UNIVERZITET U BEOGRADU

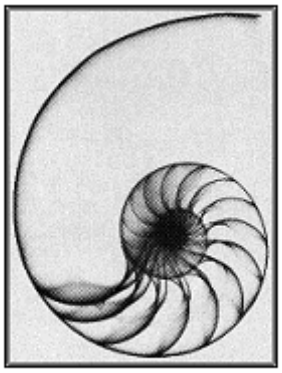
FAKULTET ORGANIZACIONIH NAUKA

(Katedra za softversko inženjerstvo)

PROJEKTOVANJE SOFTVERA

(SKRIPTA)

Autor: Dr Siniša Vlajić, doc.



Beograd - 2009.

Autor
Dr Siniša Vlajić

Naslov
PROJEKTOVANJE PROGRAMA (SKRIPTA)

Izdavač
Dr Siniša Vlajić doc. , Beograd

E-mail izdavača
vlajic@fon.rs

SADRŽAJ

UVOD

Kada se noć približi svitanju i kada se java počne mešati sa snom, počinješ da shvataš da još uvek radiš i da rešenje još uvek nije tu. Ustaješ i odlaziš od mašine koja daje i uzima od svakoga pomalo, neprimetno iz dana u dan, dok ne prođu godine. A onda jednog jutra poželiš da svedeš račune sa sobom, da na jednu stranu staviš sve šta si radio, čitao, proučavao a na drugoj da vidiš šta si naučio i uradio. I tada počinješ da shvataš nesrazmeru između uloženog rada i ostvarenih efekata. Pitaš se čemu sve to vodi, ako uopšte vodi, i koja je granica između iscrpljivanja i neke jasne spoznaje koja će utvrditi tvoje znanje i sabrati tvoje misli. Granica se pomera, pod naletima novih tehnologija, u korist iscrpljivanja. Otpor slabi vremenom i čovek postaje zarobljenik želje da sazna nešto novo, ne shvatajući da se slična (često ista) priča, po ko zna koji put prepričava na različite načine. Pitaš se gde su ta znanja koja su stabilna i koja se ne menjaju svaki put kada neko želi da proda svoj novi operativni sistem ili programski jezik. Pitaš se i odgovor očekuješ ...

...

Ovu knjigu sam počeo da pišem nakon nekoliko godina intezivnog proučavanja, Larmanove metode razvoja softvera, Jedinstvenog procesa razvoja softvera (JPRS) i primene uzora (paterna) u razvoju softvera.

U prvom delu knjige je objašnjeno šta je softverski sistem i kako se on razvija kroz sve faze životnog ciklusa. Prva faza – prikupljanje zahteva je opisana preko modela slučaja korišćenja. Druga faza – analiza, koja definiše logičku strukturu i ponašanje softverskog sistema, je opisana pomoću sistemskih dijagrama sekvenci i ugovora (ponašanje soft. sistema) i konceptualnog i relacionog modela (struktura soft. sistema). Treća faza – projektovanje, koja definiše fizičku strukturu i ponašanje softverskog sistema (arhitektura softverskog sistema), je opisana pomoću dijagrama klasa, dijagrama saradnje, sistemskih dijagrama sekvenci i dijagrama prelaza stanja. Svaki od navedenih dijagrama opisuje različite aspekte projektovanja softverskog sistema. Projektovanje elemenata arhitekture softverskog sistema (korisnički interfejs, aplikaciona logika, skladište podataka) je detaljno opisano. Na kraju su objašnjene faze implementacije i testiranja, pri čemu faza testiranja nije detaljno razmatrana. Metoda koja je korišćena u razvoju je uglavnom zasnovana na Larmanovoj metodi. Faze prikupljanja zahteva i analize su u potpunosti preuzeti od Larmanove metode,

dok je faza projektovanja malo pojednostavljena i prilagođena tronivojskoj arhitekturi. U fazi projektovanja je dodato projektovanje korisničkog interfejsa, o kome Larman govori u prvoj verziji knjige *Applying UML and patterns, a* koje je izbačeno u drugoj verziji iste knjige. Razvoj softverskog sistema prati studijski primer, koji detaljno razrađuje svaki od razmatranih koncepata. Svaka od faza je objašnjena preko skupa jednostavnih definicija, pravila i preporuka.

U drugom delu knjige su objašnjeni uzori koji se koriste u razvoju softvera. U tom smislu su objašnjeni oblici predstavljanja uzora i njihova klasifikacija. Na kraju su dati neki od GOF i GRASP uzora koji se koriste u primeru iz prvog dela knjige.

U trećem delu knjige su dati uzori za razne arhitekture distribucije objekata. Objašnjeni su ECF uzor, MVC uzor i J2EE arhitektura. Navedene arhitekture su potrebne kako bi se shvatio primer koji je dat u četvrtom delu knjige kada se objašnjava JPRS.

U četvrtom delu knjige je objašnjen JPRS. Date su osnovne osobine, struktura i faze JPRS. Nakon toga je razmatrano modeliranje poslovnih sistema. JPRS je objašnjen kao poslovni sistem, koji se sastoji od skupa poslovnih procesa (radnih tokova), pri čemu se svaki proces sastoji od skupa aktivnosti. Svakom od poslovnih procesa odgovora po jedna faza životnog ciklusa softverskog proizvoda. Na kraju je dat studijski primer koji je razvijen korišćenjem JPRS.

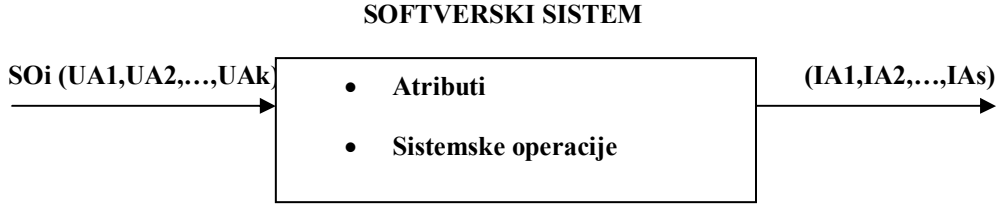
...

Nadam se da će navedeni materijal pomoći da se granica između iscrpljivanja i jasne spoznaje metodologija koje koristimo, pomeri u korist spoznaje i da će znanja koja naši studenti steknu biti stabilna u godinama što slede.

A u t o r

1. Razvoj softverskog sistema[1]

Softverski sistem[2] se sastoji od **atributa** i **sistemskih operacija**[3]. Atributi opisuju **strukturu** sistema, dok sistemske operacije opisuju **ponašanje** sistema. Sistemske operacije predstavljaju **osnovne** (atomske) **funkcije** sistema, koje se mogu koristiti iz okruženja sistema.
Dopustivi ulaz u softverski sistem je definisan potpisom (signaturom), koji sadrži naziv sistemske operacije koja se poziva i skup ulaznih argumenata. **Izlaz** iz softverskog sistema je predstavljen preko skupa izlaznih argumenata. Izlaz se dobija kao rezultat izvršenja neke od sistemskih operacija nad atributima sistema.



Atributi = {AT1, AT2,…,ATn}

Sistemske operacije = {SO1,SO2,..,SOm} - Osnovne funkcije sistema

SOi ∈ Sistemske operacije , i = (1,..,m)

UA1,UA2,…,UAk – Ulazni argumenti

(IA1,IA2,…,IAs) – Izlazni argumenti

Razvoj softverskog sistema sastoji se iz sledećih faza:
- Prikupljanje zahteva od korinisnika
- Analize
- Projektovanja
- Implementacije
- Testiranja

Svaka od faza je objašnjena[4]:
- odgovarajućim modelom koji je opisuje,
- definicijama elemenata modela,
- pravilima i preporukama kojih se treba držati i
- konkretnim studijskim primerom.

[1] Radeći na ovom materijalu, često sam dolazio u naizgled nerešive situacije kod povezivanje specifikacije problema sa njenom realizacijom (implementacijom). Ono što je pravilo najveći problem jeste shvatanje šta je softverski sistem i koje su njegove granice. Pitanje koje mi je zadavalo najviše glavobolje odnosilo se na korisnički interfejs: *Da li je korisnički interfejs deo softverskog sistema ili je on izvan sistema.*
Odgovor na navedeno pitanje, i na mnoga druga oko shvatanja šarenila mnogih pojmova i termina koji se koriste u softverskom inženjerstvu, dao mi je prof. Lazarević Branislav. On mi je rekao da je korisnički interfejs realizacija ulaza i/ili izlaza softverskog sistema. U prvom trenutku to mi je izgledalo nelogično, ali nakon nekog vremena, jasna i precizna argumentacija prof. Lazarevića me je uverila u ispravnost njegovih stavova. Navedeni stav smo predstavili preko sledeće definicije:

Def. Laz1: Korisnički interfejs predstavlja realizaciju ulaza i/ili izlaza softverskog sistema.

Navedena definicija, koliko god jednostavno izgledala, je otvorila put da softverski sistem shvatimo kao sistem u pravom smislu definicije sistema[PB].
Nadam se da će sistemski pristup u shvatanju razvoja softvera, konačno od softverskog inženjerstva napraviti nešto što će biti vredno naučnog a ne samo tehnološkog poštovanja.

[2] U daljem tekstu sistem.

[3] JPRS koristi termin sistemska operacija kada želi da naglasi razliku između operacija softverskog sistema i operacija koje se nalaze izvan softverskog sistema.

[4] U ovom trenutku nije detaljno razmatrana faza testiranje.

1.1 Zahtevi (Requirements)

Zahtevi predstavljaju svojstva i uslove koje sistem ili šire gledajući projekat mora da zadovolji [Larman]. Postoje različiti tipovi zahteva koje sistem mora da zadovolji i oni su kategorizovani prema **FURPS+** (**Functional** - *Funkcionalnost*, **Usability** - *Upotrebljivost*, **Reliability** - *Pouzdanost*, **Performance** - *Performanse*, **Supportability** - *Podrživost*) modelu:

- *Funkcionalnost* predstavlja sposobnost (**capabilities**) sistema da obezbedi zahtevane funkcije (ponašanje sistema). Zaštita (**security**) sistema predstavlja jednu od osnovnih funkcija koju sistem treba da obezbedi.
- *Upotrebljivost* predstavlja sposobnost sistema da se može jednostavno koristiti. To se postiže pomoću raznih uputstava i dokumentcije koji opisuju način njegovog korišćenja.
- *Pouzdanost* predstavlja sposobnost sistema da može uspešno obraditi problem (**failure**) koji se dešava u toku izvršenja sistema. U tom smislu sistem mora da obezbedi način oporavka (**recoverability**) podataka u slučaju nasilnog prekida rada sistema. Takođe sistem treba da omogući predviđanje (**predictability**) mogućih ponašanja sistema.
- *Performanse* sistema se odnose na vreme odziva (**response time**) zahtevanih funkcija, propusnu moć (**throughput**) mreže kroz koju prolaze podaci, tačnost (**accuracy**) izvršenja funkcija, mogućnost korišćenja odnosno raspoloživost (**availability**) funkcija sistema i način korišćenje raspoloživih resursa(**resource usage**) sistema.
- *Podrživost* sistema se odnosi na lakoću njegovog prilagođavanja (**adaptability**) i održavanja (**maintainability**), internacionalizaciju (**internationalization**) u smislu njegove prilagodljivosti različitim znakovnim sistemima koji se koriste u svetu i načinu konfigurisanja (**configurability**) sistema.

Zahtevi se često kategorizuju kao funkcionalni i ne funkcionalni zahtevi. Funkcionalni zahtevi definišu zahtevane funkcije sistema, dok ne funkcionalni zahtevi definišu sve ostale zahteve. U tom smislu ne funkcionalni zahtevi (upotrebljivost, pouzdanost, performanse i podrživost sistema) predstavljaju atribute kvaliteta (**quality attributes**) softverskog sistema.

U FURPS+ modelu znak '**+**' ukazuje na pomoćne zahteve koji se odnose na:

- *Implementaciju* (**Implementation**) sistema – do kojih granica se mogu koristiti raspoloživi resursi (**resource limitations**). Koji se programski jezici (**programming languages**) i alati (**tools**) mogu koristiti. Pored toga implemantacioni zahtevi se odnose i na hardver (**hardware**) koji će se koristiti.
- *Interfejs* (**Interface**) sistema – ograničenja koja postoje u komunikaciji sistema sa njegovim okruženjem (eksternim sistemima).
- *Operacije* (**Operations**) sistema – upravljanje sistemom i njegovim operacijama.
- *Pakovanje* (**Packaging**) sistema – način fizičkog organizovanja sistema u pakete, koji predstavljaju upravljive jedinice sistema.
- *Legalnost* (**Legal**) – mogućnost upotrebe sistema u smislu njegove legalnosti (licence i prava korišćenja sistema).

U daljem tekstu mi ćemo glavni naglasak staviti na razmatranje funkcionalnih zahteva, od njihovog prikupljanja do implementacije. Smatramo da istovremeno objašnjenje funkcionalnih i ne-funkcionalnih zahteva, koje prati studijski primer, može dosta da usloži shvatanje suštine razvoja jednog softverskog sistema iz ugla njegove osnovne funkcionalnosti. U studijskom primeru mi ćemo uvesti neki od ne-funkcionalnih zahteva koji su direktno povezani sa funkcionalnošću sistema (*pouzdanost i podrživost sistema*) i neke od pomoćnih zahteva (*implementacija, operacije i pakovanje sistema*) Ostali ne-funkcionalni i pomoćni zahtevi neće biti razmatrani, budući da su oni u najvećoj meri ortogonalno postavljeni u odnosu na funkcije sistema. To znači da oni ne utiču na shvatanje i objašnjenje razvoja funkcija sistema.

1.1.1 OPIS ZAHTEVA POMOĆU MODELA SLUČAJA KORIŠĆENJA

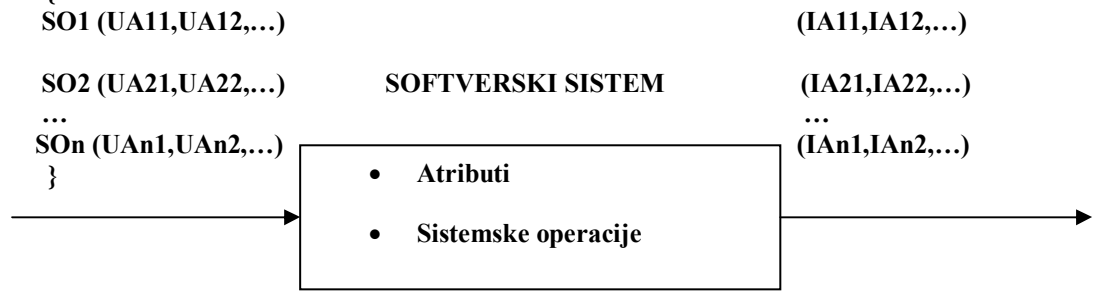
Zahtevi se kod Larmana opisuju pomoću UML[UML] *Modela Slučaja Korišćenja*[5] (**Use-Case Model**). Ukoliko je model SK veliki on se deli u pakete (**packages**).

