrlist)>0:
            instr = self.instrlist[0]
        methodcall.curr_instr = instr
        self.args2vars()
    def step(self):
        ## M Korak
        if self.block.buffer.is_callbuffer_empty(): # OLD: is_stack_empty
            self.block.output('Exit: Buffer Stack empty.')
            return False
        methodcall = self.block.buffer.get_curr_call()
        instr = methodcall.curr_instr
        if instr is not None:
            try:
                instr.execute()
                return True
```

```
            except:
                return_value = self.block.handle_exception()
                if return_value not in (False,None):
                    instr.on_error()
                return return_value
        else:
            self.vars2args()
            self.block.output('End method %s'%str(self))
            return False
    def call(self,*l,**kw):
        ## M Poziv
        self.start(*l,**kw)
        return self.loop()
    def __str__(self):
        if self.event is not None:
            return '<Method-Event %s>'%str(self.event.key)
        return '<Method %s>'%self.name
    def __repr__(self):
        return self.__str__()
    def __call__(self,*l,**kw):
        return self.call(*l,**kw)
    def loop(self):
        ## M Iteracija
        while self.step() and self.block.running:
            self.block.stepcounter += 1
        # exit current methodcall and return value
        methodcall = self.block.buffer.get_curr_call()
        return_value = methodcall.return_value
        print "SHOULD REPORT:"
        print " Should_insert:",methodcall.should_insert
        print " Should_delete:",methodcall.should_delete
        print " Should_update:",methodcall.should_update
        self.block.buffer.exit_curr_call()
        return return_value

class _Instruction:
    def __init__(self,parent,level,**kw):
        self.parent = parent # method
        ## A Roditelj
        self.level = level
        ## A Nivo
        self.index = None
```

```
    ## A Indeks
    self.block = self.parent.block
    ## A Blok
    self.parent.add(self)
    self.condition = kw.get('condition',True) # True, False or expression
    ## A Uvjet
def on_error(self):
    ## M Obrada_greske
    self.next()
    self.block.show_prompt('ERROR IN %s'%str(self))
def find_next(self):
    ## M Pronadji_slijedecu_instrukciju
    # finds next instr of any level. None if last
    if self.index+1<len(self.parent.instrlist):
        return self.parent.instrlist[self.index+1]
    return None
def find_skip(self):
    ## M Pronadji_visu_ili_jednaku_slijedecu_instrukciju
    # finds next instr of equal or greater level. None if last
    instr = self.find_next()
    while instr is not None and instr.level>self.level:
        instr = instr.find_next()
    return instr
def next(self):
    ## M Skoci_na_slijedecu_instrukciju
    self.goto(self.find_next())
def execute(self):
    ## M Izvrsi
    self.set_level()
    self.next()
    self.block.show_prompt(self)
def set_level(self,data=None):
    ## M Postavi_nivo
    methodcall = self.block.buffer.get_curr_call()
    methodcall.level_set(self.level,self,data)
def get_level(self,level=None):
    ## M Dohvati_nivo
    if level is None:
        level = self.level
    methodcall = self.block.buffer.get_curr_call()
    instr, data = methodcall.level_get(level)
    return instr,data
```

```
    def loop_resolution(self,instr):
        ## M Rijesi_iteraciju
        # loop resolution - if going to higher level
        if self.level>0: # only if in sublevel
            highest = 0
            if instr is not None:
                highest = instr.level
            if highest<self.level: # candidate for loop searching
                for x in xrange(self.level-highest):
                    linstr,ldata = self.get_level(self.level-x-1)
                    if linstr.__class__ is WhileInstr:
                        return linstr
        return instr
    def goto(self,instr):
        ## Skoci
        instr = self.loop_resolution(instr)
        methodcall = self.block.buffer.get_curr_call()
        methodcall.curr_instr = instr

class UpdateInstr(_Instruction):
    ## V Instrukcija_pridruzivanja
    def __init__(self,parent,level,field,argument,**kw):
        _Instruction.__init__(self,parent,level,**kw)
        self.field = field # field to update
        ## A Varijabla
        self.argument = argument # expression or field
        ## A Vrijednost
    def execute(self):
        self.set_level()
        if common.boolean(self.condition) is True:
            if self.field is not None:
                self.field.set(self.argument.get())
            else:
                self.argument.get()
        self.next()
        self.block.show_prompt(self)
    def __str__(self):
        return '<UpdateInstr %s with %s>'%(self.field,self.argument)
    def __repr__(self):
        return self.__str__()

class EvaluateInstr(_Instruction):
```

```
    ## V Instrukcija_evaluacije
    def __init__(self,parent,level,argument,**kw):
        _Instruction.__init__(self,parent,level,**kw)
        self.argument = argument # expression or field
        ## A Vrijednost
    def execute(self):
        self.set_level()
        if common.boolean(self.condition) is True:
            self.argument.get()
        self.next()
        self.block.show_prompt(self)
    def __str__(self):
        return '<EvaluateInstr %s>'%self.argument
    def __repr__(self):
        return self.__str__()


class DumpInstr(_Instruction):
    ## V Instrukcija_komentiranja
    def __init__(self,parent,level,comment,*arguments,**kw):
        _Instruction.__init__(self,parent,level,**kw)
        self.comment = comment
        ## A Komentar
        self.arguments = arguments
        ## A Argumenti
    def execute(self):
        self.set_level()
        if common.boolean(self.condition) is True:
            self.block.output("  DUMP %s:"%self.comment)
            for a in self.arguments:
                self.block.output("  %s = %s:"%(str(a),str(a.get())))
        self.next()
        self.block.show_prompt(self)
    def __str__(self):
        return '<DumpInstr %s>'%self.comment
    def __repr__(self):
        return self.__str__()


class CallExtInstr(_Instruction):
    ## V Instrukcija_vanjskog_poziva
    def __init__(self,parent,level,modulefield,methodname,**kw):
```

```
        _Instruction.__init__(self,parent,level,**kw)
        self.modulefield = modulefield
        ## A Varijabla_vanjskog_modula
        self.methodname = methodname
        ## A Ime_metode_vanjskog_modula
        self.arglist = kw.get('arglist',[])
        ## A Lista_argumenata
        self.argdict = kw.get('argdict',{})
        ## A Niz_argumenata
    def execute(self):
        self.set_level()
        if common.boolean(self.condition) is True:
            getattr(self.modulefield.get(),self.methodname)(self.block,self.modulefield,\
                                    *self.arglist,**self.argdict)
        self.next()
        self.block.show_prompt(self)
    def __str__(self):
        return '<CallExtInstr %s.%s>'%(self.modulefield.get(),self.methodname)
    def __repr__(self):
        return self.__str__()

class IfInstr(_Instruction):
    ## V Ako_instrukcija
    def __init__(self,parent,level,**kw):
        _Instruction.__init__(self,parent,level,**kw)
        # uses regular instruction condition
    def __str__(self):
        return '<IfInstr %s>'%self.condition
    def __repr__(self):
        return self.__str__()
    def execute(self):
        cnd = common.boolean(self.condition)
        self.set_level(cnd)
        if cnd is True:
            instr = self.find_next()
            self.goto(instr)
        else:
            instr = self.find_skip()
            self.goto(instr)

class ElseInstr(_Instruction):
    ## V Inace_instrukcija
```

```
    def __init__(self,parent,level,**kw):
        _Instruction.__init__(self,parent,level,**kw)
        # uses regular instruction condition
    def __str__(self):
        return '<ElseInstr %s>'%self.condition
    def __repr__(self):
        return self.__str__()
    def execute(self):
        linstr,ldata = self.get_level() # previous instr
        if linstr is not None and linstr.__class__ in (IfInstr,ElseInstr) \
                            and ldata is False:
            cnd = common.boolean(self.condition)
            self.set_level(cnd)
        else:
            cnd = False
            self.set_level(ldata)
        if cnd is True:
            instr = self.find_next()
            self.goto(instr)
        else:
            instr = self.find_skip()
            self.goto(instr)

class WhileInstr(_Instruction):
    ## V Instrukcija_iteracije
    def __init__(self,parent,level,**kw):
        _Instruction.__init__(self,parent,level,**kw)
        # uses regular instruction condition
    def __str__(self):
        return '<WhileInstr %s>'%self.condition
    def __repr__(self):
        return self.__str__()
    def execute(self):
        cnd = common.boolean(self.condition)
        self.set_level(cnd)
        if cnd is True:
            instr = self.find_next()
            self.goto(instr)
        else:
            instr = self.find_skip()
            self.goto(instr)
```

```
class ExecSQLStatementInstr(_Instruction):
   ## SQL_instrukcija
   def __init__(self,parent,level,sqlstatement,**kw):
      _Instruction.__init__(self,parent,level,**kw)
      self.sqlstatement = sqlstatement # sqlstatement
      ## A SQL_element
      self.arguments = kw
      ## A Argumenti
   def execute(self):
      self.set_level()
      if common.boolean(self.condition) is True:
         print "Statement processed",self.sqlstatement
         kw = self.arguments
         self.sqlstatement.buffer_process(**kw)
      self.next()
      self.block.show_prompt(self)
   def __str__(self):
      return '<ExecSQLStatementInstr %>'%self.sqlstatement
   def __repr__(self):
      return self.__str__()

class CallInstr(_Instruction):
   ## V Instrukcija_poziva_bloka
   # Root block call
   def __init__(self,parent,level,blockname,**kw):
      _Instruction.__init__(self,parent,level,**kw)
      self.blockname = blockname
      ## A Ime_bloka
      self.arguments = kw
      ## A Argumenti
   def execute(self):
      self.set_level()
      if common.boolean(self.condition) is True:
         kw = self.arguments
         self.block.call(self.blockname,**kw)
      self.next()
      self.block.show_prompt(self)
   def __str__(self):
      return '<CallInstr %>'%self.blockname
   def __repr__(self):
      return self.__str__()
```

```
class SubCallInstr(_Instruction):
    ## V Instrukcija_poziva_podbloka
    # calling sub-blocks
    pass

class CommitInstr(_Instruction):
    ## V Commit_instrukcija
    def __init__(self,parent,level,conn,**kw):
        _Instruction.__init__(self,parent,level,**kw)
        self.conn = conn # connection
        ## A Konekcija
        self.arguments = kw
        ## A Argumenti
    def execute(self):
        self.set_level()
        if common.boolean(self.condition) is True:
            print "Commit on %s",self.conn
            kw = self.arguments
            self.conn.commit()
            print "Commit done."
        self.next()
        self.block.show_prompt(self)
    def __str__(self):
        return '<CommitInstr on %>'%self.conn
    def __repr__(self):
        return self.__str__()

class RollbackInstr(_Instruction):
    ## V Rollback_instrukcija
    def __init__(self,parent,level,conn,**kw):
        _Instruction.__init__(self,parent,level,**kw)
        self.conn = conn # connection
        ## A Konekcija
        self.arguments = kw
        ## A Argumenti
    def execute(self):
        self.set_level()
        if common.boolean(self.condition) is True:
            print "Rollback on %s",self.conn
            kw = self.arguments
            self.conn.rollback()
```

```
            print "Rollback done."
        self.next()
        self.block.show_prompt(self)
    def __str__(self):
        return '<RollbackInstr on %>'%self.conn
    def __repr__(self):
        return self.__str__()

class ReturnInstr(_Instruction):
    ## V Povratna_instrukcija
    def __init__(self,parent,level,value,**kw):
        _Instruction.__init__(self,parent,level,**kw)
        self.value = value # value to return: constant, field or expression
        ## A Vrijednost
    def execute(self):
        self.set_level()
        if common.boolean(self.condition) is True:
            # set methodcall return value and exit method execution
            methodcall = self.block.buffer.get_curr_call()
            methodcall.curr_instr = None
            methodcall.return_value = self.block.getvalue(self.value)
        else:
            self.next()
        self.block.show_prompt(self)
    def __str__(self):
        return '<ReturnInstr %s>'%(self.value)
    def __repr__(self):
        return self.__str__()
```

MODUL: _base_condition.py

```
import common

class Operator:
    def __init__(self,name,minargs,maxargs):
        self.name = name
        self.minargs = minargs
        self.maxargs = maxargs

op_eq = Operator(common.OP_EQ,1,1)
op_lt = Operator(common.OP_LT,1,1)
op_le = Operator(common.OP_LE,1,1)
```

```
op_gt = Operator(common.OP_GT,1,1)
op_ge = Operator(common.OP_GE,1,1)
op_ne = Operator(common.OP_NE,1,1)
op_isnull = Operator(common.OP_ISNULL,0,0)
op_in = Operator(common.OP_IN,1,None)
op_between = Operator(common.OP_BETWEEN,2,2)
op_like = Operator(common.OP_LIKE,1,1)

OPERATORS = [op_eq,op_lt,op_le,op_gt,op_ge,op_ne,op_isnull,op_in,op_between,op_like]

class Condition:
   ## E Uvjet
   def __init__(self,block,usercond=False):
      self.block = block
      ## A Blok
      block.conditionlist.append(self)
      self.usercond = usercond
      ## A Korisnicki_uvjet
      self.type = None
      ## A Tip
      self.table=None
      ## A Tablica
      self.having = False
      ## A Having_uvjet
      self.join = None
      ## A Veza

class OpCond(Condition):
   ## PE Uvjet_s_operacijom
   def __init__(self,block,field,operator,*arglist,**kw):
      Condition.__init__(self,block,kw.get('usercond',False))
      self.field = field
      ## A Polje
      self.table = field.dbtable
      self.operator = operator
      ## A Operacija
      self.arglist = arglist
      ## A Lista_argumenata
      self.notop = kw.get('notop',False)
      ## A Ne_operacija
      self.type = common.COND_OP
      self.having = kw.get('having',False)
```

```
        self.join = kw.get('join',None)

class ClauseCond(Condition):
    ## PE Uvjet_klauzule
    def __init__(self,block,table,clause,*arglist,**kw):
        Condition.__init__(self,block,kw.get('usercond',False))
        self.table = table
        self.clause = clause
        ## A Klauzula
        self.arglist = arglist
        ## A Lista_argumenata
        self.type = common.COND_CLAUSE
        self.having = kw.get('having',False)
        self.join = kw.get('join',None)

class ExprCond(Condition):
    ## PE Uvjet_s_izrazom
    def __init__(self,block,expr,usercond=False):
        Condition.__init__(self,block,usercond)
        self.expr = expr
        ## A Izraz
        self.type = common.COND_EXPR
```

MODUL: _base_datatype.py

```
from decimal import Decimal

class Type:
    def __init__(self):
        self.length = None
        self.precision = 0
        self.picture = None
    def db2buf(self,value,field): pass
        ## M Baza_Spremnik
    def buf2db(self,value,field): pass
        ## M Spremnik_baza
    def buf2str(self,value,field): pass
        ## M Spremnik_znakovni
    def str2buf(self,value,field): pass
        ## M Znakovni_spremnik

class GenericType(Type):
```

```
    def __init__(self):
        Type.__init__(self)
    def db2buf(self,value,field):
        return str(value)
    def buf2db(self,value,field):
        return str(value)
    def buf2str(self,value,field):
        return str(value)
    def str2buf(self,value,field):
        return str(value)


class StringType(Type):
    def __init__(self):
        Type.__init__(self)
    def db2buf(self,value,field):
        if value is None:
            return None
        return str(value)
    def buf2db(self,value,field):
        if value is None:
            return None
        return str(value)
    def buf2str(self,value,field):
        if value is None:
            return ''
        return str(value)
    def str2buf(self,value,field):
        if value is None or len(value.strip())==0:
            return None
        return str(value)

class NumberType(Type):
    def __init__(self):
        Type.__init__(self)
    def db2buf(self,value,field):
        if value is None:
            return None
        if field.precision <> 0:
            return Decimal(str(round(value,field.precision)))
        return int(float(value))
    def buf2db(self,value,field):
```

