UNIVERZITET U NOVOM SADU

FAKULTET TEHNIČKIH NAUKA U NOVOM SADU

Fakultet Tehničkih nauka

Održavanje i kontrola kvaliteta elektroenergetskog softvera

Projektni zadatak - Dokumentacija

Novi Sad, 2017

1. SISTEM

U ovom odeljku se nalazi kratak opis projektnog zadatka, dizajn sistema, kao i arhitektura sistema.

1.1 OPIS SISTEMA

Implementirati klijent – servis model koji simulira sistem za vođenje istorijskih vrednosti. Glavna aplikacija je organizovana kao servis.

Modelovanje lokalnog sistema : bafer čuva podatke u obliku kolekcije – CollectionDescription (CD).

Collection Description sadrži :

- ID
- Dataset
- Dumping Property Collection

Dumping Property Collection sadrži :

- Niz Dumping Property-a

Dumping Property sadrži :

- Code
- Dumping Value

Prilikom slanja podataka Historical komponenti, bafer kreira posebnu strukturu Delta CD.
Delta CD sadrži :

- Transaction ID
- Collection Description Add
- Collection Description Update
- Collection Description Remove

Delta CD sadrži 3 Collection Description strukture, po jednu za operacije dodavanja, ažuriranja i brisanja skladišta.

Merenja Dumping Value (organizuje se kao pseudoslučajan broj u granicama od Pmin do Pmax).

Merenja u obliku Delta CD se šalju Historical-u nakon 10 pristiglih vrednosti. Stanje servisa, local / remote izbor , takođe se šalju Historical-u.

Ukoliko je prekidač na remote stanju Historical zadaje deadband.

Deadband predstavlja uslov da se podatak prosledi Historical-u. Jedini izuzetak je CODE_ANALOG i za njega se ne proverava deadband već se odmah ažurira.

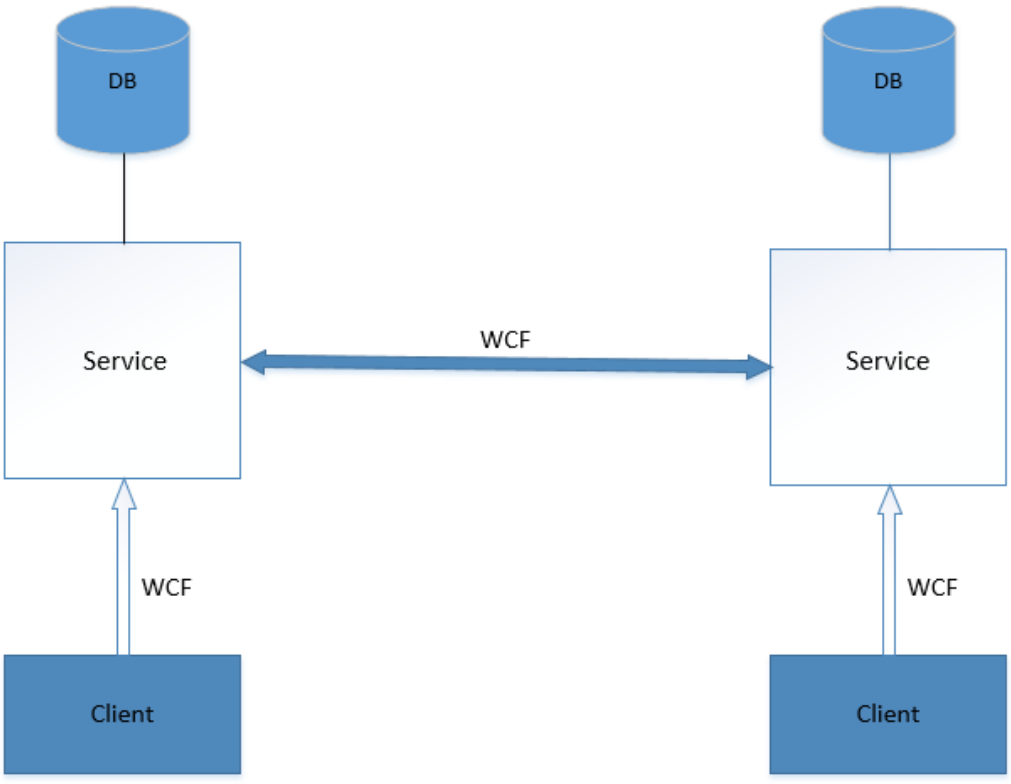
Dok je prekidač na local stanju, vrednosti se skladište u Delta CD komponentu i bazu bafera, a nakon prelaska u remote stanje vrednosti se iz Delta CD obrađuju i salju na Historical.