Definicije Modela SK i njegovih elemenata

Def. ZA1: Model SK se sastoji od skupa SK, aktora (**actors**) i veza između SK i aktora.

Def. ZA2: Slučaj korišćenja opisuje skup **scenarija** (**use-case pojavljivanja**), odnosno skup željenih korišćenja sistema od strane aktora. Iz toga proizilazi da **scenario** opisuje jedno željeno korišćenje sistema od strane aktora. Scenario je opisan preko: a) sekvence akcija i b) interakcija između aktora i sistema. SK se sastoji iz glavnog i alternativnih scenarija.

Napomena: *Scenarija definišu željene fukcije sistema. Željene fukcije sistema, kada se izvršavaju, pozivaju po određenom redosledu osnovne funkcije sistema.*



Sistemske operacije = {SO1,SO2,..,SOn} - Osnovne funkcije sistema (atomske funkcije)

Slučajevi korišćenja = {SK1,SK2,..,SKp} - Željene funkcije sistema (molekulske funkcije)

SKi ∈ Slučajevi korišćenja, i =(1..p)

Def. ZA3: Aktor (učesnik) predstavlja spoljnog korisnika sistema. On postavlja zahtev sistemu da izvrši jednu ili više sistemskih operacija, po unapred definisanom scenariu[6].
Sistem odgovara na postavljeni zahtev aktora, šaljući mu vrednost izlaznih argumenata kao rezultat izvršenja operacija. Aktor se obično definiše kao neko ili nešto (npr: ljudi, kompjuterski sistemi ili organizaciona jedinica) što ima ponašanje.

Pravilo ZA1 (Nezavisnost scenarija SK): Scenarija SK ne treba da budu u međusobnoj interakciji[7]. Ona se trebaju definisati kao atomska, u smislu da se izvršavaju u potpunosti samostalno. Na taj način se olakšava njihov razvoj i održavanje[JPRS][8].

Pravilo ZA2 (Glass' low) [EA1] **:** Nedostaci kod definisanja zahteva su osnovni razlog mogućeg neuspeha u razvoju projekta (programa).

Pravilo ZA3 (Boehm's first low) [EA1]**:** Ukoliko se ne uoče greške u toku definisanja zahteva, iste se veoma teško mogu ukloniti u kasnijim fazama razvoja programa.

Pravilo ZA4 (Boehm's second low) [EA1] **:** Pravljenje prototipova značajno smanjuje moguće greške kod definisanja zahteva i njegovog razvoja, naročito kod definisanja korisničkog interfejsa.

---

[5] U daljem tekstu termin Slučajevi Korišćenja ćemo predstaviti sa skraćenicom SK.

[6] Zahtev za izvršenje jedne ili više sistemskih operacija se ne odigrava kontinualno nego diskretno. To znači da korisnik interaktivno poziva jednu po jednu sistemsku operaciju, u diskretnim vremenskim intervalima. Iz navedenog može da se zaključi da scenario opisuje interaktivno korišćenje softverskog sistema.

[7] U razgovoru sa prof. Bratislavom Petrovićem rekao sam da nezavisnost scenarija, povećava redudansu u opisu ponašanja sistema ali istovremeno znatno olakšava održavanje sistema. Rekao sam da su redudansa i održavanje obrnuto srazmerni. Prof. Petrović je rekao da je njihov proizvod verovatno neki koeficijent.
Bilo bi veoma interesantno kada bi neko uspeo da formalno objasni taj odnos i da pronađe navedeni koeficijent.

[8] Prva verzija našeg programa, koja je pravljena za firmu Perihard inženjering, nije uzela u obzir navedeno pravilo. Kod održavanja tog programa, kod koga su bila spregnuta scenarija, imali smo velikih problema. U sledećoj verziji programa u potpunosti smo izbacili sve interakcije koje su postojale između scenarija. Tu novu verziju programa koristimo i danas (posle 3 godine) i možemo da kažemo da je ona **neuporedivo** bolja od prve verzije programa u smislu njene nadogradnje i održavanja.

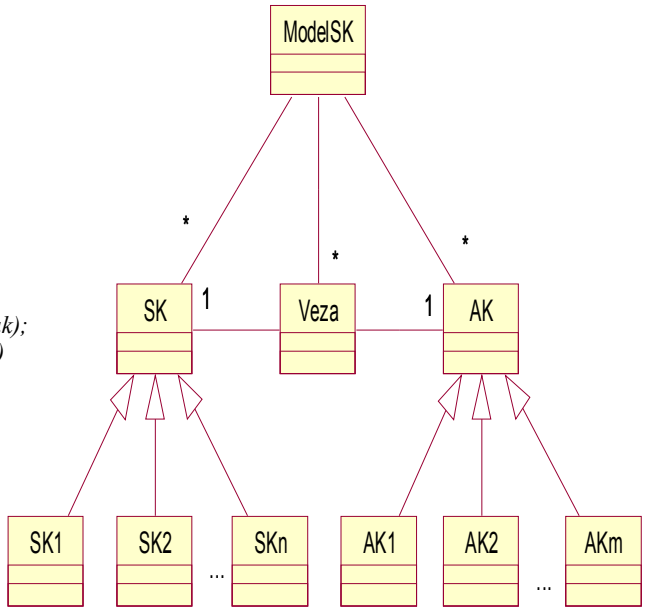
Objašnjenje definicija Modela SK[9]

U ovom poglavlju biće objašnjene Def ZA1 – DefZA3.

Objašnjenje Def. ZA1 - Model SK[10]

Model SK se može **objasniti** preko niže navedenog objektnog pseudokoda, koji je veoma sličan programskom jeziku Javi.

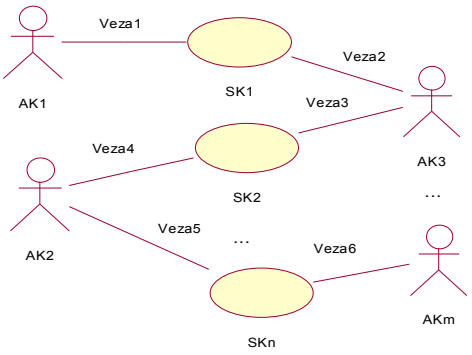
```
class ModelSK
{ SK skn[] = new SK[n];     // n – broj slučaja korišćenja.
  AK akn[] = new AK[m]     // m – broj aktora
 Veza vn[] = new Veza[p]; // p – broj veza između Aktora i Slučaja korišćenja

  ModelSK() // Inicijalizacija elemenata modela SK.
   { SK sk;
     AK ak;
     // Inicijalizacija SK
     for(int i=0; i<n-1; i++)
        {   Odredi(sk);  skn[i] = new SK(sk);
        }
    // Inicijalizacija AK
     for(int j=0; j<m-1; j++)
        {   Odredi(ak); akn[j] = new AK(sk);
        }
    // Inicijalizacija Veza
      for(int q=0; q<p-1; q++)
        {   Odredi(sk,ak); vn[q] = new Veza(sk,ak);
            // vn je skup (ne može imati duplikate)
        }
   }
}


class Veza
  { SK;
    AK;
  }
```



Model SK se može **objasniti** i preko UML-ovog dijagrama klasa

```
class SK
{...}
class SK1 extends SK        class SK2 extends SK        ...   class SKn extends SK
{...}                       {...}                             {...}
class AK
{...}
class AK1 extends AK        class AK2 extends AK        ...   class AKm extends AK
{...}                       {...}                             {...}
```

Model SK se **predstavlja** preko dijagrama SK UML-a[11]:



[9] U daljem tekstu ćemo, radi boljeg shvatanja definicija, pojmova i koncepata koje budemo razmatrali, koristiti objektni pseudokod koji je sličan Java programskom jeziku i neke od standardnih simbola koji se koriste u matematici (∀,∃,...).

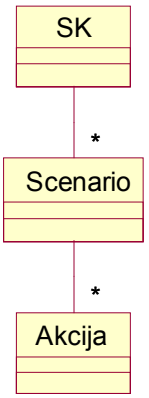
[10] Prof. Lazarević smatra da preslikavanje 1:1 između aktora i SK u potpunosti odražava semantiku te interakcije. Smatram da je prof. Lazarević u potpunosti u pravu.

[11] U konkretnim slučajevima model SK se isključivo predstavlja preko dijagrama SK.

Sa navedene slike se vidi da jedan aktor može da koristi više SK. Jedan SK može biti korišćen od više aktora.

Objašnjenje Def. ZA2 – Slučajevi korišćenja

```
class SK
{ ...

 ScenarioSK() // Scenario je opisan preko sekvence akcija
  { Akcija1();
    Akcija2();
    ...
   switch (Akcijai() ) //  U toku izvrsenja neke akcije (npr: Akcijai()) moze se desiti situacija koja ce napraviti
                          // novih d alternativnih scenarija.
       case 1: Akcijai1(); break;
       case 2: Akcijai2();break;
       ...
       case d:Akcijaid();break;
    }
   ...
   Akcijam-1();
   Akcijam();   // m – broj akcija
  } ...}
```



Dijagram klasa SK

Za navedeni primer imamo sledeći skup scenarija:

1. *Akcija1(); Akcija2();...;**Akcijai();**...;Akcijam-1(); Akcijam(); - glavni scenario*
2. *Akcija1(); Akcija2();...;**Akcijai();Akcijai1();**...;Akcijam-1(); Akcijam(); - AS1*[12]
3. *Akcija1(); Akcija2();...;**Akcijai();Akcijai2();** ...;Akcijam-1(); Akcijam(); - AS2*

...

4. *Akcija1(); Akcija2();...;**Akcijai();Akcijaid();** ...;Akcijam-1(); Akcijam(); - ASd*

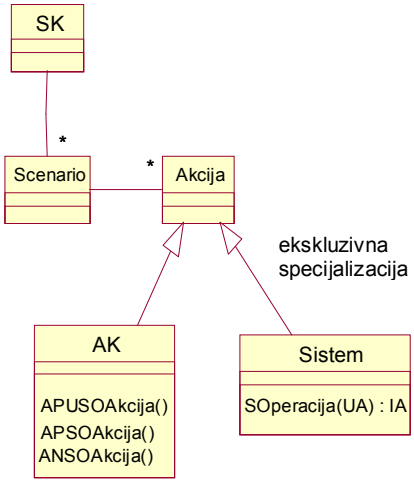
Objašnjenje Def. ZA3 – Aktor

Jednu akciju scenaria izvodi ili aktor ili sistem. U tom smislu:

- **Aktor** izvodi jednu od tri vrste akcija:
  a) **A**ktor **P**riprema **U**laz (**U**lazne **A**rgumente(**UA**)) za **S**istemsku **O**peracije(**APUSO**):
  Aktor.APUSOAkcija() {UA.set()}
  b) **A**ktor **P**oziva sistem da izvrši **S**istemsku **O**peraciju(**APSO**):
  Aktor.APSOAkcija() { IA = Sistem.SOperacija(UA) }
  Sistemskoj operaciji se prosleđuju ulazni argumenti. Nakon toga se izvršava **S**istemska **O**peracija (**SO**) koja kao rezultat daje jedan ili više izlaznih argumenata (**IA**).
  c) **A**ktor izvršava **N**e**S**istemsku **O**peraciju(**ANSO**):
  Aktor.ANSOAkcija();
- Na osnovu b) može se zaključiti da **Sistem** izvodi dve akcije u kontinuitetu:
  a) Sistem izvršava **S**istemsku **O**peraciju(**SO**):
  *IA Sistem.SOperacija(UA) { **izvršenje sistemske operacije;** return IA; }*
  a) Rezultat sistemske operacije (Izlazni argumenti(**IA**)) se prosleđuje do aktora:
  ***IA** Sistem.SOperacija(UA) { izvršenje sistemske operacije; **return IA;** }*

[12] AS – Alternativni Scenario

```
class SK
{ AK akn[] = new AK[m1];   // m1 – broj aktora nekog SK
  ...
  ScenarioSK()
    { Akcija1();
      Akcija2();
      ...
     Akcijam(); // m – broj akcija
    }

  Akcijai() // i ∈ (1,..m)  // Opšti oblik akcija scenarija
  {   AK ak = Odredi(akn);
      switch (šta radi aktor)
          {   case 1: ak.APUSOAkcija(); break;
              case 2: ak.APSOAkcija();    break;
              case 3: ak.ANSOAkcija();   break;
          }
  }
}
```



Dijagram klasa SK (detaljniji)

```
class AK
 {  // APUSO, APSO i ANSO operacije opisuju ponašanje aktora
    // Aktor može imati više APUS,APSO i ANSO Akcija. Ovde ćemo navesti opšti oblik  svake od navedenih vrsta
    // Akcija.
    APUSOAkcija() {UA.set()} //  aktor postavlja vrednosti ulaznim parametrima
    APSOAkcija()
          {   // Korisnik poziva sistemsku operaciju - Interakcija između aktora i sistema
              /*************************************************************************
                 IA = Sistem.SOperacija(UA);
                 // Matematički to bi moglo da se zapiše:
                        Sistem.SOperacija(UA) → (IA)
              /*************************************************************************

          }
    ANSOAkcija();
}
class Sistem
 {  // opisuju strukturu sistema
    Atribut1;
    Atribut2;
    ...
    Atributn;

// Sistem može da ima više sistemskih operacija. Sistemske operacije opisuju ponašanje sistema.
// Ovde ćemo navesti opšti oblik sistemskih operacija
IA SOperacija (UA);
}
```

Iz navedenog razmatranja možemo da zaključimo da postoji 5 tipova mogućih dešavanja u toku izvršenja jednog slučaja korišćenja:

1. Aktor priprema ulazne parametre za sistemsku operaciju(APUSO).
2. Aktor poziva sistem da izvrsi sistemsku Operaciju(APSO).
3. Aktor izvršava nesistemsku operaciju(ANSO).
4. Sistem izvršava sistemsku operaciju(SO).
5. Rezultat (izlazni argumenti) izvršenja sistemske operacije se prosleđuje do aktora (IA).