```
        if value is None:
            return None
        return float(value)
    def buf2str(self,value,field):
        if value is None:
            return ''
        return str(value)
    def str2buf(self,value,field):
        if value is None or len(value.strip())==0:
            return None
        if field.precision <> 0:
            return Decimal(str(round(value,field.precision)))
        return int(float(value))

class BooleanType(Type):
    def __init__(self):
        Type.__init__(self)
    def db2buf(self,value,field):
        if value is None:
            return None
        if value==0:
            return False
        return True
    def buf2db(self,value,field):
        if value is True:
            return 1
        if value is False:
            return 0
        return None
    def buf2str(self,value,field):
        if value is True:
            return 'Yes'
        if value is False:
            return 'No'
        return ''
    def str2buf(self,value,field):
        if value == 'Yes':
            return True
        if value == 'No':
            return False
        return None
```

```
class DateTimeType(Type):
   def __init__(self):
      Type.__init__(self)
   def db2buf(self,value,field):
      return str(value)
   def buf2db(self,value,field):
      return str(value)
   def buf2str(self,value,field):
      return str(value)
   def str2buf(self,value,field):
      return str(value)

generic = GenericType()
string = StringType()
number = NumberType()
boolean = BooleanType()
datetime = DateTimeType()
```

MODUL: _base_expression.py

```
import common

class Expression:
   ## E Izraz
   def
__init__(self,block,value,type=None,length=None,precision=None,picture=None,**kw):
      self.block = block
      ## A Blok
      block.expressionlist.append(self)
      self.value = value
      ## A Vrijednost
      self.type = type
      ## A Tip_podatka
      if self.type is not None:
         self.length = common.nvl(length,self.type.length)
         self.precision = common.nvl(precision,self.type.precision)
         self.picture = common.nvl(picture,self.type.picture)
      else:
         self.length = length
         self.precision = common.nvl(precision,0)
         self.picture = picture
   def convert(self,value,method=None):
```

```
        ## M Pretvori
        if self.type is not None:
            if method==common.CONV_DB2BUF:
                return self.type.db2buf(value,self)
            elif method==common.CONV_BUF2DB:
                return self.type.buf2db(value,self)
            elif method==common.CONV_BUF2STR:
                return self.type.buf2str(value,self)
            elif method==common.CONV_STR2BUF:
                return self.type.str2buf(value,self)
        return value
    def get(self,**kw):
        ## Dohvati
        method = kw.get('method',None)
        return self.convert(self.value,method)
    def set(self,value,fetched=False,**kw):
        ## Postavi
        # dummy method
        pass
    def __str__(self):
        return '<Expr>'
    def __repr__(self):
        return self.__str__()

class ConstExpr(Expression):
    def __init__(self,block,const,**kw):
        Expression.__init__(self,block,const,**kw)
    def get(self,**kw):
        method = kw.get('method',None)
        return self.convert(Expression.get(self),method)
    def __str__(self):
        return '<ConstExpr %s>'%str(self.value)

class EvalExpr(Expression):
    def __init__(self,block,strexpr,**kw):
        Expression.__init__(self,block,strexpr,**kw)
    def get(self,**kw):
        rowref = kw.get('rowref',self.block.rowref)
        method = kw.get('method',None)
        return self.convert(eval(Expression.get(self),{},self.block.blockdict(rowref)),method)
    def __str__(self):
        return '<EvalExpr %s>'%str(self.value)
```

```
class LambdaExpr(Expression):
   def __init__(self,block,lambdaexpr,**kw):
      Expression.__init__(self,block,lambdaexpr,**kw)
   def get(self,**kw):
      f = Expression.get(self)
      method = kw.get('method',None)
      return self.convert(f(),method)
   def __str__(self):
      return '<LambdaExpr %s>'%str(self.value)
```

MODUL: _base_field.py

```
import common

class Field:
   ## Element varijabla
   def __init__(self,block,name,type,dbtable=None,dbname=None,length=None,\
                                 precision=None,picture=None,**kw):
      self.block = block
      ## A Blok
      self.name = name
      ## A Ime
      self.lastvalue = None
      self.cancelvalue = None
      self.globalname = kw.get('globalname',None) # global variable name
      ## A Ime_globalne_varijable
      self.scope = kw.get('scope',None) # None is global scope, otherwise method/event
      ## A Doseg
      self.first = kw.get('first',False) # If True First field to jump in the row
      ## A Prva_varijabla
      if self.first and self.block.firstfield is None:
         self.block.firstfield = self
         self.block.fieldref=self
      block.fieldlist.append(self)
      block.fielddict[self.scope,name]=self
      block.tablefieldlist[dbtable].append(self)
      self.type = type
      ## A Tip
      self.recalc = None # recalc that changes this field
      ## A Rekalkulacija
      self.recalclist = [] # list of recalcs affected by change of this field
```

```
## A Lista_rekalkulacija
self.length = common.nvl(length,self.type.length)
## A Duljina
self.precision = common.nvl(precision,self.type.precision)
## A Preciznost
self.picture = common.nvl(picture,self.type.picture)
## A Format
self.modifiable = kw.get('modifiable',True) # Is modifiable by user; Can be expr;
                                # Should be used in UI, no effect on update
## A Promjenjivost
# SQL attributes:
self.dbtable = dbtable # if None -> virtual field
## A Bazna_tablica
self.dbname = dbname # fieldname in dbtable
## A Bazno_polje
self.rowref = kw.get('rowref',False)
## A Dio_identifikatora_redka
self.rowid = kw.get('rowid',False)
## A Bazni_identifikator_redka
self.select = kw.get('select',True)
## A Dio_SQL_select_naredbe
self.update = kw.get('update',False)
## A Dio_SQL_update_naredbe
self.insert = kw.get('insert',False)
## A Dio_SQL_insert_naredbe
self.rowbuffer = kw.get('rowbuffer',False) # rowlevel or blocklevel (not dbfields)
## A Dio_redka
self.param_in = kw.get('param_in',False) # Is input parameter?
## A Ulazni_parametar
self.param_out = kw.get('param_out',False) # Is output parameter?
                #(if param in/out is yes should not be rowbuffer)
## A Izlazni_parametar
self.groupby = kw.get('groupby',False)
## A Dio_group_by_klauzule
self.aggfnc = kw.get('aggfnc',None)
## A Agregatna_funkcija
self.orderby = kw.get('orderby',None)
self.descending = kw.get('descending',False)
## A Order_by_segment
self.sqlselect = kw.get('sqlselect',None) # SQLExpr
self.sqlinsert = kw.get('sqlinsert',None) # SQLExpr
self.sqlupdate = kw.get('sqlupdate',None) # SQLExpr
```

```
    # connect to scope object
    self.data = kw.get('data',{}) # additional data for external libraries
    if self.scope is not None:
        if self.is_parameter():
            self.scope.paramlist.append(self)
        else:
            self.scope.virtuallist.append(self)
    # set rowid reference to table
    if self.dbtable is not None and self.rowid:
        self.dbtable.rowid = self
def is_parameter(self):
    return self.param_in or self.param_out
def __str__(self):
    return '<Field %s>'%self.name
def __repr__(self):
    return self.__str__()
def convert(self,value,method=None):
    ## M Pretvori
    if method==common.CONV_DB2BUF:
        return self.type.db2buf(value,self)
    elif method==common.CONV_BUF2DB:
        return self.type.buf2db(value,self)
    elif method==common.CONV_BUF2STR:
        return self.type.buf2str(value,self)
    elif method==common.CONV_STR2BUF:
        return self.type.str2buf(value,self)
    return value
def get(self,fetched=False,**kw):
    ## Dohvati
    rowref = kw.get('rowref',self.block.rowref)
    value = self.block.buffer.get(rowref,self,fetched)
    method = kw.get('method',None)
    return self.convert(value,method=method)
def set(self,value,fetched=False,**kw):
    ## Postavi
    rowref = kw.get('rowref',self.block.rowref)
    method = kw.get('method',None)
    place = kw.get('place',None)
    self.lastvalue = self.get(fetched,rowref=rowref)
    self.block.buffer.set(rowref,self,self.convert(value,method=method),fetched)
    # call CHANGE_EVENT
    change_event = self.block.findevent(common.EVT_CHANGE,self)
```

```
        if change_event is not None:
            change_event.call(self.lastvalue,place)

class DbField(Field):
    ## PE Bazno_polje
    def __init__(self,block,name,type,dbtable,dbname,*l,**kw):
        kw['rowbuffer']=False
        kw['param_in']=False
        kw['param_out']=False
        kw['globalname']=None
        Field.__init__(self,block,name,type,dbtable,dbname,*l,**kw)
    def __str__(self):
        return '<DbField %s>'%self.name

class RowField(Field):
    # PE Virtualna_varijabla_redka
    def __init__(self,block,name,type,**kw):
        kw['rowbuffer']=True
        kw['dbname']=None
        kw['dbtable']=None
        kw['param_in']=False
        kw['param_out']=False
        kw['globalname']=None
        Field.__init__(self,block,name,type,**kw)
    def __str__(self):
        return '<RowField %s>'%self.name

class BlockField(Field):
    # PE Virtualna_varijabla_bloka
    def __init__(self,block,name,type,**kw):
        kw['rowbuffer']=False
        kw['dbname']=None
        kw['dbtable']=None
        kw['param_in']=False
        kw['param_out']=False
        kw['globalname']=None
        Field.__init__(self,block,name,type,**kw)
    def __str__(self):
        return '<BlockField %s>'%self.name

class GlobalField(Field):
    # PE Globalna_varijabla
```

```
    def __init__(self,block,name,type,globalname,**kw):
        kw['rowbuffer']=False
        kw['dbname']=None
        kw['dbtable']=None
        kw['param_in']=False
        kw['param_out']=False
        kw['globalname']=globalname
        Field.__init__(self,block,name,type,**kw)
    def __str__(self):
        return '<GlobalField %s of %s>'%(self.name,self.globalname)

class IOParameter(Field):
    # PE Ulazno_izlazni_parametar
    def __init__(self,block,name,type,**kw):
        kw['rowbuffer']=False
        kw['dbname']=None
        kw['dbtable']=None
        kw['param_in']=True
        kw['param_out']=True
        kw['globalname']=None
        Field.__init__(self,block,name,type,**kw)
    def __str__(self):
        return '<IOParemeter %s>'%self.name

class InputParameter(Field):
    # PE Ulazni_parametar
    def __init__(self,block,name,type,**kw):
        kw['rowbuffer']=False
        kw['dbname']=None
        kw['dbtable']=None
        kw['param_in']=True
        kw['param_out']=False
        kw['globalname']=None
        Field.__init__(self,block,name,type,**kw)
    def __str__(self):
        return '<IOParemeter %s>'%self.name

class OutputParameter(Field):
    # PE Izlazni_parametar
    def __init__(self,block,name,type,**kw):
        kw['rowbuffer']=False
        kw['dbname']=None
```

```
        kw['dbtable']=None
        kw['param_in']=False
        kw['param_out']=True
        kw['globalname']=None
        Field.__init__(self,block,name,type,**kw)
    def __str__(self):
        return '<IOParemeter %s>'%self.name

class Recalc:
    ## E REKALKULACIJA
    # Recalculation object
    def __init__(self,block,targetfield,initargument,argument,sourcefieldlist=[],**kw):
        self.block = block
        ## A Blok
        self.block.recalclist.append(self)
        self.targetfield = targetfield
        ## A Ciljna_varijabla
        self.targetfield.recalc = self
        self.initargument = initargument
        ## A Argument_inicijalizacije
        self.argument = argument
        ## A Argument
        self.sourcefieldlist = sourcefieldlist
        ## A Lista_izvornih_varijabli
        for f in sourcefieldlist:
            f.recalclist.append(self)
    def process(self,rowref):
        ## M Procesiraj
        self.block.buffer.set_recalc(rowref,self.targetfield,self.block.getvalue(self.argument))
    def process_init(self,rowref):
        ## M Procesiraj_inicijalizaciju
        self.block.buffer.set_recalc(rowref,self.targetfield,self.block.getvalue(self.initargument))
```

MODUL: _base_join.py

```
import common

class Join:
    ## E Veza
    def __init__(self,block,left_table,right_table):
        self.left_table = left_table
        ## A Lijeva_tablica
```

```
        self.right_table = right_table
        ## A Desna_tablica
        self.type = None
        ## A Tip
        self.block = block
        ## A Blok
        block.joinlist.append(self)
        block.tablejoindict[left_table,right_table]=self
        block.righttablejoindict[right_table]=self

class InnerJoin(Join):
    def __init__(self,block,left_table,right_table):
        Join.__init__(self,block,left_table,right_table)
        self.type = common.SQL_INNER_JOIN

class LeftOuterJoin(Join):
    def __init__(self,block,left_table,right_table):
        Join.__init__(self,block,left_table,right_table)
        self.type = common.SQL_LEFT_OUTER_JOIN
```

MODUL: _base_sqlstatement.py

```
import common

class DirectSQL:
    # For use within SQL Statement
    def __init__(self,statement,argdict={}):
        self.statement = statement
        self.argdict = argdict

class SQLExpr:
    # For use within SQL Statement
    def __init__(self,statement,arglist=[]):
        self.statement = statement
        self.arglist = arglist

class SQLStatement:
    # Generic class
    ## E SQL Element
    def __init__(self,connection,tablelist,**kw):
        self.connection = connection
        ## A Konekcija
```

```
    self.tablelist = tablelist
    ## A Lista_tablica
    self.type = None
    ## A Tip
    self.block = None
    ## A Blok
    self.directsql = kw.get('directsql',None)
    ## A Direktni_sql
    self.statement=''
    ## A Naredba
    self.argdict={}
    ## A Argumenti
    self.reexecuting = True # Execute every time True,False (Must set True for Links)
    ## A Ponovno_izvrsavanje
    self.fetching = False # True for select,false for insert,update,delete,callproc...
    ## A Dohvat
    self.fetchnum = None  # Nomber of rows to fetch in each cycle (None=All)
    ## A Slogova_za_dohvatiti
    self.fetchcycle = 0   # 0 = Not fetched yet, 1 = fetched ones... (rebuild reset)
    ## A Ciklus_dohvata
    self.connection.open_cursor(self)
    ## Overwrite: A Pregazi_spremnik T/F
    ## Delchanged: A Izbrisi_promjenjene_podatke T/F
    ## Deleterow: A Izbrisi_redak_spremnika T/F
def block_assign(self,block):
    self.block = block
    block.sqlstatementlist.append(self)
def close(self):
    ## M Zatvori
    self.connection.close_cursor(self)
def rebuild(self):
    ## M Izgradi
    ## Tu je razlika
    if self.directsql is not None:
        # Direct SQL as is: no additional changes are possible
        self.statement = self.directsql.statement
        self.argdict = self.directsql.argdict
        return
    self.statement = ''
    self.argdict={}
    self.fieldlist = []
    ## A Lista_polja_select_naredbe
```

```
self.groupbylist = []
## A Lista_polja_group_by_klauzule
self.aggfnclist = []
## A Lista_agregatnih_funkcija
self.wherecondlist = []
## A Lista_uvjeta_where_klauzule
self.havingcondlist = []
## A Lista_uvjeta_having_klauzule
self.joinconddict = {}
## A Niz_uvjeta_join_klauzule
self.rowreflist = []
## A Lista_polja_identifikatora_redka
self.updatelist = []
## A Lista_polja_update_naredbe
self.insertlist = []
## A Lista_polja_insert_naredbe
self.orderbylist = []
## A Lista_polja_order_by_klauzule
# fields
orderbydict = {}
for f in self.block.fieldlist:
   if f.dbtable in self.tablelist and f.select:
      self.fieldlist.append(f)
      if f.groupby:
         self.groupbylist.append(f)
      if f.aggfnc is not None:
         self.aggfnclist.append(f)
      if f.rowref:
         self.rowreflist.append(f)
      if f.insert:
         self.insertlist.append(f)
      if f.update:
         self.updatelist.append(f)
      if f.orderby is not None:
         orderbydict[f.orderby]=f
obkeylist = orderbydict.keys()
obkeylist.sort()
for key in obkeylist:
   self.orderbylist.append(orderbydict[key])
# conditions
for c in self.block.conditionlist:
   if c.table in self.tablelist:
```