Potrebno je prikazati statistiku rada bafera, gde korisnik unosi proizvoljan termin za koji želi da pogleda statistiku rada bafera.

UI klijent je thin client tip aplikacije koji komunicira sa servisom radi prikupljanja i slanja podataka. On takođe prikazuje sistem sa detaljima rada, vrši vitualizaciju svih vrednosti, i podešava rezultate.

1.2 ARHITEKTURA SISTEMA

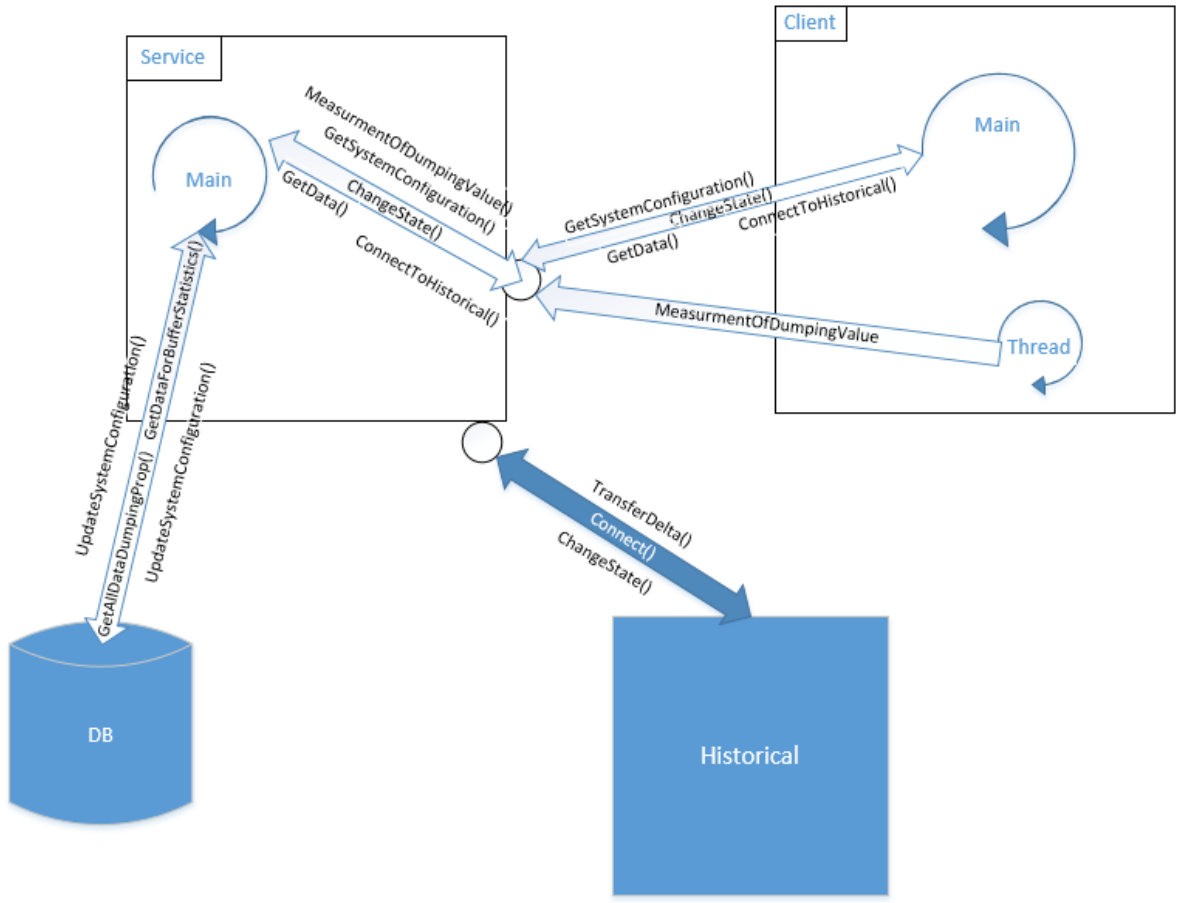
Arhitekturu sistema čine server, klijentska aplikacija, i baza podataka. U bazi podataka se nalaze dve tabele. Prva tabela *dumping_property* sadrži sve vrednosti merenja dumping property-ja, kao i trenutak kada je to merenje izvršeno(timestamp). U drugoj tabeli *system_configuration* se nalaze podaci vezani za trenutno stanje sistema kao što su deadband, stanje sistema, kao i min i max vrednosti koje može imati dumping value prilikom merenja. Klijent ima mogućnost provere trenutne konfiguracije sistema. Nakon izbora ove opcije, klijentu se prikazuje trenutno stanje sistema, deadband, kao i min i max vrednosti dumping value, koje servis dobavi. Ukoliko klijent izabere opciju prikaza svih podataka tada će servis dobaviti sve podatke merenja iz baze i te podatke klijent može pogledati i analizirati po potrebi. Statistika rada bafera omogućava klijentu da unese proizvoljan termin u okviru kojeg želi da vidi sva merenja, i tada će servis dobaviti iz baze samo merenja koja su izvršena u tom periodu i dostaviti podatke klijentu na uvid. Klijent takođe u svakom trenutku može da promeni stanje servisa na local ili remote stanje i tada će se aplikacija ponašati u skladu sa izabranim stanjem. U svakom trenutku klijent može da pokuša da se poveže sa historical-om radi odgovarajuće komunikacije, i tom prilikom je potrebno uneti ip adresu, kao i port računara na kojem je podignut historical servis.