Način predstavljanja modela SK

SK se u početnim fazama razvoja softvera predstavljaju tekstualno [Larman, JPRS] dok se kasnije oni predstavljaju preko sekvencnih dijagrama[Larman][13], dijagrama saradnje[JPRS], dijagrama prelaza stanja ili dijagrama aktivnosti.
Teksualni opis SK ima sledeću strukturu[JPRS]:

- Naziv SK.
- Aktore SK.
- Učesnike SK.
- Preduslovi koji moraju biti zadovoljeni da bi SK počeo da se izvršava.
- Osnovni scenario izvršenja SK.
- Postuslovi koji moraju biti zadovoljeni da bi se potvrdilo da je SK uspešno izvrpen.
- Alternativna scenarija izvršenja SK.
- Specijalni zahtevi.
- Tehnološki zahtevi.
- Otvorena pitanja.

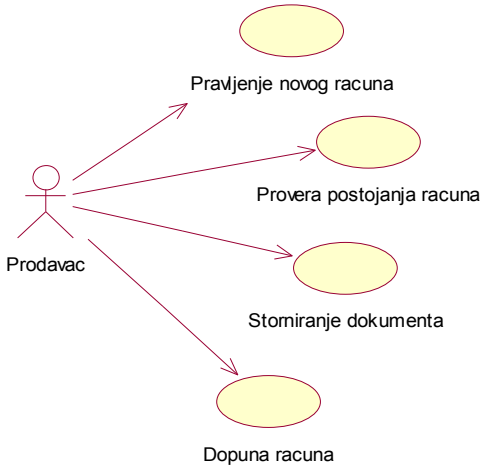
Operacije kod tekstualnog opisa SK se obično predstavljaju preko **glagola**, dok se atributi SK predstavljaju preko **imenica ili prideva.**

Konkretan primer modela SK

U našem primeru imamo sledeće SK-a:

1. *Pravljenje novog računa.*
2. *Provera postojanja računa.*
3. *Storniranje dokumenta.*
4. *Dopuna računa.*

Navedene SK koristi Prodavac (aktor).
Model SK se može predstaviti preko sledećeg dijagrama SK:



SKPZ1*: Slučaj korišćenja – Pravljenje novog računa*

Naziv SK
Pravljenje novog računa

Aktori SK
Prodavac
Učesnici SK
Prodavac i program (u daljem tekstu sistem).

***Preduslov**: Sistem je uključen i prodavac je ulogovan pod svojom šifrom. Sistem prikazuje formu za obradu računa (Kada korisnik pozove sistem da se izvrši (APSO), sistem inicijalno prikazuje formu na kojoj se nalazi zadnji uneti račun(IA)).*

[13] Larman koristi sekvencni dijagram kada opisuje SK.

Osnovni scenario SK

1. *Prodavac poziva sistem da kreira novi račun. (APSO)*
2. *Sistem kreira novi račun. (SO)*
3. *Sistem prikazuje prodavcu novi račun. (IA)*
4. *Prodavac unosi podatke u račun. (APUSO)*
5. *Prodavac poziva sistem da izracuna iznose stavki racuna i ukupni iznos racuna.(APSO)*
6. *Sistem računa iznose po svakoj stavci računa i ukupni iznos racuna.(SO)*
7. *Sistem prikazuje prodavcu izmenjen račun. (IA)*
8. *Prodavac kontroliše da li je uneo sve potrebne podatke na računu. (ANSO)*
9. *Prodavac poziva sistem da izvrsi konacnu obradu računa. (APSO)*
10. *Sistem obrađuje račun.(SO)*
12. *Sistem prikazuje prodavcu obrađen račun.(IA)*

Alternativna scenarija

- *5.1 Ukoliko sistem ne može da računa iznose stavki računa i ukupni iznos računa on prikazuje prodavcu poruku da ne može da obradi račun.(IA) Prekida se izvršenje scenaria.*
- *9.1 Ukoliko sistem ne može da obradi račun on prikazuje prodavcu poruku da ne može da obradi račun.(IA) Prekida se izvršenje scenaria.*

SKPZ2*: Slučaj korišćenja – Provera postojanja računa*

Naziv SK
Provera postojanja računa

Aktori SK
Prodavac
Učesnici SK
Prodavac i program.

***Preduslov**: Sistem je uključen i prodavac je ulogovan pod svojom šifrom. Sistem prikazuje formu za obradu računa (IA).*

Osnovni scenario SK

1. *Prodavac unosi broj računa koji želi da proveri. (APUSO)*
2. *Prodavac poziva sistem da proveri da li račun sa zadatim brojem postoji.(APSO)*
3. *Sistem proverava postojanje računa.(SO)*
4. *Sistem prikazuje prodavcu račun. (IA)*

Alternativna scenarija

- *3.1 Ukoliko račun sa zadatim brojem ne postoji sistem prikazuje prodavcu poruku da račun ne postoji(IA). Prekida se izvršenje scenaria.*

SKPZ3*: Slučaj korišćenja – Storniranje računa*

Naziv SK
Storniranje računa

Aktori SK
Prodavac
Učesnici SK
Prodavac i program.

***Preduslov**: Sistem je uključen i prodavac je ulogovan pod svojom šifrom. Sistem prikazuje formu za obradu računa (IA).*

Osnovni scenario SK

1. *Prodavac unosi broj računa koji želi da stornira. (APUSO)*
2. *Prodavac poziva sistem da proveri da li račun sa zadatim brojem postoji.(APSO)*
3. *Sistem proverava postojanje računa.(SO)*
4. *Sistem prikazuje prodavcu račun ukoliko postoji. (IA)*
5. *Prodavac poziva sistem da stornira zadati račun.(APSO)*
6. *Sistem stornira račun. (SO)*
7. *Sistem prikazuje prodavcu storniran račun. (IA)*

Alternativna scenarija

- *2.1 Ukoliko račun sa zadatim brojem ne postoji sistem prikazuje prodavcu poruku da račun ne postoji(IA). Prekida se izvršenje scenaria.*
- *5.1 Ukoliko račun ne može da se stornira sistem prikazuje prodavcu poruku da ne može da stornira račun (IA). Prekida se izvršenje scenaria.*

SKPZ4*: Slučaj korišćenja – Dopuna računa*

Naziv SK
Dopuna računa

Aktori SK
Prodavac
Učesnici SK
Prodavac i program.

***Preduslov**: Sistem je uključen i prodavac je ulogovan pod svojom šifrom. Sistem prikazuje formu za obradu računa (IA).*

Osnovni scenario SK
1. Prodavac unosi broj računa koji želi da dopuni. (APUSO)
2. Prodavac poziva sistem da proveri da li račun sa zadatim brojem postoji.(APSO)
3. Sistem proverava postojanje računa.(SO)
4. Sistem prikazuje prodavcu račun. (IA)
5. Prodavac unosi dopunske podatke u račun. (APUSO)
6. Prodavac poziva sistem da izracuna iznose stavki racuna i ukupni iznos racuna.(APSO)
7. Sistem računa iznose po svakoj stavci računa i ukupni iznos racuna.(SO)
8. Sistem prikazuje prodavcu izmenjen račun. (IA)
9. Prodavac kontroliše da li je uneo sve potrebne podatke na računu. (ANSO)
10. Prodavac poziva sistem da izvrsi konacnu obradu računa. (APSO)
11. Sistem obrađuje račun.(SO)
12. Sistem javlja prodavcu da je obradio račun.(IA)

Alternativna scenarija

- *3.1 Ukoliko račun ne postoji sistem prikazuje prodavcu poruku da račun ne postoji(IA). Prekida se izvršenje scenaria.*
- *6.1 Ukoliko sistem ne može da računa iznose stavki računa i ukupni iznos računa on prikazuje prodavcu poruku da ne može da obradi račun.(IA) Prekida se izvršenje scenaria.*
- *10.1 Ukoliko sistem ne može da obradi račun on prikazuje prodavcu poruku da ne može da obradi račun.(IA) Prekida se izvršenje scenaria.*

1.2. ANALIZA

Faza analize opisuje logičku strukturu i ponašanje softv. sistema (poslovnu logiku soft. sistema). Ponašanje softverskog sistema je opisano pomoću sistemskih dijagrama sekvenci, koji se prave za svaki SK, i pomoću ugovora o sistemskim operacijama, koje se dobijaju na osnovu sistemskih dijagrama sekvenci. Struktura softv. sistema se opisuje pomoću konceptualnog i relacionog modela.

1.2.1. PONAŠANJE SOFT. SISTEMA - SIST. DIJAGRAMI SEKVENCI

Ponašanje sistema se može opisati preko UML-ovih **sekvencnih dijagrama** [Larman]**,** odnosno preko **dijagrama saradnje**[JPRS]**.**

Definicije sistemskog dijagrama sekvenci i koncepta događaja

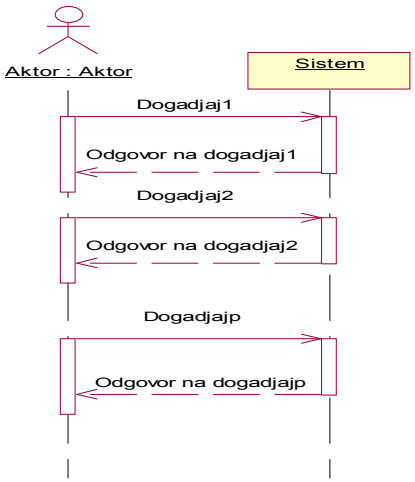
Def. AN1: Sistemski dijagram sekvenci prikazuje, za izdvojeni scenario SK, događaje u određenom redosledu, koji uspostavljaju interakciju između aktora i softverskog sistema.

Definicije događaja, koje ćemo navesti, su u kontekstu sistemskih dijagrama sekvenci.

Def. AN2: Događaj koji **napravi** aktor je pobuda za poziv sistemske operacije. Preciznije rečeno, događaj koji napravi aktor prihvata primalac događaja koji nakon toga poziva sistemsku operaciju. To znači da aktor ne poziva sistemsku operaciju neposredno već to čini preko posrednika (primaoca događaja). Poziv sistemske operacije ukazuje na interakciju između aktora i sistema. Za događaj koji predstavlja pobudu za poziv SO se često kaže da je to **sistemski događaj.**
Sistemski događaj je predstavljen potpisom (porukom)[14]. U tom smislu može da se shvati def. AN1.

Objašnjenje definicija sistemskog dijagrama sekvenci i koncepta događaja

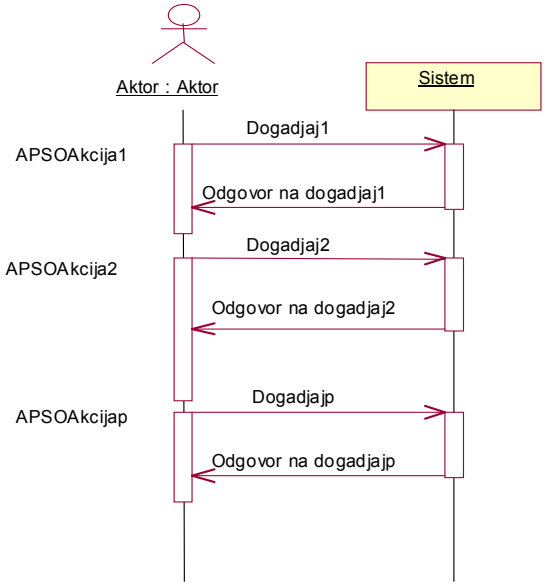
Objašnjenje Def. AN1 - Sistemski dijagram sekvenci



Objašnjenje Def. AN2 – Događaj

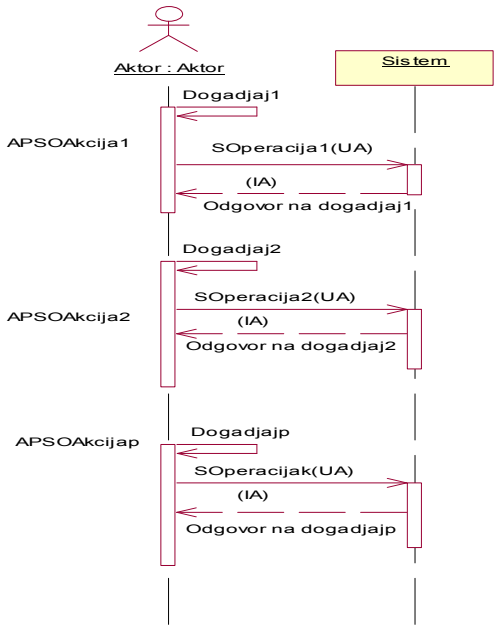
```
class SK
{ AK akn[] = new AK[m1];   // m1 – broj aktora nekog SK
 ...
 ScenarioSK()    { Akcija1();       ...    Akcijam(); // m – broj akcija   }

 Akcijai() // i ∈ (1,..m)  // Opšti oblik akcija scenaria
 {  AK ak = Odredi(akn);
    switch (šta radi aktor)
        {  case 1: ak.APUSOAkcija(); break;
           case 2: ak.APSOAkcija();  break;
           //  Događaje, u kontekstu sistemskih dijagrama sekvenci, prave aktori  jedino u okviru APSO akcija.

           case 3: ak.ANSOAkcija();   break;
        }
 }}
```

[14] Poruka ili potpis (signatura) imaju isto značenje, u kontekstu poziva SO.



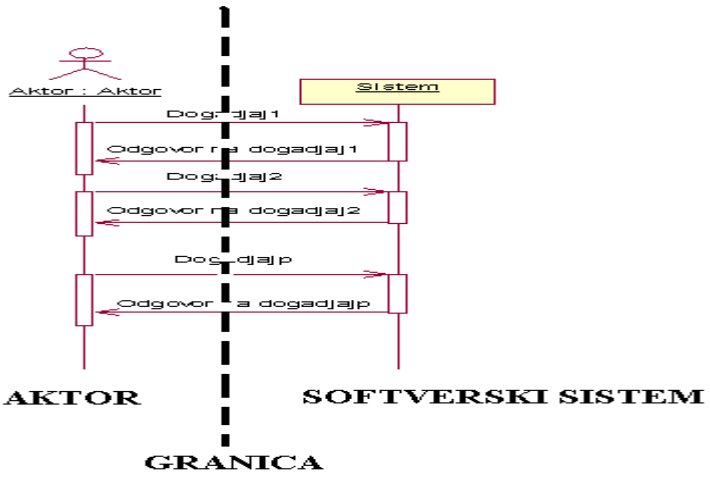
Sa navedene slike može da se zaključi da je broj događaja jednak broju APSO akcija koje izvodi aktor.

```
class AK
{    APSOAkcija()
       {    // Aktor pravi događaj na primaocu događaja
            Događaj dog= Aktor.PraviDogadjaj();
            // Primalac događaja prihvata događaj
            PrimalacDogađaja.PrihvatiDogađaj(dog);
       }
}

class PrimalacDogađaja
{ // Primalac događaja poziva sistemsku operaciju
  PrihvatiDogađaj(Događaj dog)  { IA = Sistem.SOperacija(UA); }
}
```



Događaj je predstavljen (reprezentovan) preko poruke.