```
            if c.join is not None:
                self.joinconddict[c.join] = self.joinconddict.get(c.join,[]) + [c]
            elif c.having:
                self.havingcondlist.append(c)
            else:
                self.wherecondlist.append(c)
    def execute(self,*l,**kw):
        ## M Izvrsi
        return self.connection.execute(self,*l,**kw)
    def buffer_process(self,*l,**kw):
        ## M Procesiraj_spremnik
        ## Tu je razlika
        pass
    def get_rowref(self,*l,**kw):
        return kw.get('rowref',self.block.rowref)
    def recalc_argdict(self,*l,**kw):
        ## M Rekalkuliraj_argumente
        # recalculate argument values (expressions only)
        retdict = {}
        rowref = kw.get('rowref',self.block.rowref)
        for key in self.argdict.keys():
            retdict[key]=self.argdict[key].get(rowref=rowref,method=common.CONV_BUF2DB)
# Problem with Decimal
        return retdict
    def show(self):
        return str((self.statement,self.argdict))
    def empty_fieldlist(self,rowref):
        for field in self.fieldlist:
            self.block.buffer.set(rowref,field,None,fetched=True)
            self.block.buffer.delchanged(rowref,field)
    def __str__(self):
        return '<%s on %s>'%(self.__class__.__name__,str(self.tablelist))
    def get(self):
        return self.statement,self.argdict

class SQLSelect(SQLStatement):
    def __init__(self,block,connection,tablelist,**kw):
        SQLStatement.__init__(self,connection,tablelist,**kw)
        self.type = common.SQL_SELECT
        self.fetching = True
        self.fetchnum = kw.get('fetchnum',self.fetchnum)
        self.reexecuting = kw.get('reexecuting',False)
```

```
        self.overwrite = kw.get('overwrite',True)
        self.block_assign(block)
        common.makefieldable(self,block,kw.get('field',None))
    def rebuild(self):
        SQLStatement.rebuild(self)
        # generating sql
        if self.directsql is None:
            self.statement,self.argdict=self.connection.sql.gen_sql_select(self)
        self.connection.rebuild(self)
    def buffer_process(self,**kw):
        # fetch data from db and update buffer
        if self.overwrite:
            rowref = kw.get('rowref',self.block.rowref)
        else:
            rowref = common.ROWREF_NO_RECORD
        data = self.execute(rowref=rowref)
        if self.overwrite:
            self.empty_fieldlist(rowref)
        if data is not None and len(data)>0:
            for row in data:
                d = {}
                for i in xrange(len(self.fieldlist)):
                    d[self.fieldlist[i]]=row[i]  # d[field]=value
                if not self.overwrite:
                    rowref = self.block.buffer.create_rowref(self.rowreflist,d)
                    self.block.buffer.create_row(rowref)
                for field in d.keys():
                    value = field.convert(d.get(field,None),common.CONV_DB2BUF) # conversion
                    self.block.buffer.set(rowref,field,value,fetched=True)
        return data

class SQLSelectRefetch(SQLStatement):
    def __init__(self,block,connection,tablelist,**kw):
        SQLStatement.__init__(self,connection,tablelist,**kw)
        self.type = common.SQL_SELECT_REFETCH
        self.fetching = True
        self.fetchnum = 1
        self.delchanged = kw.get('delchanged',True) # delete changed buffer
        self.block_assign(block)
        common.makefieldable(self,block,kw.get('field',None))
    def rebuild(self):
        SQLStatement.rebuild(self)
```

```
        # generating sql
        if self.directsql is None:
            self.statement,self.argdict=self.connection.sql.gen_sql_select(self)
        self.connection.rebuild(self)
    def buffer_process(self,**kw):
        rowref = kw.get('rowref',self.block.rowref)
        if rowref is common.ROWREF_NO_RECORD:
            return None
        data = self.execute(rowref=rowref)
        self.empty_fieldlist(rowref)
        if data is not None and len(data)>0:
            row = data[0]
            d = {}
            for i in xrange(len(self.fieldlist)):
                d[self.fieldlist[i]]=row[i]  # d[field]=value
            for field in d.keys():
                value = field.convert(d.get(field,None),common.CONV_DB2BUF) # conversion
                self.block.buffer.set(rowref,field,value,fetched=True)
                if self.delchanged:
                    self.block.buffer.delchanged(rowref,field)
        return data

class SQLLock(SQLStatement):
    def __init__(self,block,connection,tablelist,**kw):
        SQLStatement.__init__(self,connection,tablelist,**kw)
        self.type = common.SQL_LOCK
        self.reexecuting = False
        self.fetching = True
        self.fetchnum = kw.get('fetchnum',self.fetchnum)
        self.overwrite = kw.get('overwrite',True)
        self.block_assign(block)
        common.makefieldable(self,block,kw.get('field',None))
    def rebuild(self):
        SQLStatement.rebuild(self)
        # generating sql
        if self.directsql is None:
            self.statement,self.argdict=self.connection.sql.gen_sql_lock(self)
        self.connection.rebuild(self)
    def buffer_process(self,**kw):
        # fetch data from db and update buffer
        if self.overwrite:
            rowref = kw.get('rowref',self.block.rowref)
```

```
        else:
            rowref = common.ROWREF_NO_RECORD
        data = self.execute(rowref=rowref)
        if self.overwrite:
            self.empty_fieldlist(rowref)
        if data is not None and len(data)>0:
            for row in data:
                d = {}
                for i in xrange(len(self.fieldlist)):
                    d[self.fieldlist[i]]=row[i]  # d[field]=value
                if not self.overwrite:
                    rowref = self.block.buffer.create_rowref(self.rowreflist,d)
                    self.block.buffer.create_row(rowref)
                for field in d.keys():
                    value = field.convert(d.get(field,None),common.CONV_DB2BUF) # conversion
                    self.block.buffer.set(rowref,field,value,fetched=True)
        return data

class SQLLockRefetch(SQLStatement):
    def __init__(self,block,connection,tablelist,**kw):
        SQLStatement.__init__(self,connection,tablelist,**kw)
        self.type = common.SQL_LOCK_REFETCH
        self.fetching = True
        self.fetchnum = 1
        self.delchanged = kw.get('delchanged',True) # delete changed buffer
        self.block_assign(block)
        common.makefieldable(self,block,kw.get('field',None))
    def rebuild(self):
        SQLStatement.rebuild(self)
        # generating sql
        if self.directsql is None:
            self.statement,self.argdict=self.connection.sql.gen_sql_lock(self)
        self.connection.rebuild(self)
    def buffer_process(self,**kw):
        # fetch data from db and update buffer for specified rowref
        rowref = kw.get('rowref',self.block.rowref)
        if rowref is common.ROWREF_NO_RECORD:
            return None
        data = self.execute(rowref=rowref)
        self.empty_fieldlist(rowref)
        if data is not None and len(data)>0:
            row = data[0]
```

```
            d = {}
            for i in xrange(len(self.fieldlist)):
               d[self.fieldlist[i]]=row[i]  # d[field]=value
            for field in d.keys():
               value = field.convert(d.get(field,None),common.CONV_DB2BUF) # conversion
               self.block.buffer.set(rowref,field,value,fetched=True)
               if self.delchanged:
                  self.block.buffer.delchanged(rowref,field)
         return data

class SQLInsertRow(SQLStatement):
   def __init__(self,block,connection,tablelist,**kw):
      SQLStatement.__init__(self,connection,tablelist,**kw)
      self.type = common.SQL_INSERT_ROW
      self.fetching = False
      self.block_assign(block)
      common.makefieldable(self,block,kw.get('field',None))
   def rebuild(self):
      SQLStatement.rebuild(self)
      # generating sql
      if self.directsql is None:
         self.statement,self.argdict=self.connection.sql.gen_sql_insert(self)
      self.connection.rebuild(self)
   def buffer_process(self,**kw):
      rowref = kw.get('rowref',self.block.rowref)
      rowid = self.execute(rowref=rowref)
      # write fields from changed to fetched buffer
      for field in self.fieldlist:
         self.block.buffer.writeset(rowref,field)
      if len(self.tablelist)==1:
         table = self.tablelist[0]
         # check if there is a rowref field that should be refetched
         if table.rowid is not None:
            value = table.rowid.convert(rowid,common.CONV_DB2BUF) # conversion
            self.block.buffer.set(rowref,table.rowid,value,fetched=True)
         # check if table is main table and rowref is new_record
         if table.type == common.MAIN_TABLE and rowref ==
common.ROWREF_NEW_RECORD:
            # change rowref
            newrowref = self.block.buffer.create_rowref(self.rowreflist,\
                  self.block.buffer.fetchedbuffer[rowref])
            self.block.buffer.change_rowref(rowref,newrowref)
```

```
            self.block.rowref = newrowref
        return rowid

class SQLUpdateRow(SQLStatement):
    def __init__(self,block,connection,tablelist,**kw):
        SQLStatement.__init__(self,connection,tablelist,**kw)
        self.type = common.SQL_UPDATE_ROW
        self.fetching = False
        self.block_assign(block)
        common.makefieldable(self,block,kw.get('field',None))
    def rebuild(self):
        SQLStatement.rebuild(self)
        # generating sql
        if self.directsql is None:
            self.statement,self.argdict=self.connection.sql.gen_sql_update(self)
        self.connection.rebuild(self)
    def buffer_process(self,**kw):
        # fetch data from db and update buffer for specified rowref
        rowref = kw.get('rowref',self.block.rowref)
        if rowref is common.ROWREF_NO_RECORD:
            return None
        data = self.execute(rowref=rowref)
        for field in self.fieldlist:
            self.block.buffer.writeset(rowref,field)
        return data

class SQLDeleteRow(SQLStatement):
    def __init__(self,block,connection,tablelist,**kw):
        SQLStatement.__init__(self,connection,tablelist,**kw)
        self.type = common.SQL_DELETE_ROW
        self.fetching = False
        self.block_assign(block)
        common.makefieldable(self,block,kw.get('field',None))
        self.deleterow = kw.get('deleterow',False)
    def rebuild(self):
        SQLStatement.rebuild(self)
        # generating sql
        if self.directsql is None:
            self.statement,self.argdict=self.connection.sql.gen_sql_delete(self)
        self.connection.rebuild(self)
    def buffer_process(self,**kw):
        # fetch data from db and update buffer for specified rowref
```

```
        rowref = kw.get('rowref',self.block.rowref)
        if rowref is common.ROWREF_NO_RECORD:
            return None
        data = self.execute(rowref=rowref)
        if self.deleterow:
            self.block.buffer.delete_row(rowref)
        return data
```

MODUL: _base_table.py

```
import common

class Table:
    ## ELEM Tablica
    def __init__(self,block,connection,alias,dbtablename=None,**kw):
        self.block = block
        ## ATR Blok
        block.tablelist.append(self)
        block.tablealiasdict[alias]=self
        block.tablefieldlist[self]=[]
        self.connection = connection
        ## Atr konekcija
        self.alias = alias
        ## Atr Alias
        self.dbtablename = dbtablename # if None: direct sql
        ## Atr Ime_u_bazi
        self.subquery = kw.get('subquery',None) # SQLExpr
        ## Atr SQL_Podupit
        self.type = None
        ## Atr Tip_tablice
        self.rowid = None

class MainTable(Table):
    def __init__(self,block,connection,alias,dbtablename=None,**kw):
        Table.__init__(self,block,connection,alias,dbtablename,**kw)
        self.type = common.MAIN_TABLE
        self.block.maintable = self
    def __str__(self):
        return '<MainTable %s %s>'%(self.dbtablename,self.alias)
    def __repr__(self):
        return self.__str__()
```

```
class LinkTable(Table):
   def __init__(self,block,connection,alias,dbtablename=None,**kw):
      Table.__init__(self,block,connection,alias,dbtablename,**kw)
      self.type = common.LINK_TABLE
   def __str__(self):
      return '<LinkTable %s %s>'%(self.dbtablename,self.alias)
   def __repr__(self):
      return self.__str__()
```

MODUL: common.py

```
import exceptions

class AlatUnknownItemError(exceptions.Exception):
   def __init__(self,itemname):
      self.itemname = itemname
   def __str__(self):
      return repr(self.itemname)

def nvl(value,default):
   if value is None:
      return default
   return value

def cond(condexpr,iftrue=True,iffalse=False):
   if condexpr:
      return iftrue
   return iffalse

def boolean(cond):
   if cond in (True,False,None):
      return cond
   return cond.get()

def eval_arglist(arglist):
   # evaluate list of expressions and fields
   l = []
   for element in arglist:
      l.append(element.get())
   return l

def eval_argdict(argdict):
```

```
    # evaluate dict of expressions and fields
    d = {}
    for key in argdict.keys():
        d[key] = argdict[key].get()
    return d

COND_OP = 1
COND_CLAUSE = 2
COND_EXPR = 3

OP_EQ = '='
OP_LT = '<'
OP_LE = '<='
OP_GT = '>'
OP_GE = '>='
OP_NE = '<>'
OP_ISNULL = 'IS NULL'
OP_IN = 'IN'
OP_BETWEEN = 'BETWEEN'
OP_LIKE = 'LIKE'

MAIN_TABLE = 0
LINK_TABLE = 1

SQL_SELECT = 'SA'
SQL_SELECT_REFETCH = 'SR'
SQL_LOCK_REFETCH = 'LR'
SQL_LOCK = 'LA'
SQL_INSERT_ROW = 'IR'
SQL_UPDATE_ROW = 'UR'
SQL_DELETE_ROW = 'DR'

COND_STATEMENT_LIST = [SQL_SELECT,SQL_LOCK]

SQL_INNER_JOIN = 'IJ'
SQL_LEFT_OUTER_JOIN = 'LOJ'

ROWREF_NO_RECORD = False  # Rowref is pointing on no record
ROWREF_NEW_RECORD = None  # Rowref is pointing on new record (not written yet)

EVT_BLOCK_BEFORE = 'EVENT_BLOCK_BEFORE'
EVT_BLOCK_AFTER = 'EVENT_BLOCK_AFTER'
```

```
EVT_RECORD_BEFORE = 'EVENT_RECORD_BEFORE'
EVT_RECORD_AFTER = 'EVENT_RECORD_AFTER'
EVT_FIELD_BEFORE = 'EVENT_FIELD_BEFORE'
EVT_FIELD_AFTER = 'EVENT_FIELD_AFTER'
EVT_BLOCK_FETCH = 'EVENT_BLOCK_FETCH'
EVT_RECORD_FETCH = 'EVENT_RECORD_FETCH'
EVT_EXTERNAL = 'EVENT_EXTERNAL'
EVT_BLOCK_DESCRIPTOR = 'EVENT_BLOCK_DESCRIPTOR'
EVT_FIELD_ACTION = 'EVENT_FIELD_ACTION'
EVT_CHANGE = 'EVENT_FIELD_CHANGE'
EVT_ERROR = 'EVENT_ERROR'
EVT_BLOCK_EXIT = 'EVENT_BLOCK_EXIT'

CONV_DB2BUF = 'DB2BUF'
CONV_BUF2DB = 'BUF2DB'
CONV_BUF2STR = 'BUF2STR'
CONV_STR2BUF = 'STR2BUF'

def makefieldable(object,block,field):
   object.field = field
   if field is not None:
      block.fieldablelist.append(object)
```

MODUL: db.py

```
import common

class Connection:
   ## KLASA: Konekcija_na_bazu
   def __init__(self):
      ## METODA: Konstruktor
      self.sql = SQLGenerator()
      self.conn = None
   def open_cursor(self,sqlstatement):
      ## METODA: Otvori_kursor
      pass
   def close_cursor(self,sqlstatement):
      ## METODA: Zatvori_kursor
      pass
   def begin_transaction(self):
      ## METODA: Zapocni_transakciju
      pass # TODO
```

```
    def commit(self):
        ## METODA: Commit
        pass
    def rollback(self):
        ## METODA: Rollback
        pass
    def close(self):
        ## METODA: Zatvori
        pass
    def rebuild(self,sqlstatement):
        # called at block rebuild
        sqlstatement.fetchcycle = 0
    def execute(self,sqlstatement):
        ## METODA: Izvrsi
        return []

class SQLGenerator:
    def __init__(self):
        self.nl = '\n'
    def delimiter_clean(self,stm,delimiter=','):
        j = len(delimiter)
        if stm[-j:]==delimiter:
            return stm[:-j]
        return stm
    def fmt_arg(self,block,object):
        if block.isfield(object) and object.dbtable is not None and object.dbname is not None:
            return self.get_select_fieldname(object)
        return '%s',[object]
    def norm(self,stm,arglist,prefix='X'):
        try:
            # %s is placeholder
            reflist = []
            outdict = {}
            argdict = {}
            i = 1
            for x in arglist:
                if not x in outdict:
                    argkey = '%s%d'%(prefix,len(outdict.keys())+1) # for use in argdict
                    stmkey = ':'+argkey  # for use in statement
                    argdict[argkey]=x
                    outdict[x]=stmkey
                reflist.append(outdict[x])
```