Slika 1.1 Arhitektura sistema

1.3 DIZAJN SISTEMA

Dizajn sistema sadrži detaljan model, tj. osnovne komponente sistema, tip komunikacije, kao i tok podataka između njih, što je prikazano na slici 1.2.



Slika 1.2 Dizajn sistema

2. STRUKTURE PODATAKA

U ovom poglavlju su prikazane strukture podataka koje su korišćene za implementaciju sistema.

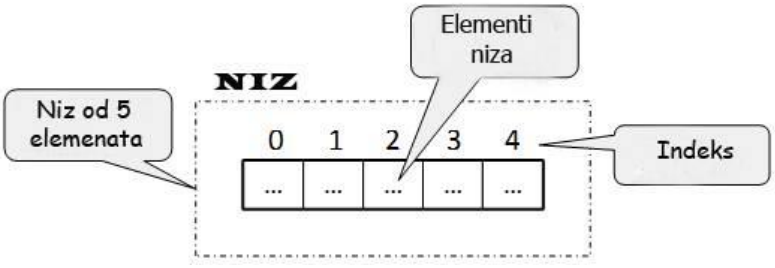
2.1 Dictionary

Dictionary je možda i najčešće korišćena kolekcija u C# koja predstavlja asocijativni kontejner. Asocijativna iz razloga što se prilikom skladištenja podacima dodeljuje ključ (key) koji služi za manipulisanje podacima u kolekciji. Dictionary važi za najbržu asocijativnu kolekciju jer u osnovi koristi HashTable strukturu. To znači da su vrednosti ključeva hash vrednosti, čime se znatno poboljšavaju performanse u radu sa ovom vrstom kolekcija. Razlika između HashTable i Dictionary kolekcije je u tome što je dictionary generički tip, što dicitonary čini type safe strkturom, odnosno nije moguće dodati slučajan tip elementa u kolekciju, a samim tip nije potrebno kastovanje podataka prilikom čitanja elemenata iz kolekcije. Vreme potrebno za dodavanje, uklanjanje i pretraga je relativno konstantno bez obzira na veličinu kolekcije.