Definisanjem sistemskih događaja jasno se određuje granica između aktora i softverskog sistema.



Kao rezultat analize scenarija SK dobijaju se zahtevi za izvršenje sistemskih operacija. Za svaku sistemsku operaciju se prave ugovori(kontrakti). Ugovore sistemskih operacija ćemo kasnije detaljno objasniti.

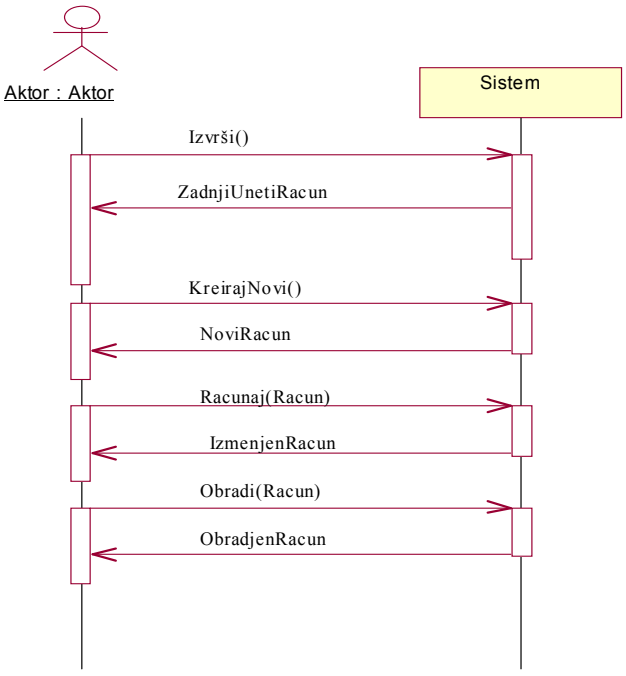
Konkretan primer sistemskog dijagrama sekvenci

Za svaki SK, preciznije rečeno za svaki scenarijo SK, prave se sistemski dijagrami sekvenci i to samo za APSO i IA akcije scenarija.

DS1: ***Dijagrami sekvenci slučaja korišćenja – Pravljenje novog računa***

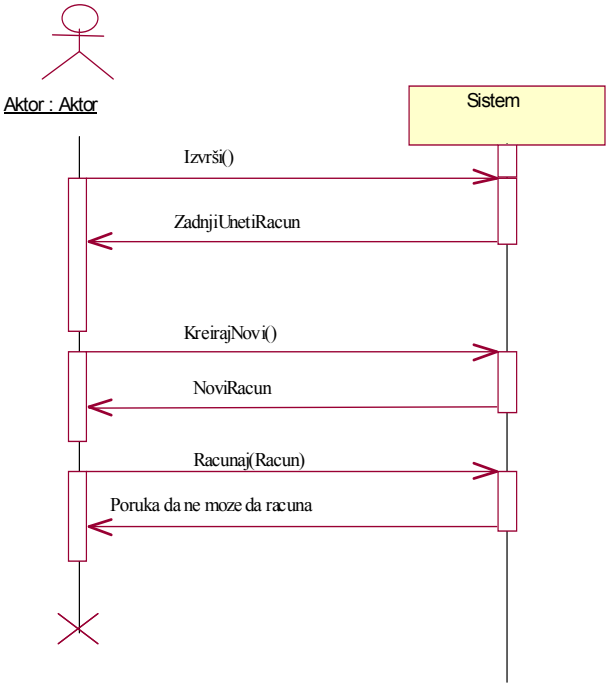
1. *Korisnik poziva sistem da se izvrši.(APSO)*
2. *Sistem prikazuje zadnji uneti račun.(IA)*
3. *Prodavac poziva sistem da kreira novi račun. (APSO)*
4. *Sistem prikazuje prodavcu novi račun. (IA)*
5. *Prodavac poziva sistem da izracuna iznose stavki racuna i ukupni iznos racuna.(APSO)*
6. *Sistem prikazuje prodavcu izmenjen račun. (IA)*
7. *Prodavac poziva sistem da izvrsi konacnu obradu računa. (APSO)*
8. *Sistem prikazuje prodavcu obrađen račun.(IA)*

Iz navedenog može da se zaključi da se na sistemskom dijagramu sekvenci ne vide APUSO, SO i ANSO akcije.

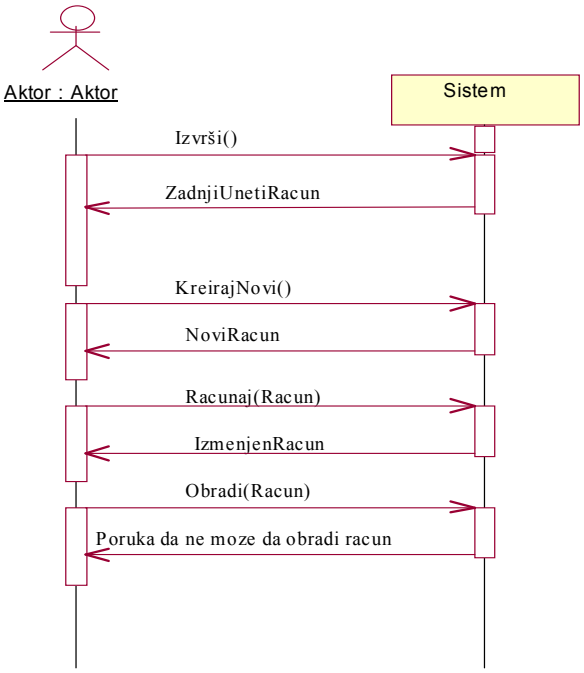


Alternativna scenarija

- *6.1 Ukoliko sistem ne može da računa iznose stavki računa i ukupni iznos računa on prikazuje prodavcu poruku da ne može da obradi račun(IA). Prekida se izvršenje scenaria.*



8.1 Ukoliko sistem ne može da obradi račun on prikazuje prodavcu poruku da ne može da obradi račun.(IA)

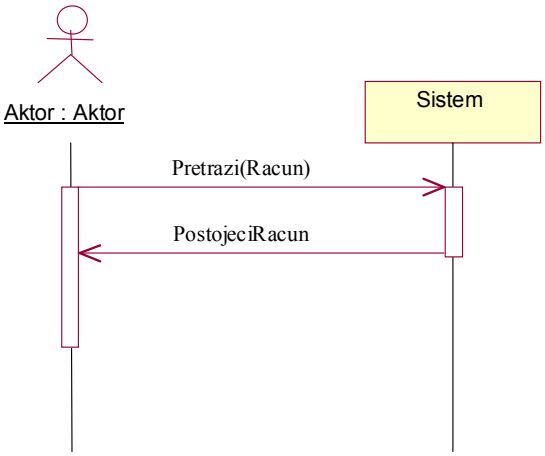


Sa navedenih sekvencnih dijagrama uočavaju se 4 sistemske operacije koje treba projektovati:

1. *ZadnjiBrojRacuna* **Izvršavanje**();
2. *NoviRacun* **KreiranjeNovogRačuna**();
3. *IzmenjenRacun* **Racunaj**(*Racun*);
4. *ObradjenRacun* **Obradi**(*Racun*)

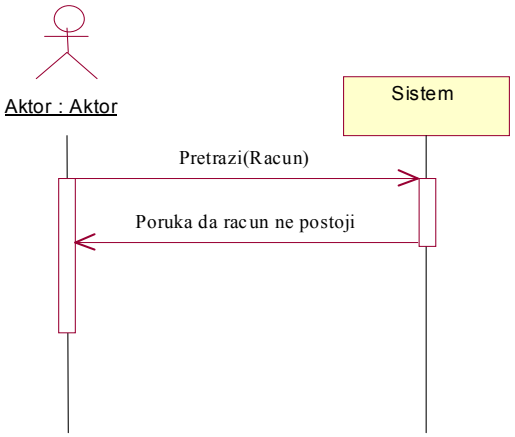
DS2: ***Dijagrami sekvenci slučaja korišćenja – Provera postojanja računa***

1. Prodavac poziva sistem da proveri da li račun sa zadatim brojem postoji.(APSO)
2. Sistem prikazuje prodavcu račun. (IA)



Alternativna scenarija

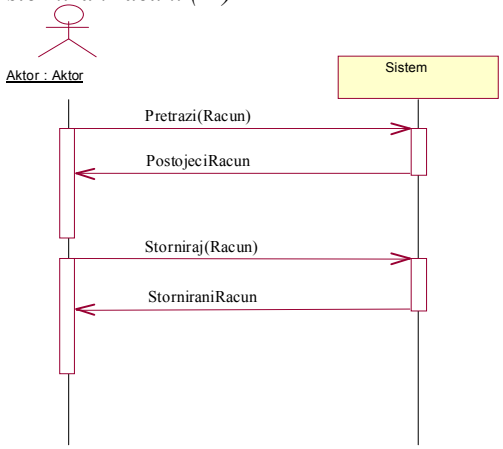
- *2.1 Ukoliko račun sa zadatim brojem ne postoji sistem prikazuje prodavcu poruku da račun ne postoji(IA).*



Sa navedenih sekvencnih dijagrama uočava se još jedna sistemska operacija koju treba projektovati:
1. PostojeciRacun **ProveraPostojanjaRacuna**(*Racun*)

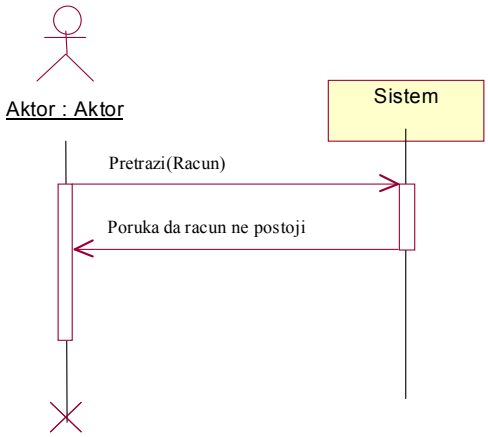
DS3: ***Dijagrami sekvenci slučaja korišćenja – Storniranje računa***

Osnovni scenario SK
1. Prodavac poziva sistem da proveri da li račun sa zadatim brojem postoji.(APSO)
2. Sistem prikazuje prodavcu račun ukoliko postoji. (IA)
3. Prodavac poziva sistem da stornira zadati račun.(APSO)
4. Sistem prikazuje prodavcu storniran račun. (IA)

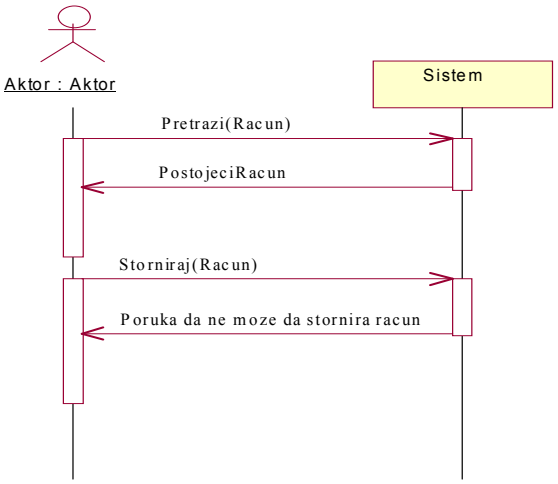


Alternativna scenarija

- *2.1 Ukoliko račun sa zadatim brojem ne postoji sistem prikazuje prodavcu poruku da račun ne postoji(IA). Prekida se izvršenje scenaria.*



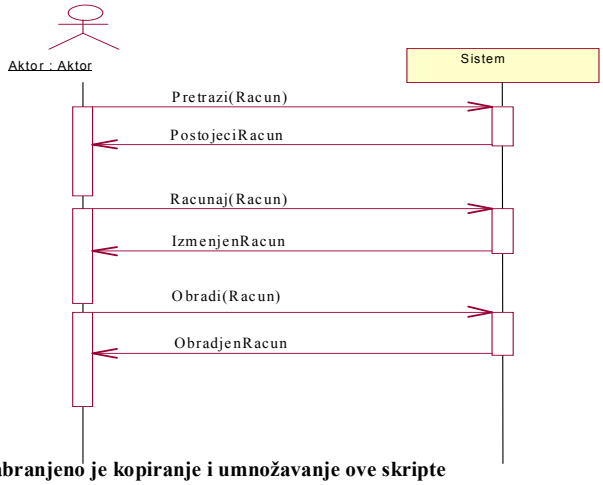
- *4.1 Ukoliko račun ne može da se stornira sistem prikazuje prodavcu poruku da ne može da stornira račun (IA).*



Sa navedenih sekvencnih dijagrama uočava se još jedna sistemska operacija koju treba projektovati:
1. StorniraniRacun **StornirajRacuna**(*Racun*)

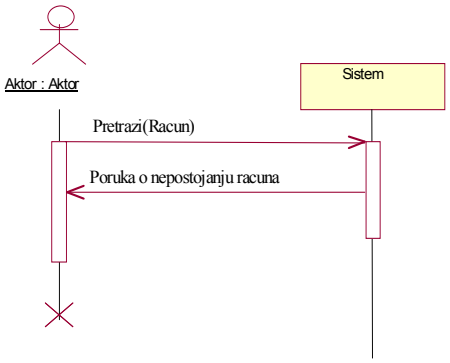
DS4: ***Dijagrami sekvenci slučaja korišćenja – Dopuna računa***

1. Prodavac poziva sistem da proveri da li račun sa zadatim brojem postoji.(APSO)
2. Sistem prikazuje prodavcu račun. (IA)
3. Prodavac poziva sistem da izracuna iznose stavki racuna i ukupni iznos racuna.(APSO)
4. Sistem prikazuje prodavcu izmenjen račun. (IA)
5. Prodavac poziva sistem da izvrsi konacnu obradu računa. (APSO)
6. Sistem prikazuje prodavcu obrađen račun.(IA)