```
            outstm = stm%tuple(reflist)
            return outstm,argdict
        except:
            print stm,reflist
            raise 'Norm error!'
    def get_select_fieldname(self,field,noalias=False):
        if field.sqlselect is not None:
            return field.sqlselect.statement,field.sqlselect.arglist
        if noalias:
            return  '%s'%field.dbname,[]
        return  '%s.%s'%(field.dbtable.alias,field.dbname),[]
    def get_select_tablename(self,table):
        if table.dbtablename is not None:
            return '%s %s'%(table.dbtablename,table.alias),[]
        else:
            #subquery
            return '(%s) %s'%(table.subquery.statement,table.alias),table.subquery.arglist
    def listtable_join_sort(self,sqlstatement):
        right = []
        left = []
        leftdict = {}
        for table in sqlstatement.tablelist:
            join = sqlstatement.block.righttablejoindict.get(table,None)
            if join is not None:
                left.append(join.left_table)
                right.append(join.right_table)
                leftdict[join.left_table] = leftdict.get(join.left_table,[])\
                                        +[join.right_table]
        roots = []
        for table in left:
            if not table in right+roots:
                roots.append(table)
        for table in sqlstatement.tablelist:
            if not table in roots and not table in right and not table in left:
                roots.append(table)
        answer = []
        for roottable in roots:
            ltables = leftdict.get(roottable,[])
            answer += [roottable]
            if len(ltables)>0:
                inlist = [roottable]
                while len(inlist)>0:
```

```
            table = inlist.pop(0)
            ltables = leftdict.get(table,[])
            inlist += ltables
            for righttable in ltables:
                answer += [righttable]
    return answer
# SQL Statement generators
def gen_sql_insert(self,sqlstatement):
    stm = 'INSERT INTO'
    arglist = []
    # Tables
    table = sqlstatement.tablelist[0]
    stm += ' %s'%table.dbtablename
    # fields,values
    if len(sqlstatement.insertlist)>0:
        s1 = ''
        s2 = ''
        for field in sqlstatement.insertlist:
            s1 += ' '+field.dbname+','
            if field.sqlinsert is None:
                s2 += ' %s,'
                arglist.append(field)
            else:
                s2 += ' '+field.sqlinsert.statement+','
                arglist += field.sqlinsert.arglist
        s1 = self.delimiter_clean(s1)
        s2 = self.delimiter_clean(s2)
        stm += ' ('+s1+') VALUES ('+s2+')'
    # Finally
    return self.norm(stm,arglist)
def gen_sql_update(self,sqlstatement):
    stm = 'UPDATE '
    arglist = []
    # Tables
    table = sqlstatement.tablelist[0]
    stm += ' %s'%table.dbtablename
    # Set fields
    if len(sqlstatement.updatelist)>0:
        stm += self.nl + ' SET'
        for field in sqlstatement.updatelist:
            if field.sqlupdate is None:
                stm += ' '+field.dbname+' = %s,'
```

```
                arglist.append(field)
            else:
                stm += ' '+field.dbname+' = '+field.sqlupdate.statement+','
                arglist += field.sqlupdate.arglist
        stm = self.delimiter_clean(stm)
    # Where
    wstm,warglist = self.gen_where_row_clause(sqlstatement,noalias=True)
    stm += wstm
    arglist += warglist
    # Finally
    return self.norm(stm,arglist)
def gen_sql_delete(self,sqlstatement):
    stm = 'DELETE FROM'
    arglist = []
    # Tables
    table = sqlstatement.tablelist[0]
    stm += ' %s'%table.dbtablename
    # Where
    wstm,warglist = self.gen_where_row_clause(sqlstatement,noalias=True)
    stm += wstm
    arglist += warglist
    # Finally
    return self.norm(stm,arglist)
def gen_sql_lock(self,sqlstatement):
    return self.gen_sql_select(sqlstatement,forupdate='FOR UPDATE NOWAIT')
def gen_sql_select(self,sqlstatement,forupdate=None):
    stm = 'SELECT'
    arglist = []
    # Fields
    for field in sqlstatement.fieldlist:
        fstm,farglist = self.get_select_fieldname(field)
        if field.aggfnc is None:
            stm +=  ' %s as %s,'%(fstm,field.name)
            arglist += farglist
        else:
            stm +=  ' %s(%s) as %s,'%(field.aggfnc,fstm,field.name)
            arglist += farglist
    stm = self.delimiter_clean(stm)
    # From clause
    stm += self.nl + ' FROM'
    prevtables = []
    join = None
```

```
joinsorttablelist = self.listtable_join_sort(sqlstatement)
for table in joinsorttablelist:
    tstm,targlist = self.get_select_tablename(table)
    join = sqlstatement.block.righttablejoindict.get(table,None)
    if join is None or join.left_table not in prevtables:
        stm += ' %s,'%tstm
        arglist+=targlist
    else:
        stm = self.delimiter_clean(stm)
        jstm = 'INNER JOIN'
        if join.type == common.SQL_LEFT_OUTER_JOIN:
            jstm = 'LEFT OUTER JOIN'
        condlist = sqlstatement.joinconddict.get(join,[])
        onstm,onarglist = '',[]
        if len(condlist)>0:
            onstm = ' ON ('
            for cond in condlist:
                cstm,carglist = self.gen_cond(sqlstatement,cond)
                onstm += cstm+' AND '
                onarglist += carglist
            onstm = self.delimiter_clean(onstm,'AND ')
            onstm += ')'
        stm += ' %s %s%s,'%(jstm,tstm,onstm)
        arglist += targlist+onarglist
    prevtables.append(table)
stm = self.delimiter_clean(stm)
# Where clause
if sqlstatement.type in (common.SQL_SELECT,common.SQL_LOCK):
    wstm,warglist = self.gen_where_all_clause(sqlstatement)
else:
    wstm,warglist = self.gen_where_row_clause(sqlstatement)
stm += wstm
arglist += warglist
# groupby
if len(sqlstatement.groupbylist)>0:
    stm += self.nl + ' GROUP BY'
    for field in sqlstatement.groupbylist:
        fstm,farglist = self.get_select_fieldname(field)
        stm +=  ' %s,'%fstm
        arglist += farglist
    stm = self.delimiter_clean(stm)
# For update clause
```

```
    if forupdate is not None:
        stm += self.nl + ' '+forupdate
    # Having clause
    if sqlstatement.type in (common.SQL_SELECT,common.SQL_LOCK):
        hstm,harglist = self.gen_having_all_clause(sqlstatement)
        stm += hstm
        arglist += harglist
    # Order by clause
    if sqlstatement.type in (common.SQL_SELECT,common.SQL_LOCK):
        ostm,oarglist = self.gen_orderby_clause(sqlstatement)
        stm += ostm
        arglist += oarglist
    # Finally
    return self.norm(stm,arglist)
def gen_where_row_clause(self,sqlstatement,noalias=False):
    stm = ''
    arglist = []
    if len(sqlstatement.rowreflist)>0:
        stm += self.nl + ' WHERE'
        for field in sqlstatement.rowreflist:
            fstm,farglist = self.get_select_fieldname(field,noalias)
            stm += ' '+fstm+' = %s'
            arglist += farglist
            arglist.append(field)
            stm += '  AND'
        stm = self.delimiter_clean(stm,'  AND')
    return stm,arglist
def gen_cond(self,sqlstatement,cond):
    stm = ''
    arglist = []
    if cond.type == common.COND_OP:
        fstm,farglist = self.get_select_fieldname(cond.field)
        if cond.notop:
            stm += ' NOT'
        if cond.operator.name == common.OP_BETWEEN:
            amin,lmin = self.fmt_arg(sqlstatement.block,cond.arglist[0])
            amax,lmax = self.fmt_arg(sqlstatement.block,cond.arglist[1])
            stm += ' '+fstm+' BETWEEN %s AND %s'%(amin,amax)
            arglist += farglist+lmin+lmax
        elif cond.operator.name == common.OP_IN:
            al = '('
            for x in cond.arglist:
```

```
                afmt,lfmt = self.fmt_arg(sqlstatement.block,x)
                al += ' %s,'%afmt
                arglist += lfmt
            al = self.delimiter_clean(al)+')'
            stm += ' '+fstm+' IN '+al
            arglist += farglist
        elif cond.operator.name == common.OP_ISNULL:
            stm += ' '+fstm+' IS NULL'
            arglist += farglist
        else:
            afmt,lfmt = self.fmt_arg(sqlstatement.block,cond.arglist[0])
            stm += ' '+fstm+' '+cond.operator.name+' %s'%afmt
            arglist += farglist + lfmt
    elif cond.type == common.COND_CLAUSE:
        afmtlist = []
        for x in cond.arglist:
            afmt,lfmt = self.fmt_arg(sqlstatement.block,x)
            afmtlist.append(afmt)
            arglist+=lfmt
        if len(afmtlist)>0:
            clause = cond.clause%tuple(afmtlist)
        else:
            clause = cond.clause
        stm += ' '+clause
    return stm,arglist
def gen_where_all_clause(self,sqlstatement):
    stm = ''
    arglist = []
    if len(sqlstatement.wherecondlist)>0:
        stm += self.nl + ' WHERE'
        for cond in sqlstatement.wherecondlist:
            cstm,carglist = self.gen_cond(sqlstatement,cond)
            stm += cstm + '  AND'
            arglist += carglist
        stm = self.delimiter_clean(stm,'  AND')
    return stm,arglist
def gen_having_all_clause(self,sqlstatement):
    stm = ''
    arglist = []
    if len(sqlstatement.havingcondlist)>0:
        stm += self.nl + ' HAVING'
        for cond in sqlstatement.havingcondlist:
```

```
            cstm,carglist = self.gen_cond(sqlstatement,cond)
            stm += cstm + '  AND'
            arglist += carglist
        stm = self.delimiter_clean(stm,'  AND')
      return stm,arglist
   def gen_orderby_clause(self,sqlstatement):
      stm = ''
      arglist = []
      if len(sqlstatement.orderbylist)>0:
         stm += self.nl + ' ORDER BY'
         for field in sqlstatement.orderbylist:
            fstm,farglist = self.get_select_fieldname(field)
            stm += ' '+fstm+common.cond(field.descending,' DESC','')+','
            arglist += farglist
         stm = self.delimiter_clean(stm)
      return stm,arglist
```

MODUL: db_sqlite3.py

```
import sqlite3
import common
import db

class Connection(db.Connection):
   def __init__(self,database):
      db.Connection.__init__(self)
      self.database = database
      self.conn = sqlite3.connect(database)
      self.isolation_level = 'DEFERRED'
   def open_cursor(self,sqlstatement):
      sqlstatement.cursor = self.conn.cursor()
   def close_cursor(self,sqlstatement):
      sqlstatement.cursor.close()
   def commit(self):
      self.conn.commit()
   def rollback(self):
      self.conn.rollback()
   def close(self):
      self.conn.close()
   def execute(self,sqlstatement,*l,**kw):
      if sqlstatement.reexecuting or sqlstatement.fetchcycle == 0:
         rowref = sqlstatement.get_rowref(*l,**kw)
```

```
            sqlstatement.cursor.execute(sqlstatement.statement,\
                        sqlstatement.recalc_argdict(rowref=rowref))
            if sqlstatement.type == common.SQL_INSERT_ROW:
               # returns last inserted rowid accornding to DB-API 2.0
               return sqlstatement.cursor.lastrowid
         if sqlstatement.fetching:
            sqlstatement.fetchcycle+=1
            if sqlstatement.fetchnum is None:
               return sqlstatement.cursor.fetchall()
            elif sqlstatement.fetchnum == 1:
               ret = sqlstatement.cursor.fetchone()
               if ret is not None:
                  return [ret]
               return []
            else:
               return sqlstatement.cursor.fetchmany(sqlstatement.fetchnum)
         return []
```

MODUL: lib.py

```
import sqlite3
import common
import db

class Connection(db.Connection):
   def __init__(self,database):
      db.Connection.__init__(self)
      self.database = database
      self.conn = sqlite3.connect(database)
      self.isolation_level = 'DEFERRED'
   def open_cursor(self,sqlstatement):
      sqlstatement.cursor = self.conn.cursor()
   def close_cursor(self,sqlstatement):
      sqlstatement.cursor.close()
   def commit(self):
      self.conn.commit()
   def rollback(self):
      self.conn.rollback()
   def close(self):
      self.conn.close()
   def execute(self,sqlstatement,*l,**kw):
      if sqlstatement.reexecuting or sqlstatement.fetchcycle == 0:
```

```
    rowref = sqlstatement.get_rowref(*l,**kw)
    sqlstatement.cursor.execute(sqlstatement.statement,\
                sqlstatement.recalc_argdict(rowref=rowref))
    if sqlstatement.type == common.SQL_INSERT_ROW:
        # returns last inserted rowid accornding to DB-API 2.0
        return sqlstatement.cursor.lastrowid
if sqlstatement.fetching:
    sqlstatement.fetchcycle+=1
    if sqlstatement.fetchnum is None:
        return sqlstatement.cursor.fetchall()
    elif sqlstatement.fetchnum == 1:
        ret = sqlstatement.cursor.fetchone()
        if ret is not None:
            return [ret]
        return []
    else:
        return sqlstatement.cursor.fetchmany(sqlstatement.fetchnum)
return []
```

PRILOG 2: IZVORNI KOD IMPLEMENTACIJE KORISNIČKOG SUČELJA

MODUL: __init__.py

```
from _containers import Application, Frame, AuiMDIParentFrame, AuiMDIChildFrame
from _panel import Panel, ScrolledPanel
from _sizer import Sizer, HSizer, VSizer
from _widgets import Label, Button, Edit
from _interface import interface
```

MODUL: _common.py

```
import wx
import wx.grid
import string

class Widget:
    def __init__(self,panel):
        self.panel = panel
        self.field = None
        self.init_events()
    def setminsize(self,width,height):
        self.SetMinSize((width,height))
    def setmaxsize(self,width,height):
        self.SetMaxSize((width,height))
    def init_events(self):
        # binds event handlers
        self.Bind(wx.EVT_SET_FOCUS,self.on_set_focus)
        self.Bind(wx.EVT_KEY_DOWN,self.panel.on_key_down)
    def on_set_focus(self,event=None,**kw):
        self.panel.process_focus(self.panel.block.rowref,self.field)
        if event is not None:
            event.Skip()
    def goto(self,*l,**kw):
        self.SetFocus()
    def go_forward(self):
        self.Navigate(wx.NavigationKeyEvent.IsForward)
    def go_backward(self):
```

```
        self.Navigate(wx.NavigationKeyEvent.IsBackward)
    def on_forward(self):
        pass
    def on_backward(self):
        pass
    def refresh(self):
        pass
    def on_exit(self):
        pass

class Grid(wx.grid.Grid,Widget):
    def __init__(self,panel):
        wx.grid.Grid.__init__(self,panel,-1)
        Widget.__init__(self,panel)
        self.panel.add_widget(self)
        if panel.grid is not None:
            raise "Only one grid allowed in panel",panel
        panel.grid = self
        self.last_focus = None
        # Define cell editor
        self.grideditor = _GridCellEditor(self)
        self.SetDefaultEditor(self.grideditor)
        self.in_editor = False
    def set_data(self,griddataobj):
        self.data = griddataobj
        self.SetTable(self.data, False)
    def refresh_scrollbars(self):
        h,w = self.GetSize()
        self.SetSize((h+1,w))
        self.SetSize((h,w))
        self.ForceRefresh()
    def init_events(self):
        Widget.init_events(self)
        self.Bind(wx.grid.EVT_GRID_SELECT_CELL,self.on_select_cell)
        # Focus set
        self.GetGridCornerLabelWindow().Bind(wx.EVT_SET_FOCUS,self.on_set_focus)
        self.GetGridColLabelWindow().Bind(wx.EVT_SET_FOCUS,self.on_set_focus)
        self.GetGridRowLabelWindow().Bind(wx.EVT_SET_FOCUS,self.on_set_focus)
        self.GetGridWindow().Bind(wx.EVT_SET_FOCUS,self.on_set_focus)
        # Key Events
        self.Bind(wx.EVT_KEY_DOWN,self.on_key_down)
    def on_key_down(self,event):
```