Elementima dictionary-a se pristupa preko vrednosti ključa. Ukoliko nismo sigurni da li se ključ po kom pristupamo elementu dictionary-a zaista nalazi u dictionary-u, potrebno izvršiti odgovarajuću proveru pre pristupanja. Ta provera se najčešće izvršava koristeći metodu ContainsKey. Međutim, ukoliko postoji potreba za čestim pristupanjem elementima sa verovatnoćom da ključ ne postoji u dictionary-u, efikasnije je koristiti metodu *TryGetValue*. Ova metoda u pozadini proverava da li navedeni ključ postoji u kolekciji, a zatim vraća vrednosti elementa koji je dodeljen tom ključu. U slučaju da ključ ne postoji, biće vraćena default vrednost koja odgovara tipu elementa. Dakle, za pristupanje elementima preko ključa treba koristiti metodu *TryGetValue* kao mnogo pouzdaniji i efikasniji način pristupanja.

2.2 Nizovi

U rešavanju raznih problema javlja se potreba za postojanjem većeg broja podataka istog tipa koje predstavljaju jednu celinu. Zbog toga se u programskim jezicima uvodi pojam niza ili u opštem slučaju pojam polja. Ovo možemo predstaviti slikom.



Elementi niza su numerisani sa 0,1,2..,n-1.

Ovi brojevi se nazivaju indeksima elemenata niza. Broj elemenata u nizu predstavlja njegovu dužinu. Nizovi mogu biti različitih dimenzija. Najčešće se koriste jednodimenzionlani nizovi ili vektori i dvo-dimenzionalni nizovi ili matrice.

Niz predstavlja složeni tip podataka, sačinjen od nekolicine drugih podataka istog ili različitog tipa. Svaki podatak u nizu se naziva njegovim elementom, a svaki element ima svoj indeks, preko kojeg pristupamo tom elementu u nizu.

3. TESTIRANJE

Vršeno je više testiranja koja su opisana u daljem tekstu. Testovi pokrivaju sve pozitivne i negativne testne slučajeve.

Dobiti testiranja :

- Testovi smanjuju broj bagova u novim i postojećim funkcionalnostima
- Testovi predstavljaju dobru dokumentaciju
- Testovi smanjuju cenu pravljenja izmena
- Testovi unapređuju arhitekturu rešenja
- Testovi definišu funkcionalnosti
- Testovi teraju na razmišljanje pre pisanja koda

3.1 NUnit testovi

Nunit testiranje predstavlja razvojni proces u kojem se najmanje jedinice aplikacije, koje se mogu testirati, pojedinačno i nezavisno detaljno proveravaju u zavisnosti od željenog ponašanja.

Nunit testovi proveravaju ponašanje jedinice, koja se testira, samo u odnosu na slučajeve od interesa. Na taj način, programeri se ohrabruju da prave izmene u kodu, bez bojazi kako će se te izmene odraziti na rad ostalih jedinica ili na rad celog programa.

Provere određenih tvrdnji (assert) su jedna od najvažnijih stvari u nunit testiranju. Tvrdnje nam govore šta se očekivalo da se dogodi, ali ipak nije. Dobre provere tvrdnji nam pomažu pri praćenju bagova i lakšem razumevanju samih testova.

Prilikom pisanja nunit testova koristili smo lažiranje objekata (*mocking objects*), prilikom testiranja nekih metoda sa klijenta koji poziva metode servisa. U ovom slučaju servis je lažiran, tj. mokovan. Lažirani objekti (*mock objects*) se koriste u simulacijama, kako bi imitirali ponašanje stvarnih objekata na kontrolisan način. Lažirani objekti mogu biti od velike pomoći pri simuliranju rada baze podataka, mrežnog (web) servisa, događaja koji kreira klijent, spajanja sa spoljnom mrežom, kao i hardvera na kojem se izvršava program. Lažirani objekti imaju i mogućnost da proizvedu razne otkaze (failure) koje je teško reprodukovati u stvarnom svetu, kao što su: loša veza sa mrežom, spor server, zagušenje mreže i slično.