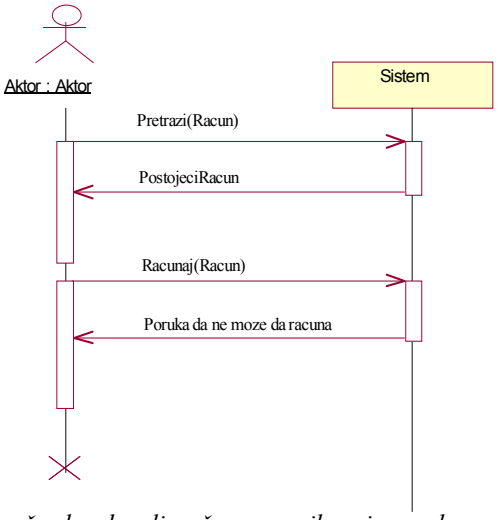


Alternativna scenarija

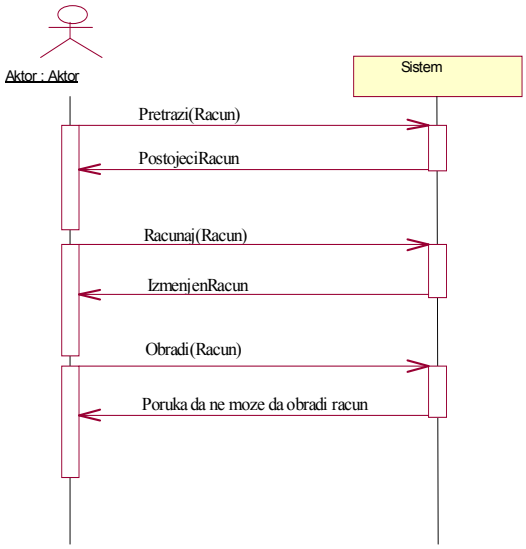
- *2.1 Ukoliko račun ne postoji sistem prikazuje prodavcu poruku da račun ne postoji(IA). Prekida se izvršenje scenaria.*



- *4.1 Ukoliko sistem ne može da računa iznose stavki računa i ukupni iznos računa on prikazuje prodavcu poruku da ne može da obradi račun.(IA) Prekida se izvršenje scenaria.*



- *6.1 Ukoliko sistem ne može da obradi račun on prikazuje prodavcu poruku da ne može da obradi račun.(IA) Prekida se izvršenje scenaria.*



Rezultat analize sist. dijagrama sekvenci

Kao rezultat analize scenarija dobijeno je ukupno 6 sistemskih operacija koje treba projektovati:

1. *ZadnjiBrojRacuna* **Izvrši** ();
2. *NoviRacun* **KreirajNovi** ();
3. *IzmenjenRacun* **Racunaj**(*Racun*);
4. *ObradjenRacun* **Obradi**(*Racun*);
5. *PostojeciRacun* **Pretrazi**(*Racun*);
6. *StorniraniRacun* **StornirajRacun**(*Racun*);

Za svaku od SO ulazni argument će biti istovremeno i izlazni argument. Svaka od SO shodno uspešnosti izvršenja vratiće odgovarajući signal, što će kasnije biti detaljno objašnjeno. Shodno navedenom objašnjenju SO imaće sledeći izgled:

1. *signal* **Izvrši** (*Racun*);
2. *signal* **KreirajNovi** (*Racun*);
3. *signal* **Racunaj**(*Racun*);
4. *signal* **Obradi**(*Racun*);
5. *signal* **Pretrazi** (*Racun*);
6. *signal* **Storniraj** (*Racun*);

1.2.2 PONAŠANJE SOFT. SISTEMA – DEFINISANJE UGOVORA (CONTRACTS) O SISTEMSKIM OPERACIJAMA

Za svaki od uočenih sistemskih operacija prave se ugovori. Navodimo njihove definicije.

Definicija sistemske operacije i ugovora

Def. AN3: Sistemska operacija opisuje ponašanje softverskog sistema. Sistemska operacija ima svoj potpis, koji sadrži ime metode i opciono ulazne i/ili izlazne argumente. Ona je javna i njoj se može pristupiti iz okruženja softverskog sistema.

Def. AN4: Ugovori se prave za sistemske operacije i oni opisuju njeno ponašanje. Ugovori opisuju šta operacija treba da radi, bez objašnjenja kako će to da radi. Jedan ugovor je vezan za jednu sistemsku operaciju.
Ugovori se sastoje iz sledećih sekcija[Larman]:

- **Operacija –** ime operacije i njeni ulazni argumenti
- **Veza sa SK –** imena SK u kojima se poziva sistemska operacija
- **Preduslov** – pre izvršenja sistemske operacije moraju biti zadovoljeni određeni preduslovi (sistem mora biti u odgovarajućem stanju).
- **Postuslovi –** posle izvršenja sistemske operacije u sistemu moraju biti zadovoljeni određeni postuslovi (sistem mora biti u odgovarajućem stanju ili se poništava rezultat operacije).

Def. AN5: Postuslovi SO ukazuju na to šta treba da se desi (efekti izvršenja SO), nakon izvršenja SO, a ne kako će to da se desi.
Postuslovi se odnose na:

- kreiranje ili brisanje pojavljivanja
- promenu atributa
- pravljenje ili uništavanje asocijacija

Postuslovi se izražavaju u prošlom vremenu, kako bi se naglasilo da je objekat došao u novo stanje, a ne da će dođe u novo stanje.Npr. *Stavka računa je kreirana*.Ne bi bilo dobro da se napiše: *Kreiranje stavke računa.*

Def. AN6: Preduslovi SO ukazuju na to šta je trebalo da se desi, kako bi SO mogla da se izvrši, a ne kako se to desilo.

Konkretan primer ugovora o sistemskim operacijama

UGOVOR UG1: *Izvrši*

Operacija: Izvrši(*Racun*):signal;
Veza sa SK: *DS1*
Preduslovi: -
Postuslovi: *Pročitan je zadnji zapamćeni račun.*

UGOVOR UG2: *KreirajNovi*

Operacija: KreirajNovi (*Racun*):signal;
Veza sa SK: *DS1*
Preduslovi:
Postuslovi: *Napravljen je novi račun.*

UGOVOR UG3: *Računaj*
Operacija: Racunaj(*Racun*):signal;

Veza sa SK: *DS1, DS4*
Preduslovi: *Ako je racun obradjen ili storniran ne moze se izvrsiti sistemska operacija.*
Postuslovi:

- *Izračunata je vrednost svake od stavki računa.*
- *Izračunata je ukupna vrednost računa.*

UGOVOR UG4: *Obradi*
Operacija: Obradi(*Racun*):signal;

Veza sa SK: *DS1, DS4*
Preduslovi: *Ako je racun obradjen ili storniran ne moze se izvrsiti sistemska operacija.*
Postuslovi:

- *Izračunata je vrednost svake od stavki računa.*
- *Izračunata je ukupna vrednost računa.*
- *Račun je obrađen.*

UGOVOR UG5: *Pretraži*
Operacija: Pretraži (*Racun*):signal;

Veza sa SK: *DS2, DS3, DS4*
Preduslovi:
Postuslovi: *Pročitan je račun ukoliko postoji.*

UGOVOR UG6: *Storniraj*
Operacija: Storniraj(*Racun*):signal;

Veza sa SK: *DS3*
Preduslovi: *Ako je racun storniran ne moze se izvrsiti sistemska operacija.*
Postuslovi: *Račun je storniran*

1.2.3 STRUKTURA SOFT. SISTEMA - KONCEPTUALNI (DOMENSKI) MODEL

Struktura soft. sistema se opisuje pomoću konceptualnog modela(Slika SSS). Navodimo definicije konceptualnog modela i njegovih elemenata.

Definicije konceptualnog modela i njegovih elemenata

Definicija AN7: Konceptualni model opisuje konceptualne klase domena problema. Konceptualni model sadrži konceptualne klase (domenske objekte) i asocijacije između konceptualnih klasa.Često se za konceptualne modele kaže da su to **domenski modeli** ili **modeli objektne analize.**

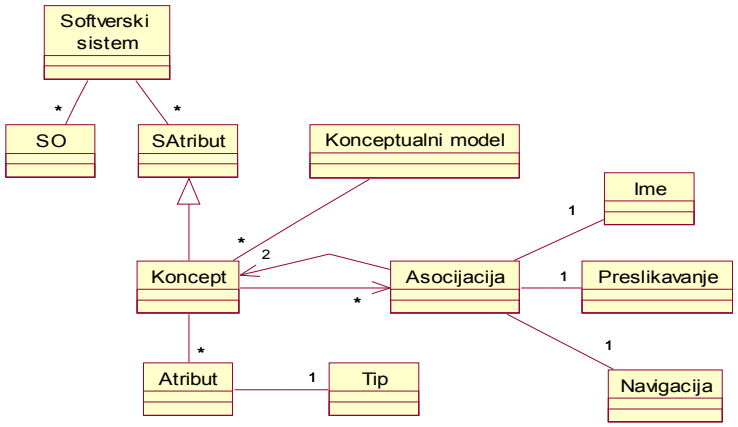
Definicija AN8: Koncepti (konceptualne klase) predstavljaju atribute[15] softverskog sistema. To znači da koncepti opisuju strukturu softverskog sistema. Konceptualne klase sastoje se od atributa, koji opisuju osobine klase.
Konceptualne klase treba razlikovati od softverskih klasa.

Definicija AN9: Atributi predstavljaju osobine koja se pridružuju do konceptualnih klasa. Svaki od atributa je vezan za određeni tip podatka.
Atribut ima konkretnu vrednost za konkretno pojavljivanje konceptualne klase.

Definicija AN10: Asocijacija je veza između konceptulanih klasa. Svaki kraj asocijacije predstavlja **ulogu** koncepta koji učestvuje u asocijaciji. Uloga sadrži ime, preslikavanje i navigaciju.
Ime uloge je zasnovano na formatu: ImeKoncKlase1 – Glagol – ImeKoncKlase2, gde glagol opisuje odnos između konceptualnih klasa u datom kontekstu.
Preslikavanje definiše koliko mnogo pojavljivanja konceptualne klase A može biti pridruženo jednom pojavljivanju konceptualne klase B.
Navigacija ukazuje na jednosmerne veze između konceptualnih klasa.
Između konceptualnih klasa može postojati više asocijacija.

Objašnjenje definicija konceptualnog modela i njegovih elemenata

Softverski sistem se sastoji od atributa (SAtribut) i sistemskih operacija (SO). Koncepti predstavljaju realizaciju atributa softverskog sistema.

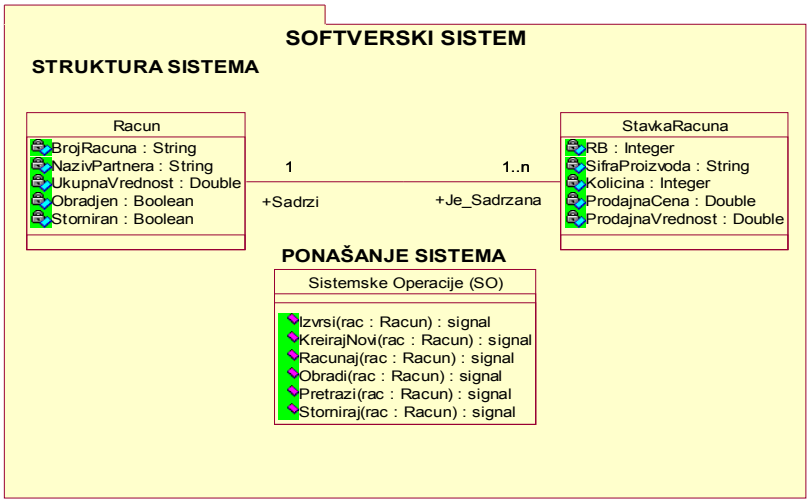


Slika SSS: Struktura Softverskog Sistema

Konkretan primer konceptualnog modela

[15] Koncepti se mogu **uočiti** iz UA koji se prosleđuju do softverskog sistema. Međutim treba naglasiti da UA nisu koncepti. UA su izvan softverskog sistema i oni predstavljaju ulaz u softverski sistem. Koncepti su unutar softverskog sistema i oni predstavljaju strukturu softverskog sistema.

Kao rezultat analize scenarija SK i pravljenja konceptualnog modela dobija se **logička struktura i ponašanje softverskog sistema**:



1.2.4 STRUKTURA SOFT. SISTEMA - RELACIONI MODEL

Na osnovu konceptualnog modela može se napraviti relacioni model, koji će da predstavlja osnovu za projektovanje relacione baze podataka[Ullman].
Na osnovu datog konceptualnog modela (Racun, StavkeRacuna) pravi se
relacioni model:

Racun(BrojRacuna[16], NazivPartnera,UkupnaVrednost,Obradjen,Storniran);
StavkaRacuna(BrojRacuna,RB,SifraProizvoda,Kolicina,ProdajnaCena,ProdajnaVrednost)

[16] Atributi koji predstavljaju primerne ključeve relacija biće podvučeni.

1.3. PROJEKTOVANJE

Faza projektovanja opisuje fizičku strukturu i ponašanje softv. sistema (arhitekturu softverskog sistema). Projektovanje arhitekture softverskog sistema obuhvata projektovanje aplikacione logike, skladišta podataka i korisničkog interfejsa. U okviru aplikacione logike se projektuju kontroler, poslovna logika i database broker. Projektovanje poslovne logika obuhvata projektovanje logičke strukture i ponašanja softverskog sistema.

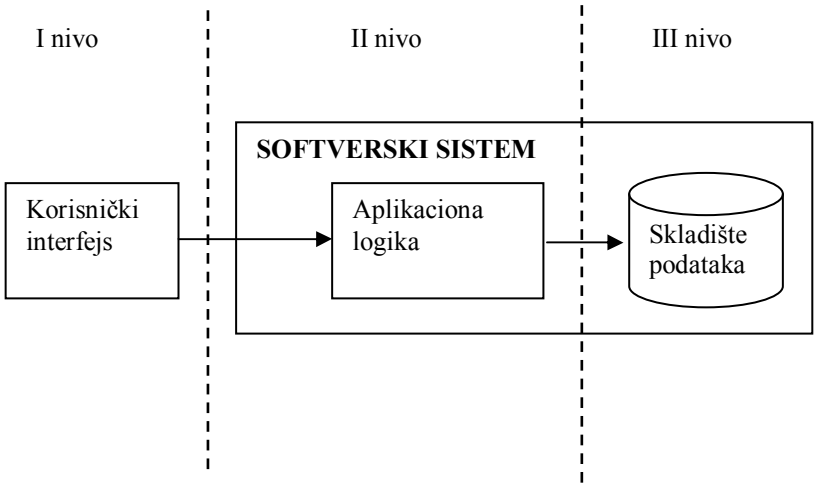
1.3.1 ARHITEKTURA SOFTVERSKOG SISTEMA

Pre nego što krenemo na projektovanje strukture i ponašanja softverskog sistema potrebno je da se definiše arhitektura softverskog sistema. Mi ćemo koristiti klasičnu tro-nivojsku arhitekturu [TK78,Garther 95].