```
    key = event.GetKeyCode()
    col,row = self.get_pos()
    if key == wx.WXK_TAB:
        if event.ShiftDown(): # Backward
            if col==0:
                self.go_backward()
            else:
                self.MoveCursorLeft(False)
        else: # Forward
            if col+1==self.GetNumberCols():
                self.go_forward()
            else:
                self.MoveCursorRight(False)
        return
    elif key == wx.WXK_RETURN:
        self.EnableCellEditControl(not self.in_editor)
        return
    event.Skip()
def on_select_cell(self,event,**kw):
    if not self.in_editor:
        newpos = (event.GetCol(),event.GetRow())
        oldpos = self.get_pos()
        self.on_set_focus(newpos=newpos)
        event.Skip()
        if self.pos2focus(newpos) == self.panel.block.get_ref() or \
                self.panel.block.block_before_executed is False:
            pass
        else:
            pass
    else:
        event.Skip()
        newpos = (event.GetCol(),event.GetRow())
        oldpos = self.get_pos()
        self.on_set_focus(newpos=newpos)
def get_pos(self):
    return self.GetGridCursorCol(),self.GetGridCursorRow()
def on_set_focus(self,event=None,**kw):
    # *FOCUS*
    pos = kw.get('newpos',self.get_pos())
    if pos<>(-1,-1):
        if not (event is not None and self.in_editor):
            #rowref,fieldref = self.pos2focus(pos)
```

```
            newfocus = self.pos2focus(pos)
            if newfocus is not None:
               rowref,fieldref = newfocus
               self.panel.process_focus(rowref,fieldref)
               ctrlfocus = self.pos2focus(pos) # See if pos target has change
               if newfocus <> ctrlfocus: # changed!
                  pass
      if event is not None:
         event.Skip()
   def goto(self,col,row):
      if self.panel.get_current_widget() is not self:
         self.SetFocus()
      self.SetGridCursor(row,col)
   def refresh(self):
      pass # should be overridden
   def pos2focus(self,pos):
      return None
   def field2widget(self,field):
      return None

class _GridCellEditor(wx.grid.PyGridCellEditor):
   def __init__(self,grid):
      wx.grid.PyGridCellEditor.__init__(self)
      self.grid = grid
      self.pos = None
   def Create(self, parent, id, evtHandler):
      self.textctrl = wx.TextCtrl(parent, id, "",\
         style = wx.TE_PROCESS_TAB)
      self.textctrl.Bind(wx.EVT_CHAR,self._on_evt_key_down)
      self.textctrl.SetInsertionPoint(0)
      self.SetControl(self.textctrl)
      if evtHandler:
         self.textctrl.PushEventHandler(evtHandler)
   def _on_evt_key_down(self,evt):
      key = evt.GetKeyCode()
      if key in (wx.WXK_UP,wx.WXK_DOWN):
         col,row = self.pos
         self.EndEdit(row,col,self.grid)
         self.grid.ProcessEvent(evt)
         return
      evt.Skip()
   def BeginEdit(self, row, col, grid):
```

```
        self.grid.in_editor = True
        self.pos = (col,row)
        self.startValue = grid.GetTable().GetValue(row, col)
        self.textctrl.SetValue(self.startValue)
        self.textctrl.SetInsertionPointEnd()
        self.textctrl.SetFocus()
    def EndEdit(self, row, col, grid):
        changed = False
        val = self.textctrl.GetValue()
        if val != self.startValue:
            changed = True
            grid.SetCellValue(row,col,val)
        self.startValue = ''
        self.textctrl.SetValue('')
        self.grid.in_editor = False
        return changed
    def Reset(self):
        self.textctrl.SetValue(self.startValue)
        self.textctrl.SetInsertionPointEnd()
    def acceptChar(self,keycode):
        return keycode < 256 and keycode >= 0 and chr(keycode) in string.printable \
            and keycode<>wx.WXK_TAB and keycode<>wx.WXK_RETURN
    def IsAcceptedKey(self, evt):
        control = self.grid
        keycode = evt.GetKeyCode()
        return self.acceptChar(keycode)
    def StartingKey(self, evt):
        keycode = evt.GetKeyCode()
        if self.acceptChar(keycode):
            control = self.grid
            self.textctrl.SetValue(chr(keycode))
            self.textctrl.SetInsertionPointEnd()
        else:
            evt.Skip()
    def StartingClick(self):
        if self.grid.IsCellEditControlEnabled():
            self.grid.DisableCellEditControl()
    def Destroy(self):
        self.base_Destroy()
    def Clone(self):
        return GridCellEditor(self.grid)
```

```
class _GridData(wx.grid.PyGridTableBase):
    def __init__(self,grid):
        wx.grid.PyGridTableBase.__init__(self)
        self.grid = grid
        self.even_attr = self.create_attr('black','white')
        self.odd_attr = self.create_attr('black','light yellow')
        self.lastsetvalue = None
    def create_attr(self,fg,bg):
        attr=wx.grid.GridCellAttr()
        attr.SetBackgroundColour(bg)
        attr.SetTextColour(fg)
        return attr
    def GetAttr(self, row, col, kind):
        attr = None
        if attr is None:
            attr = [self.even_attr, self.odd_attr][row % 2]
        attr.IncRef()
        return attr
    def GetNumberRows(self):
        return 1
    def GetNumberCols(self):
        return 1
    def IsEmptyCell(self, row, col):
        return False
    def GetValue(self, row, col):
        return str( (col, row) )
    def SetValue(self, row, col, value):
        pass
    def GetRowLabelValue(self,row):
        return 'R%d'%row
    def GetColLabelValue(self,col):
        return 'C%d'%col
    def update_values(self):
        msg                                                                                    =
wx.grid.GridTableMessage(self,wx.grid.GRIDTABLE_REQUEST_VIEW_GET_VALUES)
        self.grid.ProcessTableMessage(msg)
    def set_grid_size(self,cols,rows):
        curr_cols = self.grid.GetNumberCols()
        curr_rows = self.grid.GetNumberRows()
        self.grid.BeginBatch()
        # Rows
        if rows < curr_rows:
```

```
            # Delete
            msg = wx.grid.GridTableMessage(self,\
                wx.grid.GRIDTABLE_NOTIFY_ROWS_DELETED,\
                rows,curr_rows-rows)
            self.grid.ProcessTableMessage(msg)
        elif rows > curr_rows:
            # Append
            msg = wx.grid.GridTableMessage(self,\
                wx.grid.GRIDTABLE_NOTIFY_ROWS_APPENDED,\
                rows-curr_rows)
            self.grid.ProcessTableMessage(msg)
        # Cols
        if cols < curr_cols:
            # Delete
            msg = wx.grid.GridTableMessage(self,\
                wx.grid.GRIDTABLE_NOTIFY_COLS_DELETED,\
                cols,curr_cols-cols)
            self.grid.ProcessTableMessage(msg)
        elif cols > curr_cols:
            # Append
            msg = wx.grid.GridTableMessage(self,\
                wx.grid.GRIDTABLE_NOTIFY_COLS_APPENDED,\
                cols-curr_cols)
            self.grid.ProcessTableMessage(msg)
            # Adjust col width
            for c in xrange(self.grid.GetNumberCols()):
                w = self.get_col_width(c)
                if w is not None:
                    self.grid.SetColSize(c,w)
        self.update_values()
        self.grid.EndBatch()
        self.grid.refresh_scrollbars()
    def get_col_width(self,col):
        return None
```

MODUL: _containers.py

```
import wx
import wx.aui
import images

class Application(wx.PySimpleApp):
```

```
    def __init__(self):
        wx.PySimpleApp.__init__(self)
        self.container = None
        self.panellist = []
    def run(self,container):
        self.container=container
        self.container.Show()
        self.MainLoop()
    def add_panel(self,panel):
        self.panellist.append(panel)
    def enable_last(self,flag):
        panel = self.get_active_panel()
        if panel is not None:
            panel.Enable(flag)
    def get_panel_count(self):
        return len(self.panellist)
    def get_active_panel(self):
        if self.get_panel_count()>0:
            return self.panellist[-1]
        return None

class AuiMDIParentFrame(wx.aui.AuiMDIParentFrame):
    def __init__(self,app,title,*l,**kw):
        wx.aui.AuiMDIParentFrame.__init__(self,None,-1,title,*l,**kw)
        self.app = app
        self.cb = None
        self.create_bars()
    def create_bars(self):
        menu = wx.Menu()
        newwin = menu.Append(-1, "&New Window")
        menu.AppendSeparator()
        exit = menu.Append(-1, "E&xit")
        menubar = wx.MenuBar()
        menubar.Append(menu, "&File")
        self.SetMenuBar(menubar)
        self.CreateStatusBar()
        self.Bind(wx.EVT_MENU, self.OnNewWindow, newwin)
        self.Bind(wx.EVT_MENU, self.OnExit, exit)
    def OnExit(self, evt):
        self.Close(True)
    def OnNewWindow(self, evt):
        if self.cb is not None:
```

```
            self.cb()
    def refresh(self):
        h,w = self.GetSize()
        self.SetSize((h+1,w))
        self.SetSize((h,w))
        self.Refresh()

class Container:
    def __init__(self,app):
        self.Bind(wx.EVT_CLOSE,self.on_close)
        self.app = app
        self.panel = None
    def set_panel(self,panel):
        self.panel = panel
    def on_close(self,event,*l,**kw):
        print "CLOSE CONTAINER"
        ap = self.app.get_active_panel()
        if self.panel == ap :
            if self.panel is not None:
                self.panel.block.close()
                if not self.panel.block.running:
                    event.Skip() # actually closes panel
                    # Housekeeping
                    if ap is not None:
                        self.app.panellist.pop()
                        self.app.enable_last(True)
        else:
            pass

class AuiMDIChildFrame(wx.aui.AuiMDIChildFrame,Container):
    def __init__(self,app,parent=None,title='New frame',**kw):
        wx.aui.AuiMDIChildFrame.__init__(self,parent,-1,title,**kw)
        Container.__init__(self,app)
        self.parent = parent
    def set_panel(self,panel):
        Container.set_panel(self,panel)
        sizer = wx.BoxSizer()
        sizer.Add(panel, 1, wx.EXPAND)
        self.SetSizer(sizer)
        wx.CallAfter(self.Layout)
        self.parent.refresh()
```

```
class Frame(wx.Frame,Container):
    def __init__(self,app,parent=None,title='Frame window',**kw):
        wx.Frame.__init__(self,parent,-1,title,**kw)
        Container.__init__(self,app)
```

MODUL: _interface.py

```
from _containers import Application, Frame, AuiMDIParentFrame, AuiMDIChildFrame
from _panel import Panel, ScrolledPanel
from _sizer import Sizer, HSizer, VSizer
from _widgets import Label, Button, Edit
from _table import Table
from _matrix import Matrix
from _line import Line
import wx

class Interface:
    def application(self,block,module,appfield,*l,**kw):
        appfield.set(Application())
    def apprun(self,block,module,app,framefield,*l,**kw):
        app.get().run(framefield.get())
    def frame(self,block,module,app,framefield,*l,**kw):
        t = block.getvalue(kw.get('title','Frame Window'))
        size = block.getvalue(kw.get('size',(550,250)))
        framefield.set(Frame(app.get(),None,t,size=size))
    def refresh(self,block,module,panel,mode,*l,**kw):
        panel.get().refresh(mode)
        panel.get().container.Show()
    def close(self,block,module,frame,*l,**kw):
        frame.get().Close(True)
    def message(self,block,module,caption,text,*l,**kw):
        wx.MessageBox(text,caption=caption,style=wx.OK)
    def question(self,block,module,caption,text,field,*l,**kw):
        print "QUESTION!!!"
        value = wx.MessageBox(text,caption=caption,style=wx.YES_NO)
        if value is wx.YES:
            field.set(True)
        else:
            field.set(False)
    def autopanel(self,block,module,frame,panelfield,*l,**kw):
        ss = Sizer(5)
        panel = ScrolledPanel(frame.get(),block,5,ss)
```

```
tfields = []
efields = []
bfields = []
for f in block.fieldlist:
    if f.data.get('table',None) is True:
        tfields.append(f)
    elif f.data.get('edit',None) is True:
        efields.append(f)
    elif f.data.get('button',None) is True:
        bfields.append(f)
print tfields
print efields
print bfields
# Table part
if len(tfields)>0:
    table = Table(panel)
    for f in tfields:
        ltitle = block.getvalue(f.data.get('title',f.name))
        table.add_column(f,ltitle)
    table.setminsize(200,100)
    ss.add(table,pos=(0,0),flag=wx.EXPAND)
    ss.grow_col(0)
    ss.grow_row(0)
# Edits part
if len(efields)>0:
    less = Sizer(5)
    i = 0
    for f in efields:
        ltitle = block.getvalue(f.data.get('title',f.name))
        l = Label(panel,'%s'%ltitle)
        e = Edit(panel,size=(150,-1),field=f)
        less.add(l,pos = (0,i))
        less.add(e,pos = (1,i))
        i += 1
    if len(tfields)>0:
        ss.add(less,pos = (1,0))
    else:
        ss.add(less,pos = (0,0))
# Buttons part
if len(bfields)>0:
    bss = Sizer(5)
    i = 0
```

```
            for f in bfields:
                btitle = block.getvalue(f.data.get('title',f.name))
                b = Button(panel,f,'%s'%btitle)
                bss.add(b,pos=(i+1,0))
                i += 1
            bss.grow_col(0)
            if len(tfields)>0 and len(efields)>0:
                ss.add(bss,pos=(0,1),span=(2,1),flag=wx.EXPAND)
            elif len(tfields)>0 or len(efields)>0:
                ss.add(bss,pos=(0,1),flag=wx.EXPAND)
            else:
                ss.add(bss,pos=(0,0),flag=wx.EXPAND)
        panelfield.set(panel)

interface = Interface()
```

MODUL: _panel.py

```
import wx
import wx.lib.scrolledpanel
import _sizer

class _Panel:
    def __init__(self,container,block,border,sizer):
        self.container = container
        self.container.set_panel(self)
        self.container.app.enable_last(False)
        self.container.app.add_panel(self)
        self.container.app.enable_last(True)
        self.block = block
        self.grid = None
        self.border = border
        self.sizer = sizer
        self._define_sizers()
        self.zorderlist=[]
        self.fielddict = {}
        self.Bind(wx.EVT_KEY_DOWN,self.on_key_down)
        self.Bind(wx.EVT_NAVIGATION_KEY,self.on_navigation)
        self.Bind(wx.EVT_IDLE,self.on_idle)
        self.should_process_focus = True
    def on_idle(self,event):
        if self.block.block_before_executed and self.block.running:
```

```
            self.force_widget_focus()
        event.Skip()
    def on_navigation(self,event):
        if event.GetDirection(): # forward direction
            self.forward()
        else:
            self.backward()
    def on_key_down(self,event):
        key = event.GetKeyCode()
        if key == wx.WXK_F4:
            print 'F4 pressed'
            self.insert_line()
            self.refresh(0)
            return
        elif key == wx.WXK_F3:
            print 'F3 pressed'
            self.delete_line()
            self.refresh(0)
            return
        elif key == wx.WXK_F5:
            print 'F5 pressed'
            self.action_line()
            self.refresh(0)
            return
        elif key == wx.WXK_F2:
            print 'F2 pressed'
            self.cancel_line()
            self.refresh(0)
            return
        elif key == wx.WXK_UP and self.grid is not None:
            print "UP pressed"
            newrowref = self.block.get_prev_rowref()
            self.process_focus(newrowref,self.block.fieldref)
            return
        elif key == wx.WXK_DOWN and self.grid is not None:
            print "DOWN pressed"
            newrowref = self.block.get_next_rowref()
            self.process_focus(newrowref,self.block.fieldref)
            return
        event.Skip()
    def delete_line(self):
        self.block.process_delete(self.block.rowref,self.block.fieldref)
```