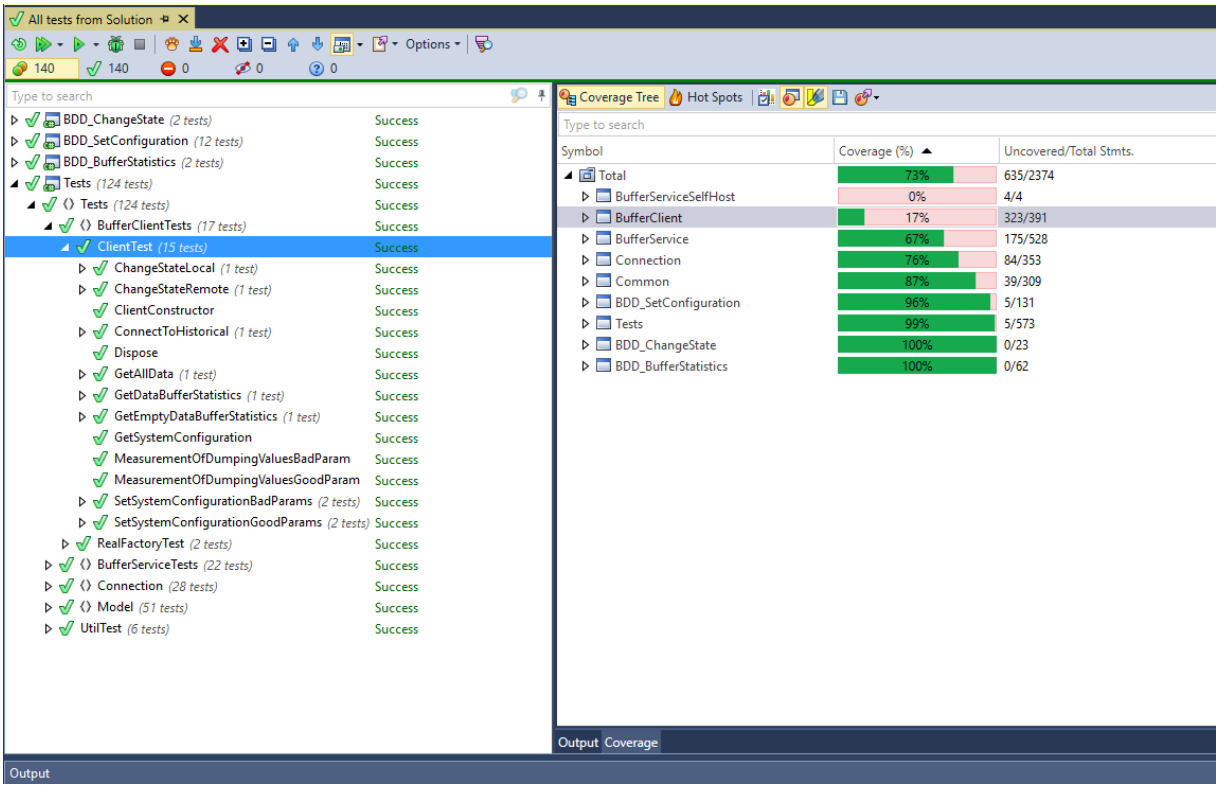
U računarskim naukama, pokrivenost koda je mera koja se koristi za opis koliki deo izvornog koda programa je pozvan od strane određenih testova. Program koji ima visok stepen pokrivenosti, mereno u procentima, ima veći deo svog izvornog koda prozvanog tokom testiranja, što sugeriše da ima manju šansu da sadrži neki skriveni problem, u odnosu na program koji ima manju pokrivenost. Mnoge metrike se mogu koristiti za računanje pokrivenosti koda. Osnovne su procenat pokrivenosti mogućih tokova programa (grana algoritma) i procenat pokrivenosti naredbi pozvanih tokom izvršavanja testova.

Kako bi izmerili koji procenat koda je izvršen od strane testova, jedan ili više kriterijuma se može primeniti. Kriterijum pokrivenosti je obično definisan kao pravilo ili uslov, koji testovi moraju da zadovolje. Osnovni su:

- Pokrivenost funkcija (function coverage) - da li su pozvane sve funkcije?
- Pokrivenost naredbi (statement coverage) - da li su pozvane sve naredbe?

- Pokrivenost uslovnih izraza (branch coverage) - da li su pozvani svi uslovni izrazi (if, switch...)? Odnosno, da li smo prošli kroz sve moguće putanje?
- Pokrivenost uslova (condition coverage) - da li je svaki pojedinačni uslov proveren i sa tačnom i sa netačnom vrednošću?

Na slici 1.3 je prikazan rezultat pokrivenosti koda za naš sistem.



Slika 1.3 Pokrivenost koda