Definicije i pravila arhitekture soft. sistema

Def. PRAR1: Tro-nivojska arhitektura se sastoji iz sledećih nivo(Slika TNA):

1. **Korisničkog interfejsa** koji predstavlja ulazno – izlaznu reprezentaciju softverskog sistema.
2. **Aplikacione logike** koja opisuje strukturu i ponašanje softverskog sistema.
3. **Skladišta podataka** koji čuva stanje atributa softverskog sistema.



Slika TNA: Tronivojska arhitektura

Pravilo PRpARH1: Aplikaciona logika se projektuje nezavisno od korisničkog interfejsa.

Pravilo PRpARH2: Aplikaciona logika može da ima različite ulazno-izlazne reprezentacije.

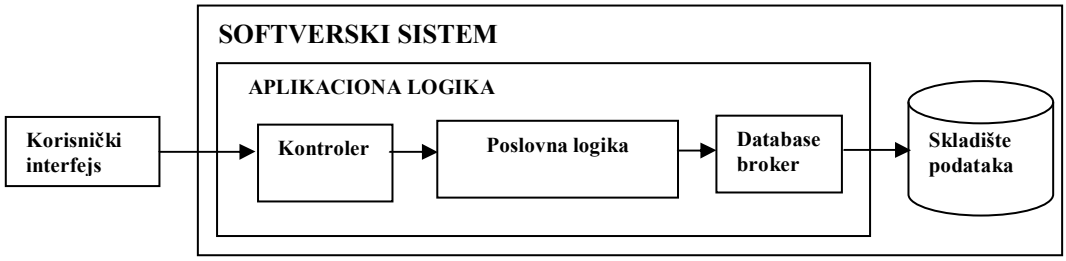
Pravilo PRpARH3 (**Model-View Separation Principle**): Aplikaciona logika (model) nema znanja o tome gde se nalazi korisnički interfejs (view).

Def PRAR2: Na osnovu **tronivojske arhitekture** su napravljeni savremeni **aplikacioni serveri**.

Def PRAR3: Aplikacioni serveri su odgovorni da obezbede servise koji će da omoguće realizaciju aplikacione logike softverskog sistema. Svaki aplikacioni server se sastoji iz tri osnovna dela:

1. deo za komunikaciju sa klijentim (kontroler)
2. deo za komunikaciju sa nekim skladištem podataka (baza podataka ili datotečni sistem)
3. deo koji sadrži poslovnu logiku

Def PRAR4: Poslovna logika je opisana sa strukturom (domenskim klasama) i ponašanjem (sistemskim operacijama).

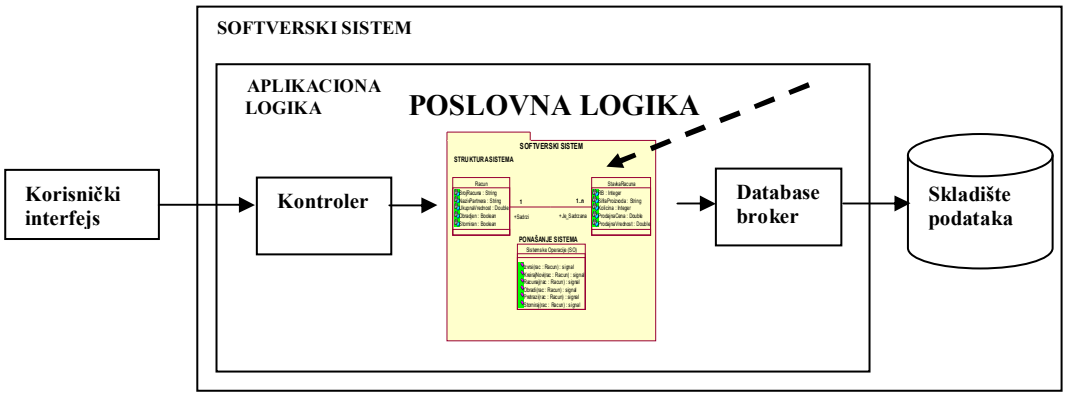


Def PRAR5: Kontroler je odgovoran da prihvati zahtev za izvršenje sistemske operacije od klijenta i da ga prosledi do poslovne logike koja je odgovorna za izvršenje sistemske operacije.

Def PRAR6: Database broker je odgovoran za komunikaciju između poslovne logike i skladišta podataka.

U daljem tekstu ćemo projektovati svaki od navedenih elemanata tronivojske arhitekture:
- kontroler
- poslovna logika – domenske klase
- poslovna logika – sistemske operacije
- database broker
- skladište podataka
- korisnički interfejs

Iz navedenog možemo da zaključimo da smo u fazama prikupljanja zahteva i analize dali specifikaciju strukture i ponašanja softverskog sistema, odnosno **specifikaciju poslovne logike softverskog sistema** (Slika ASSPL).

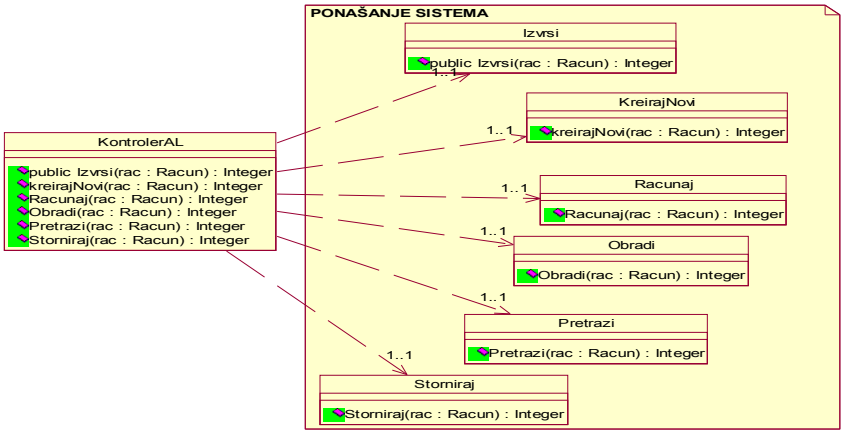


Slika ASSPL: Arhitektura soft. sistema – Poslovna logika

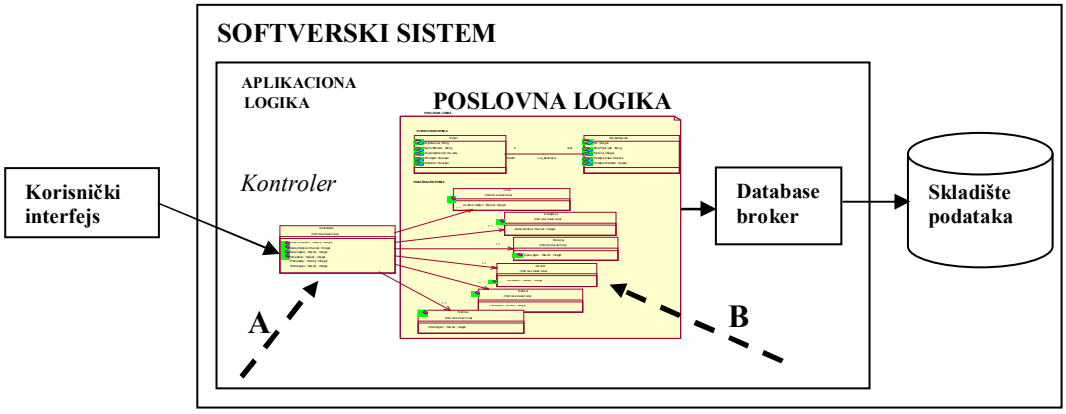
1.3.2 PROJEKTOVANJE APLIKACIONE LOGIKE – KONTROLER

Preko klase *KontrolerAL* prihvatamo zahtev od klijenata za izvršenje sistemskih operacija i iste prosleđujemo do odgovarajućih klasa koje su odgovorne za izvršenje SO.
Za svaku od SO prave se softverske klase koje treba da realizuju SO (Slika KOPO,ASSKOPO). Takve klase ćemo nazvati softverske klase ponašanja, jer SO opisuju ponašanje sistema.

```
class KontrolerAL // Kontroler aplikacione logike
{  public static int Izvrsi(Racun rac)          { return Izvrsi.Izvrsi(rac); }
   public static int KreirajNovi(Racun rac)  { return KreirajNovi.kreirajNovi(rac);}
   public static int Pretrazi(Racun rac)       { return Pretrazi.Pretrazi(rac);}
   public static int Racunaj(Racun rac)        { return Racunaj.Racunaj(rac);}
   public static int Obradi(Racun rac)         { return Obradi.Obradi(rac);}
   public static int Storniraj(Racun rac)     { return Storniraj.Storniraj(rac);}
}
```



Slika KOPO: Dijagram klasa koji prikazuje odnos između kontrolera i klasa odgovornih za izvršenje SO.



Slika ASSKOPO:Arhitektura soft. sistema - Kontroler (A) i klase (B) odgovorne za izvršenje SO

1.3.3 PROJEKTOVANJE STRUKTURE SOFT. SISTEMA (APLIKACIONA LOGIKA – POSLOVNA LOGIKA – DOMENSKE KLASE)

Na osnovu konceptualnih klasa prave se softverske klase strukture (Slika ASSSKS).

Konceptualne klase:

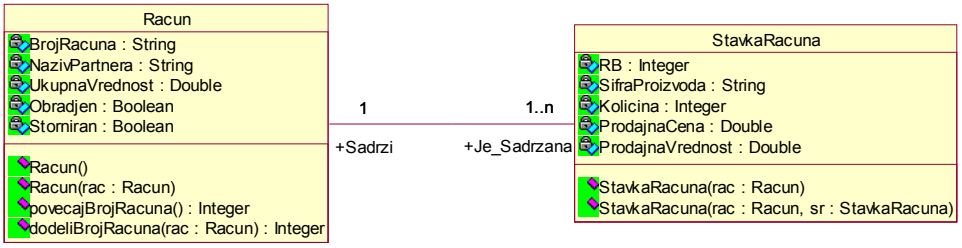
Softverske klase strukture:

```
class Racun
{  String BrojRacuna;
   String NazivPartnera;
  Double UkupnaVrednost;
  boolean Obradjen;
  boolean Storniran;
  StavkaRacuna []sracun;

// Default konstruktor klase Racun
Racun()
  { BrojRacuna = "NEMA";   NazivPartnera = "NEMA";
    UkupnaVrednost = new Double(0);
    Obradjen = false; Storniran = false;
    sracun = new StavkaRacuna[1];
    sracun[0] = new StavkaRacuna(this);
  }

// Copy konstruktor klase Racun
Racun(Racun rac)
  { BrojRacuna = new String(rac.BrojRacuna);
    NazivPartnera = new String(rac.NazivPartnera);
    UkupnaVrednost = new Double(rac.UkupnaVrednost.doubleValue());
    Obradjen = rac.Obradjen;
    Storniran = rac.Storniran;
    sracun = new StavkaRacuna[rac.sracun.length];
    for(int i=0;i<rac.sracun.length;i++)
      { sracun[i] = new StavkaRacuna(this,rac.sracun[i]);
      }
  }
// Klasa KreirajNovi u realizaciji SO kreirajNovi koristi metode povecajBrojRacuna() i dodeliBrojRacuna(int).
// Navodimo njihovu implementaciju.

/* Pocetak metoda koje koristi softverske klase ponašanja */
int povecajBrojRacuna()
{ int broj;
  try {if (BrojRacuna.equals("NEMA") || BrojRacuna == null)
         {BrojRacuna = "0001";}
       else
        if (BrojRacuna.equals("9999"))
             { return 48;} // Ne moze vise od 9999 racuna da se sacuva u bazi
        else
           {  broj = Integer.parseInt(BrojRacuna);
              broj++;    BrojRacuna = String.valueOf(broj);
              String pom1= new String("");
              for(int j=0; j<4-BrojRacuna.length(); j++)  { pom1 = pom1 + "0"; }
             BrojRacuna = pom1 + BrojRacuna;
           }
    } catch(Exception e)   { System.out.println("Izuzetak kod generisanja novog broja racuna: " + e); return 48;}
   return 47;
}
int dodeliBrojRacuna(Racun rac)
{   try { BrojRacuna = new String(rac.BrojRacuna);
        } catch(Exception e) { System.out.println("Nije dobro dodeljen broj racuna " + e); return 56;}
   return 55;
} /* Kraj metoda koje korist softverske klase ponašanja*/
}
```

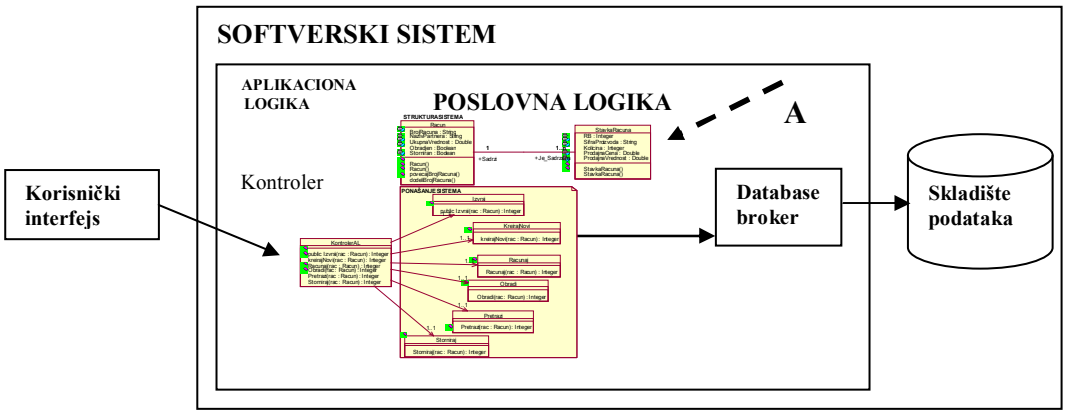
class StavkaRacuna
{ Integer RB; String SifraProizvoda; Integer Kolicina; Double ProdajnaCena; Double ProdajnaVrednost; Racun rac;

StavkaRacuna(Racun rac)
{ RB = new Integer(0); SifraProizvoda = new String("NEMA");
Kolicina = new Integer(0); ProdajnaCena = new Double(0);
ProdajnaVrednost = new Double(0); rac = rac1;}

StavkaRacuna(Racun rac,StavkaRacuna sr)
{ RB = new Integer(sr.RB.intValue()); SifraProizvoda = new String(sr.SifraProizvoda);
Kolicina = new Integer(sr.Kolicina.intValue()); ProdajnaCena = new Double(sr.ProdajnaCena.doubleValue());
ProdajnaVrednost = new Double(sr.ProdajnaVrednost.doubleValue()); rac = rac1;}
}



Dijagram softverskih klasa strukture



Slika ASSSKS: Arhitektura soft. sistema – Softverske klase strukture (A)

1.3.4 PROJEKTOVANJE PONAŠANJA SOFT. SISTEMA (APLIKACIONA LOGIKA – POSLOVNA LOGIKA – SISTEMSKE OPERACIJE)

Preporuka PR1: U početku projektovanja SO treba treba napraviti konceptualne realizacije (rešenja) za svaku SO.Konceptualne realizacije trebaju da budu direktno povezane sa logikom problema.