```
def insert_line(self):
   self.block.process_insert(self.block.rowref,self.block.fieldref)
def cancel_line(self):
   self.block.process_cancel(self.block.rowref,self.block.fieldref)
def action_line(self):
   self.block.process_action(self.block.fieldref)
def add_widget(self,widget):
   self.zorderlist.append(widget)
   if widget.field is not None:
      self.fielddict[widget.field]=widget
def forward(self):
   w = self.get_current_widget()
   if w in self.zorderlist:
      i = self.zorderlist.index(w)+1
      if i>=len(self.zorderlist):
         i = 0
      neww = self.zorderlist[i]
      neww.SetFocus()
      neww.on_forward()
def backward(self):
   w = self.get_current_widget()
   if w in self.zorderlist:
      i = self.zorderlist.index(w)-1
      if i<0:
         i = len(self.zorderlist)-1
      neww = self.zorderlist[i]
      neww.SetFocus()
      neww.on_backward()
def _define_sizers(self):
   self.vbox = _sizer.VSizer()
   self.SetSizer(self.vbox)
   self.vbox.space(self.border)
   self.hbox = _sizer.HSizer()
   self.vbox.add(self.hbox,1,wx.EXPAND|wx.ALL)
   self.hbox.space(self.border)
   self.hbox.add(self.sizer,1,wx.EXPAND|wx.ALL)
   self.hbox.space(self.border)
   self.vbox.space(self.border)
def is_active(self):
   return self.container.app.get_active_panel() == self
def goto(self,widget,*l,**kw):
   widget.goto(*l,**kw)
```

```
def field2widget(self,field):
    if field in self.fielddict.keys():
        return self.fielddict[field]
    if self.grid is not None:
        return self.grid.field2widget(field)
    return None
def get_current_widget(self):
    if self.block.fieldref is None:
        return None
    return self.field2widget(self.block.fieldref)
def refresh(self,param=0):
    # param (0 = all, 1 = grid only, 2 = non grid only)
    print "PANEL REFRESH",param,'AT',self.block.rowref,self.block.fieldref
    if self.block.running:
        self.should_process_focus = False
        if param==1 and self.grid is not None:
            self.grid.refresh()
        else:
            for widget in self.zorderlist:
                if param==0 or param==2 and widget is not self.grid:
                    widget.refresh()
        self.should_process_focus = True
def process_focus(self,newrowref,newfield):
    print "PANEL PROCESS FOCUS",newrowref,newfield
    if self.grid is not None:
        self.grid.data.update_values()
    if self.should_process_focus:
        w = self.get_current_widget()
        if w is not None:
            w.on_exit()
        if self.grid is not None:
            self.grid.data.lastsetvalue = None
        self.block.process_focus(newrowref,newfield)
        self.refresh(0)
def process_delete(self,rowref,field):
    self.block.process_focus(rowref,field)
    self.refresh(0)
def process_insert(self,rowref,field):
    self.block.process_focus(rowref,field)
    self.refresh(0)
def process_action(self,field):
    self.block.process_action(field)
```

```
        self.refresh(0)
    def force_widget_focus(self):
        brownum,bfieldref = self.block.get_pos()
        widget = self.get_current_widget()
        # if panel has grid
        if self.grid is not None:
            col,rownum = self.grid.get_pos()
            if rownum<>brownum:
                if widget==self.grid:
                    newcol = self.grid.field2col(bfieldref)
                    self.grid.goto(newcol,brownum)
                else:
                    self.grid.goto(col,brownum)
                    widget.goto()
            else:
                if widget==self.grid:
                    newcol = self.grid.field2col(bfieldref)
                    if col<>newcol:
                        self.grid.goto(newcol,brownum)
                else:
                    widget.goto()
        else: # has not grid
            widget.goto()

class ScrolledPanel(wx.lib.scrolledpanel.ScrolledPanel,_Panel):
    def __init__(self,container,block,border,sizer):
        wx.lib.scrolledpanel.ScrolledPanel.__init__(self,container,-
1,style=wx.TAB_TRAVERSAL)
        self.SetupScrolling(True,True)
        _Panel.__init__(self,container,block,border,sizer)

class Panel(wx.Panel,_Panel):
    def __init__(self,container,block,border,sizer):
        wx.Panel.__init__(self,container,-1,style=wx.TAB_TRAVERSAL)
        _Panel.__init__(self,container,block,border,sizer)
```

MODUL: _sizer.py

```
import wx

class Sizer(wx.GridBagSizer):
    def __init__(self,border):
```

```
        wx.GridBagSizer.__init__(self,hgap=border,vgap=border)
    def add(self,object,**kw):
        # switch from y,x to x,y
        if kw.has_key('pos'):
            x,y = kw['pos']
            kw['pos']=(y,x)
        if kw.has_key('span'):
            w,h = kw['span']
            kw['span']=(h,w)
        self.Add(object,**kw)
    def grow_col(self,col):
        self.AddGrowableCol(col)
    def grow_row(self,row):
        self.AddGrowableRow(row)

class _BoxSizer(wx.BoxSizer):
    def __init__(self,orientation):
        wx.BoxSizer.__init__(self,orientation)
    def add(self,object,*l,**kw):
        self.Add(object,*l,**kw)
    def space(self,value):
        self.AddSpacer(value)

class VSizer(_BoxSizer):
    def __init__(self):
        _BoxSizer.__init__(self,wx.VERTICAL)

class HSizer(_BoxSizer):
    def __init__(self):
        _BoxSizer.__init__(self,wx.HORIZONTAL)
```

MODUL: _table.py

```
import wx
import _common
import common

class Table(_common.Grid):
    def __init__(self,panel):
        _common.Grid.__init__(self,panel)
        self.SetRowLabelSize(1)
        self.SetDefaultRowSize(20,True)
```

```
    self.columnlist = []
    self.columndict = {}
    self.titlelist = []
    w = 20
    h = 100
    self.set_data(_TableData(self))
def on_forward(self):
    col,row = self.get_pos()
    col = 0
    self.SetGridCursor(row,col)
    self.MakeCellVisible(row,col)
def on_backward(self):
    col,row = self.get_pos()
    col = self.GetNumberCols()-1
    self.SetGridCursor(row,col)
    self.MakeCellVisible(row,col)
def add_column(self,field,title):
    self.columndict[field]=len(self.columnlist)
    self.columnlist.append(field)
    self.titlelist.append(title)
def refresh(self):
    self.data.set_grid_size(len(self.columnlist),\
                        len(self.panel.block.buffer.changedbuffer))
    newrownum,newfield = self.panel.block.get_pos()
    if newfield in self.columndict.keys():
        modifiable = self.panel.block.getvalue(newfield.modifiable)
        col = self.columndict[newfield]
        row = newrownum
        self.SetReadOnly(row,col,not modifiable)
        self.goto(col,row)
    else:
        col,row = self.get_pos()
        self.goto(col,newrownum)
def pos2focus(self,pos):
    col,row = pos
    rowref =  self.panel.block.rownum2rowref(row)
    fieldref = self.columnlist[col]
    if rowref == common.ROWREF_NO_RECORD:
        return None
    return rowref,fieldref
def field2widget(self,field):
    if field in self.columndict.keys():
```

```
            return self
        return None
    def field2col(self,field):
        return self.columndict[field]

class _TableData(_common._GridData):
    def __init__(self,grid):
        _common._GridData.__init__(self,grid)
        self.lastsetvalue = None
    def GetColLabelValue(self,col):
        return '%s'%self.grid.titlelist[col] # title
    def GetValue(self, row, col):
        field = self.grid.columnlist[col]
        title = self.grid.titlelist[col]
        rowref = self.grid.panel.block.buffer.rowreforder[row]
        return field.get(rowref=rowref,method=common.CONV_BUF2STR)
    def SetValue(self, row, col, val):
        print "SET VALUE",(row,col,val)
        if self.lastsetvalue <> (row,col,val):
            self.lastsetvalue = (row,col,val)
            field = self.grid.columnlist[col]
            field.set(val,method=common.CONV_STR2BUF,place='GUI')
    def get_col_width(self,col):
        field = self.grid.columnlist[col]
        width = self.grid.panel.block.getvalue(field.data.get('width',None))
        return width
```

MODUL: _widgets.py

```
import wx
import _common
import common

class Label(wx.StaticText,_common.Widget):
    def __init__(self,panel,text,**kw):
        wx.StaticText.__init__(self,panel,-1,text,**kw)
        _common.Widget.__init__(self,panel)

class Button(wx.Button,_common.Widget):
    def __init__(self,panel,field,text,**kw):
        wx.Button.__init__(self,panel,-1,text,**kw)
        _common.Widget.__init__(self,panel)
```

```
        self.field = field
        self.panel.add_widget(self)
        self.SetName('button %s'%text)
        self.cb = kw.get('cb',None)
    def init_events(self):
        _common.Widget.init_events(self)
        self.Bind(wx.EVT_BUTTON,self.on_click)
    def on_click(self,*l,**kw):
        if self.cb is not None:
            self.cb()
        self.panel.process_action(self.field)
    def refresh(self):
        newrownum,newfield = self.panel.block.get_pos()
        if self.field == newfield:
            self.goto()

class Edit(wx.TextCtrl,_common.Widget):
    def __init__(self,panel,field=None,**kw):
        wx.TextCtrl.__init__(self,panel,-1,**kw)
        _common.Widget.__init__(self,panel)
        self.field = field
        self.panel.add_widget(self)
    def refresh(self):
        if self.field is not None:
            self.ChangeValue(self.field.get(method=common.CONV_BUF2STR))
        else:
            self.ChangeValue('N/A')
        newrownum,newfield = self.panel.block.get_pos()
        if self.field == newfield:
            modifiable = self.panel.block.getvalue(self.field.modifiable)
            self.SetEditable(modifiable)
            self.goto()
    def on_exit(self):
        if self.IsModified():
            self.field.set(self.GetValue(),method=common.CONV_STR2BUF,place='GUI')
```

PRILOG 3: IZVORNI KOD PROGRAMSKOG PRIMJERA

MODUL: kreiranje_primjera.py

```
import sqlite3

artikl = "CREATE TABLE ARTIKL (SIFRA PRIMARY KEY, NAZIV, JM, CIJENA)"
dokument = "CREATE TABLE DOKUMENT (ID PRIMARY KEY, DATUM, NAPOMENA)"
stavka = "CREATE TABLE STAVKA (ID PRIMARY KEY, ID_DOKUMENTA, SIFRA_ARTIKLA, ULAZ DEFAULT 0, IZLAZ DEFAULT 0)"
skladiste = "CREATE VIEW SKLADISTE AS SELECT SIFRA_ARTIKLA, SUM(ULAZ-IZLAZ) AS STANJE FROM STAVKA GROUP BY SIFRA_ARTIKLA"

conn = sqlite3.connect('primjer.db')
c = conn.cursor()

c.execute(artikl)
c.execute(dokument)
c.execute(stavka)
c.execute(skladiste)

c.close()
conn.close()
```

MODUL: primjer.py

```
import sys
sys.path = ['..','../base']+sys.path
sys.path = ['..','../classic_gui']+sys.path

print '-'*60

import base
import db_sqlite3 as db
import common

# connection
conn = db.Connection('primjer.db')
```

```
class App(base.Application):
   def __init__(self):
      base.Application.__init__(self)
      self.conn = db.Connection('primjer.db')
   def load(self,blockname):
      if blockname=='dokument':
         return self.dokument()
      elif blockname=='stavka':
         return self.stavka()
      elif blockname=='artikl':
         return self.artikl()
      return None

   ##################
   # BLOK DOKUMENT #
   ##################
   def dokument(self):
      b = base.Block(self,'dokument')
      cgui = base.GlobalField(b,'cgui',base.generic,'g_cgui')
      capp = base.GlobalField(b,'capp',base.generic,'g_capp')
      lib = base.GlobalField(b,'lib',base.generic,'g_lib')
      cframe = base.BlockField(b,'cframe',base.generic)
      cpanel = base.BlockField(b,'cpanel',base.generic)
      # TABLE DEFINITIONS
      d = base.MainTable(b,conn,'d','dokument')
      d_id = base.DbField(b,'d_id',base.number,d,'id',first=True,select=True, \
               rowref=True,insert=True,orderby=1,descending=True,\
               modifiable=False,\
               data={'table':True,'title':'ID dokumenta','width':100})
      d_datum                                                                    =
base.DbField(b,'d_datum',base.datetime,d,'datum',select=True,insert=True,update=True,\
               data={'table':True,'title':'Datum','width':60})
      d_napomena                                                                 =
base.DbField(b,'d_napomena',base.string,d,'napomena',select=True,insert=True,update=True,\
               data={'table':True,'title':'Napomena','width':200})
      l = base.LinkTable(b,conn,'l','dokument')
      l_id = base.DbField(b,'l_id',base.number,l,'id',select=True, \
               modifiable=False, aggfnc='MAX')
      # SQL STATEMENTS
      stm_d = base.SQLSelect(b,self.conn,[d],overwrite=False)
      stm_d_refetch = base.SQLSelectRefetch(b,self.conn,[d])
      stm_d_update = base.SQLUpdateRow(b,self.conn,[d])
```

```
    stm_d_delete = base.SQLDeleteRow(b,self.conn,[d],deleterow=True)
    stm_d_insert = base.SQLInsertRow(b,self.conn,[d])
    stm_l = base.SQLSelect(b,conn,[l],overwrite=True,fetchnum=1,reexecuting=True)
    # EVENT BLOCK DESCRIPTOR
    edesc = base.EventBlockDescriptor(b)
    base.DumpInstr(edesc.method,0,'Table l desc',stm_l)
    base.UpdateInstr(edesc.method,0,lib,base.EvalExpr(b,'__import__("base").lib'))
    base.UpdateInstr(edesc.method,0,cgui,\
       base.EvalExpr(b,'__import__("classic_gui").interface'))
    base.CallExtInstr(edesc.method,0,cgui,'application',arglist=[capp])
    base.CallExtInstr(edesc.method,0,cgui,'frame',arglist=[capp,cframe],\
                              argdict={'title':base.EvalExpr(b,'"Primjer - lista dokumenata"'),\
                                   'size':base.EvalExpr(b,'(400,500)')})
    base.CallExtInstr(edesc.method,0,cgui,'autopanel',arglist=[cframe,cpanel])
    base.DumpInstr(edesc.method,0,'Block descriptor method',cframe)
    base.CallExtInstr(edesc.method,0,cgui,'refresh',arglist=[cpanel,0])
    base.CallExtInstr(edesc.method,0,cgui,'apprun',arglist=[capp,cframe])
    # EVENT BLOCK FETCH
    ebfetch = base.EventBlockFetch(b)
    base.ExecSQLStatementInstr(ebfetch.method,0,stm_d)
    # EVENT RECORD AFTER
    erafter = base.EventRecordAfter(b)
    base.EvaluateInstr(erafter.method,0,base.EvalExpr(b,'func_write()'))
    # METHOD func_write
    func_write = base.Method(b,'func_write',should_eval=True)

base.ExecSQLStatementInstr(func_write,0,stm_d_delete,condition=base.EvalExpr(b,'self.sho
uld_delete()'))

base.ExecSQLStatementInstr(func_write,0,stm_d_update,condition=base.EvalExpr(b,'self.sho
uld_update("d")'))
    base.IfInstr(func_write,0,condition=base.EvalExpr(b,'self.should_insert()'))
    base.ExecSQLStatementInstr(func_write,1,stm_l)
    base.UpdateInstr(func_write,1,d_id,base.EvalExpr(b,'lib.nvl(l_id,0)+1'))
    base.ExecSQLStatementInstr(func_write,1,stm_d_insert)
    base.CommitInstr(func_write,0,self.conn)
    base.ReturnInstr(func_write,0,True)
    # EVENT ACTION B_STAVKE
    b_stavke = base.BlockField(b,'b_stavke',base.number,data={'button':True,'title':'Stavke'})
    efaction_b_stavke = base.EventFieldAction(b,b_stavke)
    base.DumpInstr(efaction_b_stavke.method,0,'#### Poziv stavki ####',\
       base.EvalExpr(b,"self.call('stavka',d_id)"),condition=base.EvalExpr(b,'func_write()'))
    # EVENT ACTION B_ARTIKLI
```