Preporuka PR2: Može se predpostaviti da se podaci (stanje softverskog sistema) čuvaju u bazi. U tom smislu mogu se pozvati neke od osnovnih operacija baze (insert,update,delete,find,...), bez ulaženja u način njihove realizacije.

Preporuka PR3: Aspekti realizacije koji se odnose na konekciju sa bazom, perzistentnost i transakcije treba izbeći u početku projektovanja SO. U kasnijim fazama navedeni aspekti trebaju ortogonalno da se povežu sa projektovanim rešenjima SO, kako bi se logika rešenja problema nezavisno od njih razvijala.

Preporuka PR4: Konceptualne realizacija se mogu opisati preko objektnog pseudokoda, dijagrama saradnje [Larman,JPRS], sekvencnih dijagrama[Larman,JPRS], dijagrama aktivnosti, dijagrama prelaza stanja ili dijagrama struktura [Budgen].

Za svaki od ugovora projektuje se konceptualno rešenje.

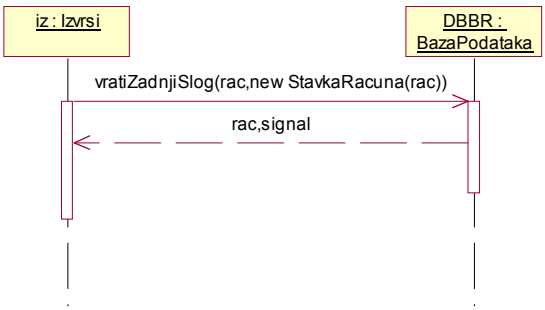
Projektovanje konceptualnih rešenja SO

UGOVOR UG1**:** ***Izvrši***

Operacija: Izvrši(*Racun*):signal;
Postuslovi: *Pročitan je zadnji zapamćeni račun.*

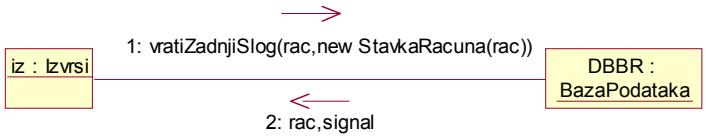
```
class Izvrsi
{
public static int Izvrsi(Racun rac)
  {    Izvrsi iz = new Izvrsi();
       return iz.izvrsenjeSO(rac);
  }

 int izvrsenjeSO(Racun rac)
  {  signal= DBBR.vratiZadnjiSlog(rac,new StavkaRacuna(rac)); //75,76
     // Rezultat navedene operacije, ukoliko je ista uspešno izvršena, vratiće zadnji zapamćeni račun,
    // što predstavlja postuslov SO izvrši.
     if (!stanjeOperacije(signal)) return 0;
     return 1;
  }

boolean stanjeOperacije(int signal)
   {  switch (signal)
      { case 75: Prikazi(signal,"Uspesno procitan zadnji slog iz baze, ako postoji");return true;
        case 76: Prikazi(signal,"Neuspesno procitan zadnji slog iz baze");return false;
      }
     return false;
   }
}
```

Interakcija između iz i DBBR objekta se može predstaviti preko interakcionih dijagrama (sekvencnog i dijagrama saradnje)



Sekvencni dijagram SO Izvrši[17].

[17] I sekvencni i dijagram saradnje čuvaju istu semantiku interakcije između objekata. Razlika između njih se ogleda u načinu njihovog predstavljanja.
Dijagrami saradnje opisuju interakciju između objekata u grafičkom ili mrežnom formatu, u kome objekti mogu da se postave bilo gde na dijagramu.



Dijagram saradnje SO izvrši.

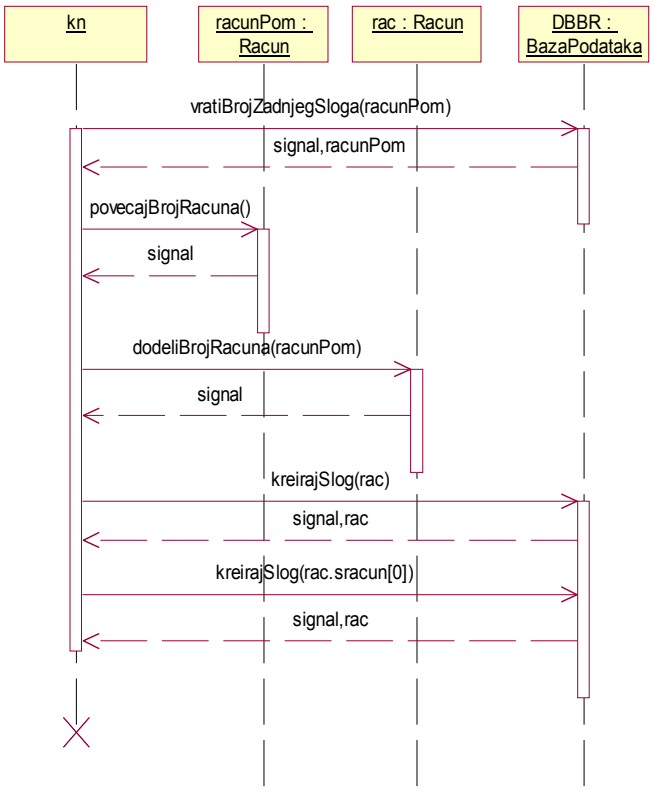
UGOVOR UG2: *KreirajNovi*

Operacija: KreirajNovi (*Racun*):signal;
Postuslovi: *Napravljen je novi račun.*

```
class KreirajNovi
{
 public static int kreirajNovi(Racun rac)
   {  KreirajNovi kn = new KreirajNovi();
      return kn.IzvrsenjeSO(rac);
   }

  int izvrsenjeSO(Racun rac)
  {Racun racunPom = new Racun();

   signal= DBBR.vratiBrojZadnjegSloga(racunPom); // 171,173
   if (!stanjeOperacije(signal)) return 0;

   signal = racunPom.povecajBrojRacuna(); // 47,48
   if (!stanjeOperacije(signal)) return 0;

   signal = rac.dodeliBrojRacuna(racunPom); // 55,56
   if (!stanjeOperacije(signal)) return 0;

   signal = DBBR.kreirajSlog(rac); // 45,46
   if (!stanjeOperacije(signal)) return 0;

   signal = DBBR.kreirajSlog(rac.sracun[0]); // 45,46
   // Rezultat izvršenja zadnje dve operacije kreirajSlog(), ukoliko su iste uspešno izvršene, daće kao rezultat novi
  // račun što predstavlja postuslov SO KreirajNovi.

   if (!stanjeOperacije(signal)) return 0;

   return 1;
}

boolean stanjeOperacije(int signal)
              {  switch (signal)
                   {  case 45:Prikazi(signal,"Uspesno kreiran novi slog"); return true;
                      case 46:Prikazi(signal,"Neuspesno kreiran novi slog");return false;
                      case 47:Prikazi(signal,"Uspesno povecan broj racuna"); return true;
                      case 48:Prikazi(signal,"Neuspesno povecan broj racuna"); return false;
                      case 55:Prikazi(signal, "Uspesno dodeljen broj racuna");return true;
                      case 56:Prikazi(signal, "Neuspesno dodeljen broj racuna");return false;
                      case 171:Prikazi(signal,"Uspesno vracen broj zadnjeg sloga"); return true;
                     case 173:Prikazi(signal,"Neuspesno vracen broj zadnjeg sloga");return false;
                   }
                 return false;
              }
}
```

Sekvencni dijagram opisuje interakciju u fence(ograda) formatu, u kome se svaki sledeći objekat dodaje desno u odnosu na predhodni objekat.
Navedeni dijagrami imaju određene prednosti i nedostatke:

- **Sekvencni dijagram:**
  - prednosti: jasno se prikazuje sekvenca poruka u vremenu, jednostavno opisivanje.
  - nedostatak: kada se dodaje novi objekat zauzima se prostor u desno.
- **Dijagram saradnje:**
  - prednost: jednostavnije se dodaju novi objekti na dijagramu. Bolje se ilustruju kompleksne grane, ciklusi i konkurentno ponašanje.
  - nedostatak: Teškoće kod posmatranja sekvence akcija. Složenija notacija.



Sekvencni dijagram SO KreirajNovi

UGOVOR UG3: ***Računaj***

Operacija: Racunaj(*Racun*):signal;

Preduslovi: *Ako je racun obradjen ili storniran ne moze se izvrsiti sistemska operacija.*

Postuslovi:

- *Izračunata je vrednost svake od stavki računa.*
- *Izračunata je ukupna vrednost računa.*

```
class Racunaj
{
 public static int Racunaj(Racun rac)
  {   Racunaj r = new Racunaj();
      return r.IzvrsenjeSO(rac);
  }

int izvrsenjeSO(Racun rac)
 { Racun rac1 = new Racun(rac);
    if (!nadjiRacuniVratiGa(rac1)) return 0;
    if (!Preduslov(rac1)) return 0;
    if (!racunajUkupneVrednosti(rac)) return 0;
    if (!azurirajRacun(rac1,rac))  return 0;
    if (!brisiStavkeRacuna(rac1)) return 0;
   dodavanjeStavkiRacuna(rac); return 0;
 }

private boolean nadjiRacuniVratiGa(Racun rac)
  { signal = DBBR.nadjiSlogiVratiGa(rac,new StavkaRacuna(rac)); // 71,72
    if (!stanjeOperacije(signal)) return false;
    return true;
  }


 private boolean Preduslov(Racun rac1) // 94,77
    {  // 1. Ako je racun obradjen nad njim se ne moze izvrsiti operacija racunanja stavki.
       // 2. Ako je racun storniran nad njim se ne moze izvrsiti operacija racunanja stavki.
      if ((rac1.Obradjen == true) || (rac1.Storniran == true))
```

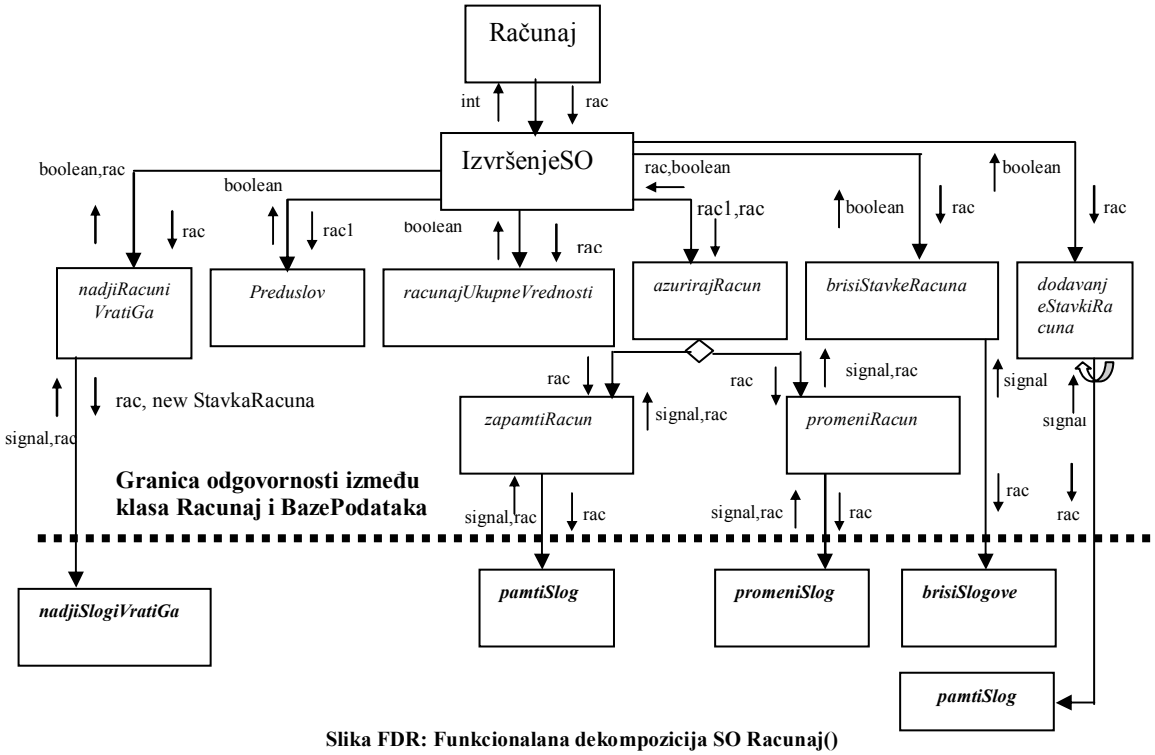
```
        {   signal = 94;  return false;    }
       signal = 77; // Zadovoljen preduslov
       return true;
  }

 private boolean racunajUkupneVrednosti(Racun rac) // 79,78
   { double Suma =0;
      try {  for(int i=0;i<rac.sracun.length;i++)
             { rac.sracun[i].ProdajnaVrednost = new Double (rac.sracun[i].ProdajnaCena.doubleValue() *
                                                            rac.sracun[i].Kolicina.intValue());
               Suma = Suma +  rac.sracun[i].ProdajnaVrednost.doubleValue();
             }
            rac.UkupnaVrednost = new Double(Suma);
           } catch(Exception e) { signal = 78; return false;}
      signal = 79;
     return true;
 }

private boolean azurirajRacun (Racun rac1,Racun rac)
     {  if (rac1.BrojRacuna.equals("NEMA"))
          return zapamtiRacun(rac);
        else
         return promeniRacun(rac);
     }

private boolean zapamtiRacun(Racun rac)
   { signal = DBBR.pamtiSlog(rac); //31,32
     if (!stanjeOperacije(signal)) return false;
    return true;
   }

private boolean promeniRacun(Racun rac)
   {  signal = DBBR.promeniSlog(rac); //35,36
              if (!stanjeOperacije(signal)) return false;
              return true;
   }


private boolean brisiStavkeRacuna(Racun rac) // 33,34
   { signal = DBBR.brisiSlogove(new StavkaRacuna(rac));
     if (!stanjeOperacije(signal)) return false;
     return true;
  }

private boolean dodavanjeStavkiRacuna(Racun rac) //31,32
{  for(int i=0;i<rac.sracun.length;i++)
      { signal = DBBR.pamtiSlog(rac.sracun[i]);
        if (!stanjeOperacije(signal)) return false;
      }
  return true;
      }

boolean stanjeOperacije(int signal)
          {
              switch (signal)
               { case 31: Prikazi(signal,"Uspesno zapamcen slog");return true;
                 case 32: Prikazi(signal,"Bezuspesno zapamcen slog");return false;
                 case 33: Prikazi(signal,"Slogovi uspesno obrisani");return true;
                 case 34: Prikazi(signal,"Slogovi bezuspesno obrisani");return false;
                 case 35: Prikazi(signal,"Uspesno promenjen slog");return true;
                case 36: Prikazi(signal,"Bezuspesno promenjen slog");return false;
                case 71: Prikazi(signal,"Uspesno pretrazivanje slogova");return true;
                case 72: Prikazi(signal,"Neuspesno pretrazivanje slogova");return false;
                case 77: Prikazi(signal,"Preduslov je zadovoljen");return true;
                case 78: Prikazi(signal,"Bezuspesno izracunata ukupna vrednost");return false;
                case 79: Prikazi(signal,"Uspesno izracunata ukupna vrednost");return true;
                case 94: Prikazi(signal,"Ne moze da radi sa racunom koji je obradjen ili storniran");return false;
               }
                return false;
          }
}
```