```
        b_artikli = base.BlockField(b,'b_artikli',base.number,data={'button':True,'title':'Artikli'})
        efaction_b_artikli = base.EventFieldAction(b,b_artikli)
        base.DumpInstr(efaction_b_artikli.method,0,'#### Poziv artikala ####',base.EvalExpr(b,"self.call('artikl')"))
        # EVENT ACTION b_izlaz
        b_izlaz = base.BlockField(b,'b_izlaz',base.number,data={'button':True,'title':'Izlaz'})
        efaction_b_izlaz = base.EventFieldAction(b,b_izlaz)
        base.CallExtInstr(efaction_b_izlaz.method,0,cgui,'close',arglist=[cframe])
        # FUNCTION RETURN
        b.rebuild()
        b.debug = True
        print 'FF',b.firstfield
        return b

    ###############
    # BLOK STAVKA #
    ###############
    def stavka(self):
        b = base.Block(self,'stavka')
        cgui = base.GlobalField(b,'cgui',base.generic,'g_cgui')
        capp = base.GlobalField(b,'capp',base.generic,'g_capp')
        lib = base.GlobalField(b,'lib',base.generic,'g_lib')
        cframe = base.BlockField(b,'cframe',base.generic)
        cpanel = base.BlockField(b,'cpanel',base.generic)
        p_dok_id = base.InputParameter(b,'p_dok_id',base.number)
        # EVENT BLOCK DESCRIPTOR
        edesc = base.EventBlockDescriptor(b)
        base.CallExtInstr(edesc.method,0,cgui,'frame',arglist=[capp,cframe],\
                          argdict={'title':base.EvalExpr(b,'"Stavke dokumenta %d"%p_dok_id'),\
                                   'size':base.EvalExpr(b,'(650,400)')})
        base.CallExtInstr(edesc.method,0,cgui,'autopanel',arglist=[cframe,cpanel])
        base.DumpInstr(edesc.method,0,'Block descriptor method',cframe)
        base.CallExtInstr(edesc.method,0,cgui,'refresh',arglist=[cpanel,0])
        # TABLE DEFINITIONS
        s = base.MainTable(b,conn,'s','stavka')
        a = base.LinkTable(b,conn,'a','artikl')
        l = base.LinkTable(b,conn,'l','stavka')
        sk = base.LinkTable(b,conn,'sk','skladiste')
        # FIELLD DEFINITIONS
        s_id = base.DbField(b,'s_id',base.number,s,'id',first=True,select=True, \
                rowref=True,insert=True,orderby=1,\
```

```
            modifiable=False,\
            data={'table':True,'title':'ID stavke','width':70})
    s_id_dokumenta                                                    =
base.DbField(b,'s_id_dokumenta',base.number,s,'id_dokumenta',select=True,insert=True,upda
te=True,\
            modifiable = False,\
            data={'table':True,'title':'ID dokumenta','width':100})
    s_sifra_artikla                                                   =
base.DbField(b,'s_sifra_artikla',base.number,s,'sifra_artikla',select=True,insert=True,update=
True,\
            data={'table':True,'title':'Sifra artikla','width':100})
    s_ulaz                                                            =
base.DbField(b,'s_ulaz',base.number,s,'ulaz',select=True,insert=True,update=True,\
            data={'table':True,'title':'Ulaz','width':60})
    s_izlaz                                                           =
base.DbField(b,'s_izlaz',base.number,s,'izlaz',select=True,insert=True,update=True,\
            data={'table':True,'title':'Izlaz','width':60})
    l_id = base.DbField(b,'l_id',base.number,l,'id',select=True, \
            modifiable=False, aggfnc='MAX')
    a_sifra = base.DbField(b,'a_sifra',base.number,a,'sifra',select=True, \
            orderby=1,modifiable=False)
    a_naziv = base.DbField(b,'a_naziv',base.string,a,'naziv',select=True,modifiable=False,\
            data={'edit':True,'title':'Naziv artikla','width':150})
    a_jm = base.DbField(b,'a_jm',base.string,a,'jm',select=True,modifiable=False,\
            data={'edit':True,'title':'JM','width':60})
    a_cijena                                                          =
base.DbField(b,'a_cijena',base.number,a,'cijena',select=True,modifiable=False,\
            data={'edit':True,'title':'Cijena','width':60})
    sk_sifra_artikla                                                  =
base.DbField(b,'sk_sifra_artikla',base.number,sk,'sifra_artikla',select=True, \
            modifiable=False)
    sk_stanje                                                         =
base.DbField(b,'sk_stanje',base.number,sk,'stanje',select=True,modifiable=False,\
            data={'edit':True,'title':'Stanje','width':60})
    # CONDITIONS
    c1 = base.OpCond(b,s_id_dokumenta,base.op_eq,base.EvalExpr(b,'p_dok_id'))
    c2 = base.OpCond(b,a_sifra,base.op_eq,base.EvalExpr(b,'s_sifra_artikla'))
    c3 = base.OpCond(b,sk_sifra_artikla,base.op_eq,base.EvalExpr(b,'s_sifra_artikla'))
    # RECALCS
    r1 = base.Recalc(b,s_id_dokumenta,base.EvalExpr(b,'p_dok_id'),None)
    # SQL STATEMENTS
    stm_s = base.SQLSelect(b,self.conn,[s],overwrite=False)
    stm_s_refetch = base.SQLSelectRefetch(b,self.conn,[s])
```

```
    stm_s_update = base.SQLUpdateRow(b,self.conn,[s])
    stm_s_delete = base.SQLDeleteRow(b,self.conn,[s],deleterow=True)
    stm_s_insert = base.SQLInsertRow(b,self.conn,[s])
    stm_l = base.SQLSelect(b,conn,[l],overwrite=True,fetchnum=1,reexecuting=True)
    stm_a = base.SQLSelect(b,conn,[a],overwrite=True,fetchnum=1,reexecuting=True)
    stm_sk = base.SQLSelect(b,conn,[sk],overwrite=True,fetchnum=1,reexecuting=True)
    # METHOD link_refresh
    link_refresh = base.Method(b,'link_refresh')
    base.DumpInstr(link_refresh,0,'LINK REFRESH:',s_sifra_artikla)
    base.ExecSQLStatementInstr(link_refresh,0,stm_a)
    base.ExecSQLStatementInstr(link_refresh,0,stm_sk)
    # EVENT BLOCK FETCH
    ebfetch = base.EventBlockFetch(b)
    base.DumpInstr(ebfetch.method,0,'Parameter P_DOK_ID:',p_dok_id)
    base.ExecSQLStatementInstr(ebfetch.method,0,stm_s)
    # EVENT RECORD FETCH
    erfetch = base.EventRecordFetch(b)
    base.EvaluateInstr(erfetch.method,0,base.EvalExpr(b,'link_refresh()'))
    # EVENT FIELD AFTER
    efafter_ssa = base.EventFieldAfter(b,s_sifra_artikla)
    base.EvaluateInstr(efafter_ssa.method,0,base.EvalExpr(b,'link_refresh()'))
    base.IfInstr(efafter_ssa.method,0,condition=base.EvalExpr(b,'s_sifra_artikla is not None
and s_sifra_artikla<>a_sifra'))
    base.CallExtInstr(efafter_ssa.method,1,cgui,'message',arglist=['Obavijest','Ne postoji taj
artikl!'])
    base.ReturnInstr(efafter_ssa.method,1,False)
    # EVENT RECORD BEFORE
    erbefore = base.EventRecordBefore(b)
    base.EvaluateInstr(erbefore.method,0,base.EvalExpr(b,'link_refresh()'))
    # EVENT RECORD AFTER
    erafter = base.EventRecordAfter(b)

base.UpdateInstr(erafter.method,0,s_ulaz,base.EvalExpr(b,'0'),condition=base.EvalExpr(b,'s_
ulaz is None'))

base.UpdateInstr(erafter.method,0,s_izlaz,base.EvalExpr(b,'0'),condition=base.EvalExpr(b,'s_
izlaz is None'))

base.ExecSQLStatementInstr(erafter.method,0,stm_s_delete,condition=base.EvalExpr(b,'self.
should_delete()'))

base.ExecSQLStatementInstr(erafter.method,0,stm_s_update,condition=base.EvalExpr(b,'self.
should_update("s")'))
```

```
        base.IfInstr(erafter.method,0,condition=base.EvalExpr(b,'self.should_insert()'))
        base.ExecSQLStatementInstr(erafter.method,1,stm_l)
        base.UpdateInstr(erafter.method,1,s_id,base.EvalExpr(b,'lib.nvl(l_id,0)+1'))
        base.ExecSQLStatementInstr(erafter.method,1,stm_s_insert)
        base.CommitInstr(erafter.method,0,self.conn)
        # EVENT ACTION B_ARTIKLI
        b_artikli = base.BlockField(b,'b_artikli',base.number,data={'button':True,'title':'Artikli'})
        efaction_b_artikli = base.EventFieldAction(b,b_artikli)
        base.DumpInstr(efaction_b_artikli.method,0,'#### Poziv artikala ####',base.EvalExpr(b,"self.call('artikl')"))
        base.EvaluateInstr(efaction_b_artikli.method,0,base.EvalExpr(b,'link_refresh()'))
        # EVENT ACTION b_izlaz
        b_izlaz = base.BlockField(b,'b_izlaz',base.number,data={'button':True,'title':'Izlaz'})
        efaction_b_izlaz = base.EventFieldAction(b,b_izlaz)
        base.CallExtInstr(efaction_b_izlaz.method,0,cgui,'close',arglist=[cframe])
        # FUNCTION RETURN
        b.rebuild()
        b.debug = True
        return b

    ###############
    # BLOK ARTIKL #
    ###############
    def artikl(self):
        b = base.Block(self,'artikl')
        cgui = base.GlobalField(b,'cgui',base.generic,'g_cgui')
        capp = base.GlobalField(b,'capp',base.generic,'g_capp')
        lib = base.GlobalField(b,'lib',base.generic,'g_lib')
        cframe = base.BlockField(b,'cframe',base.generic)
        cpanel = base.BlockField(b,'cpanel',base.generic)
        # EVENT BLOCK DESCRIPTOR
        edesc = base.EventBlockDescriptor(b)
        base.CallExtInstr(edesc.method,0,cgui,'frame',arglist=[capp,cframe],\
                                argdict={'title':base.EvalExpr(b,'"Artikli"'),\
                                    'size':base.EvalExpr(b,'(400,400)')})
        base.CallExtInstr(edesc.method,0,cgui,'autopanel',arglist=[cframe,cpanel])
        base.DumpInstr(edesc.method,0,'Block descriptor method',cframe)
        base.CallExtInstr(edesc.method,0,cgui,'refresh',arglist=[cpanel,0])
        # TABLE DEFINITIONS
        a = base.MainTable(b,conn,'a','artikl')
        a_sifra = base.DbField(b,'a_sifra',base.number,a,'sifra',first=True,select=True, \
                rowref=True,insert=True,orderby=1,\
```

```
            modifiable=base.EvalExpr(b,'lib.isnewrecord(self)'),\
            data={'table':True,'title':'Sifra artikla','width':100})
    a_naziv =
base.DbField(b,'a_naziv',base.string,a,'naziv',select=True,insert=True,update=True,\
            data={'table':True,'title':'Naziv','width':150})
    a_jm = base.DbField(b,'a_jm',base.string,a,'jm',select=True,insert=True,update=True,\
            data={'table':True,'title':'JM','width':60})
    a_cijena =
base.DbField(b,'a_cijena',base.number,a,'cijena',select=True,insert=True,update=True,\
            data={'table':True,'title':'Cijena','width':60})
    # SQL STATEMENTS
    stm_a = base.SQLSelect(b,self.conn,[a],overwrite=False)
    stm_a_refetch = base.SQLSelectRefetch(b,self.conn,[a])
    stm_a_update = base.SQLUpdateRow(b,self.conn,[a])
    stm_a_delete = base.SQLDeleteRow(b,self.conn,[a],deleterow=True)
    stm_a_insert = base.SQLInsertRow(b,self.conn,[a])
    # EVENT BLOCK FETCH
    ebfetch = base.EventBlockFetch(b)
    base.ExecSQLStatementInstr(ebfetch.method,0,stm_a)
    # EVENT RECORD AFTER
    erafter = base.EventRecordAfter(b)

base.ExecSQLStatementInstr(erafter.method,0,stm_a_delete,condition=base.EvalExpr(b,'self.
should_delete()'))

base.ExecSQLStatementInstr(erafter.method,0,stm_a_update,condition=base.EvalExpr(b,'self
.should_update("a")'))

base.ExecSQLStatementInstr(erafter.method,0,stm_a_insert,condition=base.EvalExpr(b,'self.s
hould_insert()'))
    base.CommitInstr(erafter.method,0,self.conn)
    # EVENT ACTION b_izlaz
    b_izlaz = base.BlockField(b,'b_izlaz',base.number,data={'button':True,'title':'Izlaz'})
    efaction_b_izlaz = base.EventFieldAction(b,b_izlaz)
    base.CallExtInstr(efaction_b_izlaz.method,0,cgui,'close',arglist=[cframe])
    # FUNCTION RETURN
    b.rebuild()
    b.debug = True
    return b

app = App()
app.call('dokument')
```

PRILOG 4: RJEČNIK POJMOVA

ALGORITAM

Algoritam jest precizan opis konačne liste postupaka za izračun neke funkcije. Polazeći od nekog početnog stanja, izvršavanje opisanih postupaka u konačnom broju koraka mora dovesti do željenog završnog stanja. Algoritmi se mogu opisivati tekstom, grafičkim prikazom ili nekim od programskih jezika. Algoritmi moraju zadovoljiti slijedeće kriterije (Pandey, 2008, 1-5):

a) Početno stanje - opis mora imati početno stanje od kojeg postupci kreću. Početno stanje može ali i ne mora sadržavati parametre.

b) Završno stanje - opis mora sadržavati željeno završno stanje

c) Jednoznačnost - postupci moraju biti definirani jasno i jednoznačno

d) Konačnost - algoritam mora imati konačan broj postupaka

e) Jednostavnost - postupci moraju biti opisani na jednostavan način da se mogu izvršiti i ručno

APLIKACIJA

Aplikacija je računalni program napravljen da pomaže korisnicima u obavljanju specifičnih poslova na računalu. Područja primjene aplikacija mogu biti brojne, poput računovodstva, uredskih poslova, medicine, obrazovanja i brojnih drugih (Fowler, 2003, 4-7).

ATRIBUT

Atribut je element kojim je jednoznačno određena vrsta svojstva. Relacijski model razlikuje jednostavne i sastavljenje artibute. Vrijednost jednostavnog atributa je pojedinačni podatak,

dok je vrijednost sastavljenog atributa uređena n-torka pojedinačnih podataka (Tkalac, 1993, 19).

BAZA PODATAKA

Baza podataka je sustav za obradu organiziranog skupa podataka, tipično u računalnoj formi. Nad skupom podataka mogu biti izvršene slijedeće osnovne operacije (Date, 1995, 2):

a) Čitanje podataka

b) Unos novih podataka

c) Promjena postojećih podataka

d) Brisanje postojećih podataka

BAZNO POLJE

Bazno polje u relacijskoj bazi podataka je skup podataka istog značenja, po jednog u svakom slogu relacijske tablice. Umjesto naziva polje koristi se i naziv kolona, a u relacijskom modelu odgovara pojmu atributa (Date, 1995, 79).

COMMIT

Operacija COMMIT mehanizmu za upravljanje transakcija signalizira uspješan završetak transakcije, što znači da će sve promjene podataka biti trajno zapisane u bazu podataka (Date, 1995, 376).

DIMENZIJA

U skladištenju podataka, pojam dimenzija označava tablicu koja u sebi sadrži kategorije, poput organizacijskih jedinica, vrsta osiguranja i slično. Dimenzije se vezuju na tablice činjenica ili druge dimenzijske tablice (Ponniah, 2010, 130).

DOGAĐAJ

Događaj je akcija koja je započeta obično izvan aplikacije i koja izaziva reakciju unutar aplikacije. Događaj može biti pritisak neke tipke na tipkovnici, ali i neki složeni uvjet koji zahtijeva reakciju korisnika, npr. potvrdu prilikom izlaska iz aplikacije. Sustav koji obrađuje događaje i reakcije na njih naziva se događajni sustav (Smart, Hock, Csomor, 2006, 25).

DOMENA

U relacijskom modelu domena je skup svih vrijednosti koje određeni atribut može poprimiti. Atribut mora imati domenu i može imati samo jednu domenu, a više različitih atributa može biti zadano na istoj domeni (Tkalac, 1993, 19).

ENTITET

U realnom svijetu entitet je element koji možemo jednoznačno odrediti i na taj način ga izdvojiti odnosno prepoznati u svijetu. Entiteti imaju svojstva, a svojstvo se sastoji od atributa i vrijednosti atributa (Tkalac, 1993, 9).

FOKUS

Koncept fokusa vezan je uz dizajniranje korisničkog sučelja. Korisničko sučelje sastoji se od niza elemenata koji reagiraju na korisničke akcije.

Najvažnije akcije su unos podataka s tipkovnice i kontrola mišem. Proces unosa podataka s tipkovnice zahtijeva jednoznačno označavanje elementa na koji se unos odnosi. Jednoznačna oznaka elementa korisničkog sučelja koji preuzima unos podataka s tipkovnice zove se fokus (Olsen, 1998, 112).

FORMA

Forma predstavlja osnovni element korisničkog sučelja koji služi za prikaz i prikupljanje podataka jedne funkcionalne cjeline, te njihovo zapisivanje na poslužitelju. Forme mogu biti internetske stranice, klasični prozori u grafičkom korisničkom sučelju ili tekstualnom sučelju. Forme sadrže podelemente za unos polja ili kontrolu akcija (Hentzen, 2007, 317-320).

INDEKS

Indeks je objekt baze podataka koji služi ubrzanju pronalaženja traženih slogova u nekoj tablici zapisivanjem često korištenih kolona iz tablice u posebne strukture podataka. Povećana brzina pronalaženja ima za posljedicu usporen proces zapisivanja podataka jer je osim tablice potrebno ažurirati i indeks. Jedna tablica u bazi podataka može imati jedan ili više indeksa (O'Neil, O'Neil, 2000, 466).

INFORMACIJSKI SUSTAV

Informacijski sustav je kombinacija tehnologije i ljudskih aktivnosti koje služe za potporu redovnim aktivnostima i potporu procesima odlučivanja.

Informacijski sustav sastoji se od slijedećih elemenata (O'Brien, Marakas, 2010, 31):

a) IT stručnjaka i krajnjih korisnika

b) Hardvera

c) Softvera

d) Podataka

e) Mrežne infrastrukture

KLIJENT

Klijent je aplikacija ili sustav koji pristupa nekom servisu na poslužitelju. Poslužitelj se može ali i ne mora nalaziti na odvojenom računalu. U slučaju da se poslužitelj nalazi na drugom računalu, klijentski sustav mu pristupa preko mrežne infrastrukture (Rob, Coronel, 2009, 163).

KLJUČ

Ključ relacije zadane na relacijskoj shemi je podskup atributa koji zadovoljava slijedeće uvjete (Tkalac, 1993, 23-27):

a) Jednoznačnost - ne postoje dvije n-torke sa jednakim vrijednostima svih atributa u ključu

b) Minimalnost - niti jedan pravi podskup atributa ključa nema svojstvo jednoznačnosti

Postoji više vrsta ključeva:

a) Mogući ključ - bilo koji ključ u tablici

b) Primarni ključ - izabrani mogući ključ koji služi kao glavni ključ

c) Alternativni ključ - mogući ključ koji nije izabran za primarni ključ

d) Složeni ključ - ključ koji se sastoji od dva ili više atributa

e) Strani ključ - skup atributa u tablici koji zajedno pokazuju na ključ (obično primarni) u nekoj drugoj tablici

KORISNIČKI DOŽIVLJAJ

Korisnički doživljaj (eng. user experience) složeni je subjektivni dojam koji korisnik ima služeći se softverom. Korisnički doživljaj važan je aspekt komunikacije između čovjeka i stroja, te utječe na krajnju ocjenu korisnosti, lakoće uporabe i efikasnost softvera (Wilson, 2010, 3-13).

KORISNIČKO SUČELJE

Korisničko sučelje (eng. user interface) je mehanizam za interakciju između korisnika i softvera. Interakcija se odvija u interaktivnom ciklusu (Olsen, 1998, 11-13) čije su glavne faze:

a) Korisničko sučelje prikazuje podatke

b) Korisnik tumači podatke koje prikazuje korisničko sučelje

c) Korisnik formulira ciljeve i generira ulaz podataka

d) Softver tumači korisničku reakciju i ažurira prikaz podataka na korisničkom sučelju

Postoji više vrsta korisničkih sučelja: komandna linija, grafičko korisničko sučelje (eng. GUI), web sučelje, mobilno sučelje itd.

Dizajniranje korisničkog sučelja uključuje poznavanje ergonomije i psihologije, a posebna pažnja se poklanja korisničkom doživljaju.

NUL VRIJEDNOST

Relacijski model pruža mogućnost da vrijednost atributa bude nepoznata. Oznaka takve vrijednosti naziva se nul vrijednost (eng. null). Postoji problematika korištenja nul vrijednosti kao operanda neke operacije, posebno u slučaju da drugi operand nije nul. U tom slučaju rezultat operacije je također nul vrijednost (Tkalac, 1993, 51).

OBJEKTNO-ORJENTIRANO PROGRAMIRANJE

Objektno-orjentirano programiranje jest programska paradigma koja se temelji na korištenju objekata i klasa. Klase su predlošci objekata, a objekti njihove konačne instance. Objekti imaju metode i atribute. Većina modernih programskih jezika omogućuje objektno-orjentirano programiranje. Objektno-orjentirano programiranje uključuje slijedeće koncepte (Svenk, 2003, 106-109):

a) Enkapsulacija - koncept integracije podataka i funkcije objekta u jednu cjelinu, skrivajući implementacijske detalje

b) Polimorfizam - koncept korištenja iste funkcije za različite tipove podataka

c) Nasljeđivanje - nove klase mogu biti kreirane korištenjem funkcionalnosti roditeljskih klasa

POGLED

Pogled (eng. view) je objekt relacijske baze podataka koji se sastoji od pridruženog upita i djeluje kao virtualna tablica. Od tablice se razlikuje po tome što ne pohranjuje podatke već dohvaća podatke putem ugrađenog upita (Date, 1995, 62-65).

POSLUŽITELJ

Poslužitelj je sustav koji pruža jedan ili više servisa radi zadovoljavanja potreba korisnika na drugim računalima mrežne infrastrukture.
Poslužitelji mogu biti datotečni, internet poslužitelji, baze podataka, poslužitelji za ispis, poslužitelji elektronske pošte i slično. U klijentsko-poslužiteljskoj arhitekturi poslužitelj je aplikacija koja služi za obradu zahtjeva drugih aplikacija preko mrežne infrastrukture (Rob, Coronel, 2009, 163).

PROGRAMIRANJE

Programiranje je proces dizajniranja, pisanja, testiranja i održavanja programskog koda računalnog programa. Programski kod ili izvorni kod može biti pisan u nekom od programskih jezika poštujući njegovu formalnu strukturu. Ciljni računalni program mora zadovoljavati treženu funkcionalnost.
Proces programiranja zahtijeva stručnost u nekoliko disciplina: prikupljanju korisničkih zahtjeva, modeliranju podataka, formalnoj logici, specijaliziranim algoritmima, kao i domensko znanje u području traženog funkcionala (Farrell, 2012, 3-13).

PROGRAMSKI JEZIK

Programski jezik je umjetni formalni jezik namijenjen za programiranje, to jest pisanje računalnih programa koji izvršavaju zadane algoritme izvođenjem zadanih instrukcija. Programski jezici mogu biti tumači (eng. interpreter) ili prevoditelji (eng. compiler). Tumači prevode i izvode svaku instrukciju izvornog koda zasebno, dok prevoditelji prvo pretvore izvorni kod u izvršni kod strojnog jezika, koji se zatim može neovisno izvršavati (Farrell, 2012, 3-13).

RAČUNALNA PLATFORMA

Računalna platforma je spoj hardvera i softvera na kojem se izvršavaju korisnički programi (Shroff, 2010, 1-12). Primjeri računalnih platformi su mainframe računala, klijent-server arhitektura, mobilne platforme, web platforme, računalni oblaci itd.

RELACIJA

Relacija na relacijskoj shemi je konačan skup n-torki. Relacija može biti prikazana u obliku dvodimenzionalne tabele gdje sve n-torke relacije moraju biti jednake duljine a stupci odgovaraju atributima iz relacijske sheme.

Relacija ima slijedeća svojstva (Tkalac, 1993, 21-22):

a) Shema relacije ne sadrži dva jednaka atributa

b) Redoslijed stupaca u relaciji nije bitan

c) Relacija ne sadrži dvije jednake n-torke

d) Redoslijed n-torki u relaciji nije bitan

RELACIJSKI MODEL PODATAKA

Relacijski model podataka prvi je model podataka definiran kao formalni sustav, a u čijem središtu se nalazi pojam relacije. Relacijski model predstavlja teorijske temelje za izgradnju relacijskih baza podataka (Tkalac, 1993, 8).

RELACIJSKA BAZA PODATAKA

Relacijska baza podataka je upravljački sustav baze podataka temeljen na relacijskom modelu. Većina modernih baza podataka su relacijske baze podataka (Tkalac, 1993, 8).

RELACIJSKA SHEMA

Relacijska shema je konačan skup atributa gdje je svaki atribut zadan na odgovarajućoj domeni, zajedno sa skupom ograničenja koja su zadana na tom skupu atributa (Tkalac, 1993, 20-21).

ROLLBACK

ROLLBACK operacija mehanizmu za upravljanje transakcija signalizira neuspješan završetak transakcije, što znači da će sve promjene podataka izvršene unutar transakcije biti zaboravljene i neće biti trajno zapisane (Date, 1995, 376).

SHEMA SNJEŽNE PAHULJICE

U skladištenju podataka shema snježne pahuljice sastoji se od jedne tablice činjenica na koju su vezane dimenzijske tablice koje mogu biti i hijerarhijski povezane. Zvjezdasta shema je poseban slučaj sheme snježne pahuljice (Ponniah, 2010, 133).

SKLADIŠTE PODATAKA

Skladište podataka je baza podataka posebne namjene koja služi za potrebe izvješćivanja i analize podataka. Punjenje skladišta podataka obavlja se iz izvora podataka u kojima se bilježi redovna aktivnost, to jest transakcijskih sustava ali i iz ad-hoc evidencija. Skladište podataka ima korisnu funkciju konzistentnosti izvješćivanja jer onemogućuje paralelno izvješćivanje iz

više međusobno neusklađenih izvora, već formira, tzv. jedan izvor istine (eng. single source of truth) (Ponniah, 2010, 129).

SLOG

Slog (eng. row, tuple) u tablici baze podataka predstavlja jedan skup vrijednosti i to po jednu vrijednost za svako polje tablice, odnosno predstavlja jedan redak (Date, 1995, 87).

SQL

SQL (eng. Structured Query Language) je jezik posebne namjene za rad relacijskim bazama podataka (Date, 2009, 304). Sastoji se od DDL (eng. data definition language) dijela za definiciju objekata baze, DML (eng. data manipulation language) dijela za obradu podataka, te dijela za kontrolu transakcija.

TABLICA

Tablica je objekt relacijske baze podataka koji sadrži skup slogova (redaka) čije vrijednosti su organizirane u polja (kolone). Tablica ima unaprijed definirana polja, a broj redaka može biti varijabilan (Date, 1995, 62).

TABLICA ČINJENICA

U skladištenju podataka tablica činjenica sadrži podatke o mjerenjima, činjenicama i događajima, tipično kroz vrijeme. Tablica činjenica je središnja tablica u zvjezdastoj shemi i shemi snježne pahuljice, a na nju su vezane dimenzijske tablice (Ponniah, 2010, 132).

TRANSAKCIJA

U relacijskim bazama podataka transakcija predstavlja jednu logički jedinicu posla. Sve izmjene podataka u bazi događaju se unutar neke transakcije. Proces obrade transakcije garantira ili upis svih izmjena podataka unutar transakcije ili ni jednog. Dakle, transakcija može uspjeti u cjelosti (commit operacija) ili može završiti neuspjehom i biti u cjelosti zaboravljena (rollback operacija) (Date, 1995, 375-379).

TRANSAKCIJSKI SUSTAV

Transakcijski sustav je baza podataka koja služi za bilježenje redovnih transakcija neke organizacije koje su nužne za njeno redovno funkcioniranje, recimo računovodstvenih događaja nekog poduzeća. Transakcijski sustavi služe kao izvor podataka za skladište podataka (Ponniah, 2010, 129).

VEZA

Veza (eng. join) je operacija koja spaja dvije tablice temeljem zajedničkih vrijednosti u kolonama koje se uspoređuju (Date, 1995, 53). Veza može biti:

a) Unutarnja - rezultirajuća tablica ima slogove koji su povezani u obje tablice

b) Vanjska - rezultirajuća tablica ima slogove koji se pojavljuju u bilo kojoj od tablica

ZAKLJUČAVANJE SLOGOVA

Mehanizam zaključavanja slogova usko je vezan uz mehanizam transakcija. Prije izmjene podataka unutar transakcijskog bloka, baza podataka osigurava da se slogovi koji će biti izmjenjeni, ne mogu izmijeniti od strane neke druge transakcije, označivši ih kao zaključane. Zaključane slogove može mijenjati samo transakcija koja ih je zaključala. Nakon završetka

transakcije oznake zaključanosti se brišu i slogovi su spremni za izmjene drugih transakcija (Date, 1995, 395).

ZVJEZDASTA SHEMA

U skladištenju podataka zvjesdasta shema se sastoji od jedne tablice činjenica na koju je vezan jedan sloj dimenzijskih tablica što znači da dimenzijske tablice nisu međusobno povezane, pa je stoga zvjezdasta shema specijalni slučaj sheme snježne pahuljice. Ovo je vrlo efikasna organizacija podataka u uvjetima izvještavanja sa čestim grupiranjem i filtriranjem podataka (Ponniah, 2010, 133).

ŽIVOTOPIS

Siniša Pavlović rođen je 15. siječnja 1972. godine u Zagrebu.

Oženjen je i otac dvoje djece.

1991. godine završava OC Nikola Tesla kao prirodoslovno - matematički tehničar, smjer matematika - informatika.

2000. godine diplomirao je na Ekonomskom fakultetu u Zagrebu, kao diplomirani ekonomist, smjer poslovna informatika.

Radio je na slijedećim radnim mjestima:

1994 - 1998 - Profit PP d.o.o. - samostalni programer i projektant

1998 - 2003 - Kom-PA d.o.o - samostalni programer i projektant

2003 - 2004 - Poslovna inteligencija d.o.o. - stariji BI konzultant

2004 - 2006 - Fina - stariji savjetnik, voditelj projekata, samostalni programer i modeler baza podataka

2006 - 2010 Tekstilpromet d.d. - koordinator IT projekata, projektant, BI analitičar

2010 - Raiffeisen Banka Hrvatska - viši reporting manager

Popis radova i aktivnih sudjelovanja na kongresima

V. Mateljan, S. Pavlović: Unapređenje komunikacije korisnika s korisničkim sučeljem korištenjem neuronske mreže // Suvremeni promet, Vol. 20, No. 3-4, svibanj-kolovoz 2002, Zagreb 2000.

V. Mateljan, V. Grbavac, S. Pavlović: Genetsko programiranje na primjeru unaprijed zadane funkcije. // Suvremeni promet. XIII. Međunarodni prometni simpozij, Prometni sustavi 2006., Opatija, Hrvatska, 2006.

Aktivno sudjelovanje/izlaganje na međunarodnom simpoziju: Suvremeni promet, 4, svibanj – kolovoz 2002, Zagreb.

Aktivno sudjelovanje/izlaganje na međunarodnom simpoziju: Suvremeni promet. XIII. Međunarodni prometni simpozij, Prometni sustavi 2006., Opatija, Hrvatska, 2006. Izlaganje.

Aktivno sudjelovanje na međunarodnom simpoziju: 12th International Conference on Information and Intelligent Systems IIS 2001, Varaždin 2001.