Pravilo PRpARH4 (Simon's law)*: Hijerarhijske strukture smanjuju složenost*

Pošto je navedena SO složenije nego predhodne, istu smo **funkcionalno dekomponovali**(Slika FDR)**,** kako bi se jasnije shvatile apstrakcije datog rešenja[18].

Funkcionalna dekompozicija se obično predstavlja preko **Dijagrama struktura** (Structure charts) [Budgen, Cir1-2].

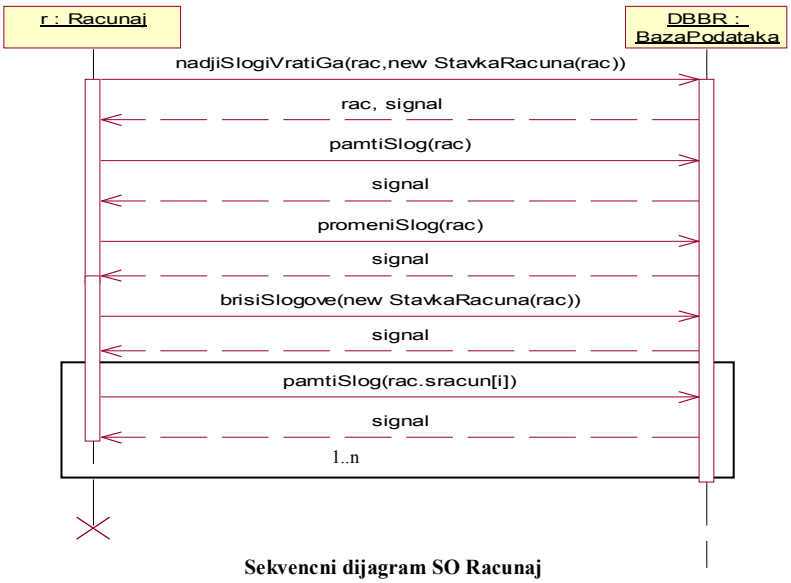
Dijagrami struktura opisuju hijerarhiju funkcija (metoda) pomoću stabla. Svaka od funkcija, koja se opisuje pomoću dijagrama struktura, se nalazi na nekom nivou hijerarhije u stablu. Ako funkcija *n* poziva funkciju *m,* ona se nalazi na višem nivou hijerahije u odnosu na funkciju *m*. Ulazni i izlazni argumenti funkcija se predstavljaju na stablu.



Slika FDR: Funkcionalana dekompozicija SO Racunaj()

Interakcija između klasa Racunaj i BazaPodataka predstavlja se preko sekvencnog dijagrama, na kome se prikazuju samo metode koje uspostavljaju interakciju između navedenih klasa.

[18] Semantika dijagrama strukture je ekvivalentna semantici dijagrama saradnje. Međutim dijagram strukture je hijerarhijski uređen dok to ne važi za dijagram saradnje (prof. Branislav Lazarević)



Sekvencni dijagram SO Racunaj

Na navedenoj slici se vidi da se pamcenje stavki računa, koje se obavlja u ciklusu, predstavlja preko uokvirenog pravugaonika koji okružuje operacije koje se ponavljaju.

UGOVOR UG4: ***Obradi***

Operacija: Obradi(*Racun*):signal;

Preduslovi: *Ako je racun obradjen ili storniran ne moze se izvrsiti sistemska operacija.*

Postuslovi:

- *Izračunata je vrednost svake od stavki računa.*
- *Izračunata je ukupna vrednost računa.*
- *Račun je obrađen.*

```
class Obradi
 {
 public static int Obradi(Racun rac)
  {   Obradi ob = new Obradi();
      return ob.IzvrsenjeSO(rac);
  }

// Prekrivanje metode klase OpstaSO
int izvrsenjeSO(Racun rac)
 { Racun rac1 = new Racun(rac);
   if (!nadjiRacuniVratiGa(rac1)) return 0;
  if (!Preduslov(rac1)) return 0;
  if (!racunajUkupneVrednosti(rac)) return 0;
  if (!obradiRacun(rac)) return 0;
  if (!azurirajRacun(rac1,rac)) return 0;
  if (!brisiStavkeRacuna(rac1)) return 0;
 dodavanjeStavkiRacuna(rac);return 0;
 }

private boolean nadjiRacuniVratiGa(Racun rac)
  { // Isto kao kod SO Računaj
  }
private boolean Preduslov(Racun rac1) // 94,77
    { // Isto kao kod SO Računaj
     }
private boolean racunajUkupneVrednosti(Racun rac) // 79,78
  {  // Isto kao kod SO Računaj }

private boolean obradiRacun(Racun rac) // 57,58
  { try{  rac.Obradjen = true;} catch(Exception e) { signal = 58; return false;}
    signal = 57;     return true;  }

private boolean azurirajRacun (Racun rac1,Racun rac)
     {  // Isto kao kod SO Računaj      }
```

```
private boolean zapamtiRacun(Racun rac)
   {  // Isto kao kod SO Računaj    }

private boolean promeniRacun(Racun rac)
   {  // Isto kao kod SO Računaj   }

private boolean brisiStavkeRacuna(Racun rac) // 33,34
 { // Isto kao kod SO Računaj}

private boolean dodavanjeStavkiRacuna(Racun rac) //31,32
{  // Isto kao kod SO Računaj}
// Prekrivanje metode klase OpstaSO
    boolean stanjeOperacije(int signal)
    { switch (signal)
        {    case 31: Prikazi(signal,"Uspesno zapamcen slog");return true;
             case 32: Prikazi(signal,"Bezuspesno zapamcen slog");return false;
           case 33: Prikazi(signal,"Slogovi uspesno obrisani");return true;
           case 34: Prikazi(signal,"Slogovi bezuspesno obrisani");return false;
           case 35: Prikazi(signal,"Uspesno promenjen slog");return true;
           case 36: Prikazi(signal,"Bezuspesno promenjen slog");return false;
           case 57: Prikazi(signal,"Uspesno obradjen racun");return true;
           case 58: Prikazi(signal,"Neuspesno obradjen racun");return false;
           case 71: Prikazi(signal,"Uspesno pretrazivanje slogova");return true;
           case 72: Prikazi(signal,"Neuspesno pretrazivanje slogova");return false;
           case 77: Prikazi(signal,"Preduslov je zadovoljen");return true;
           case 78: Prikazi(signal,"Neuspesno izracunata ukupna vrednost");return false;
           case 79: Prikazi(signal,"Uspesno izracunata ukupna vrednost");return true;
           case 94: Prikazi(signal,"Ne moze da radi sa racunom koji je obradjen ili storniran");return false;
         }
     return false;
  }
}
```

Iz navedenog se može videti da je SO *Obradi* veoma sličan SO *Računaj*. U programskom kodu smo boldovali i podvukli ono što ima SO *Obradi* a što nema SO *Računaj*.

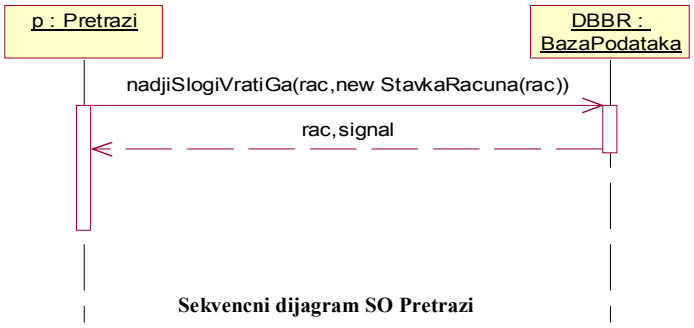
UGOVOR UG5: ***Pretraži***

Operacija: Pretraži (*Racun*):signal;

Preduslovi:

Postuslovi: *Pročitan je račun ukoliko postoji.*

```
class Pretrazi
 { public static int Pretrazi(Racun rac)
   {    Pretrazi p = new Pretrazi();
        return p.IzvrsenjeSO(rac);
   }

int izvrsenjeSO(Racun rac)
    { signal = DBBR.nadjiSlogiVratiGa(rac,new StavkaRacuna(rac)); // 71,72
       if (!stanjeOperacije(signal)) return false;
      return true;
   }

 boolean stanjeOperacije(int signal)
   {  switch (signal)
      { case 71: Prikazi(signal,"Uspesno je procitan slog iz baze (uspesno pretrazivanje)");return true;
        case 72: Prikazi(signal,"Neuspesno je procitan slog iz baze (neuspesno pretrazivanje)");return false;
      }
     return false;
   }
 }
```



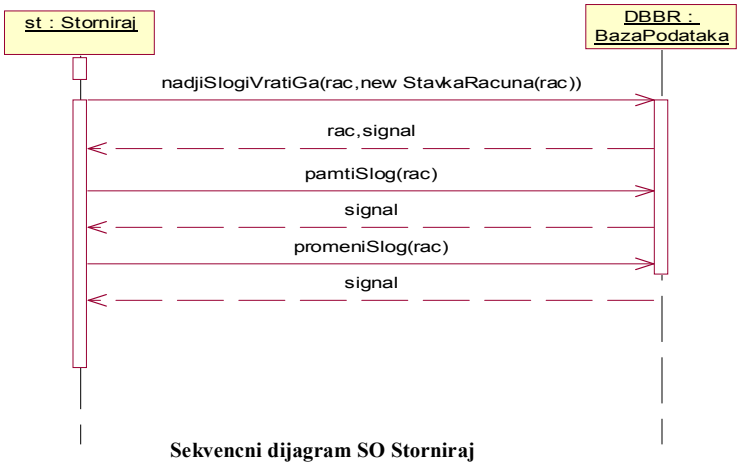
Sekvencni dijagram SO Pretrazi

UGOVOR UG6: ***Storniraj***

Operacija: Storniraj(*Racun*):signal;

Preduslovi: *Ako je racun storniran ne moze se izvrsiti sistemska operacija.*
Postuslovi: *Račun je storniran*

```
class Storniraj
 { public static int Storniraj(Racun rac)
    {  Storniraj st = new Storniraj();  return st.IzvrsenjeSO(rac);  }

int izvrsenjeSO(Racun rac)
  {  Racun rac1 = new Racun(rac);
     if (!nadjiRacuniVratiGa(rac1)) return 0;
     if (!Preduslov(rac1))     return 0;
     if (!stornirajRacun(rac)) return 0;
     azurirajRacun(rac1,rac);  return 0;
}
 private boolean Preduslov(Racun rac1)
  {  // 1. Ako je racun storniran nad njim se ne moze izvrsiti novo storniranje.
     if (rac1.Storniran == true)
       {   signal = 93; return false;}
           }
    signal = 77; // Zadovoljen preduslov
   return true;
 }
 private boolean stornirajRacun(Racun rac) // 157,158
   { try{  rac.Storniran = true;} catch(Exception e) { signal = 158; return false;}
     signal = 157;
     return true;
  }

private boolean nadjiRacuniVratiGa(Racun rac)
  { // Isto kao kod SO Računaj  }

private boolean azurirajRacun (Racun rac1,Racun rac)
     { // Isto kao kod SO Računaj }

private boolean zapamtiRacun(Racun rac)
  {  // Isto kao kod SO Računaj   }

private boolean promeniRacun(Racun rac)
  {  // Isto kao kod SO Računaj   }

boolean stanjeOperacije(int signal)
  {  switch (signal)
      {   case 31: Prikazi(signal,"Uspesno zapamcen slog");return true;
          case 32: Prikazi(signal,"Bezuspesno zapamcen slog");return false;
         case 35: Prikazi(signal,"Uspesno promenjen slog");return true;
         case 36: Prikazi(signal,"Bezuspesno promenjen slog");return false;
         case 71: Prikazi(signal,"Uspesno osvezavanje sloga kod rollback-a");return true;
         case 72: Prikazi(signal,"Neuspesno osvezavanje sloga kod rollback-a");return false;
         case 77: Prikazi(signal,"Preduslov je zadovoljen");return true;
         case 93: Prikazi(signal,"Ne moze da radi sa racunom koji je storniran");return false;
         case 157: Prikazi(signal,"Uspesno storniran racun");return true;
         case 158: Prikazi(signal,"Neuspesno storniran racun");return false;
      }
    return false;
 }
}
```

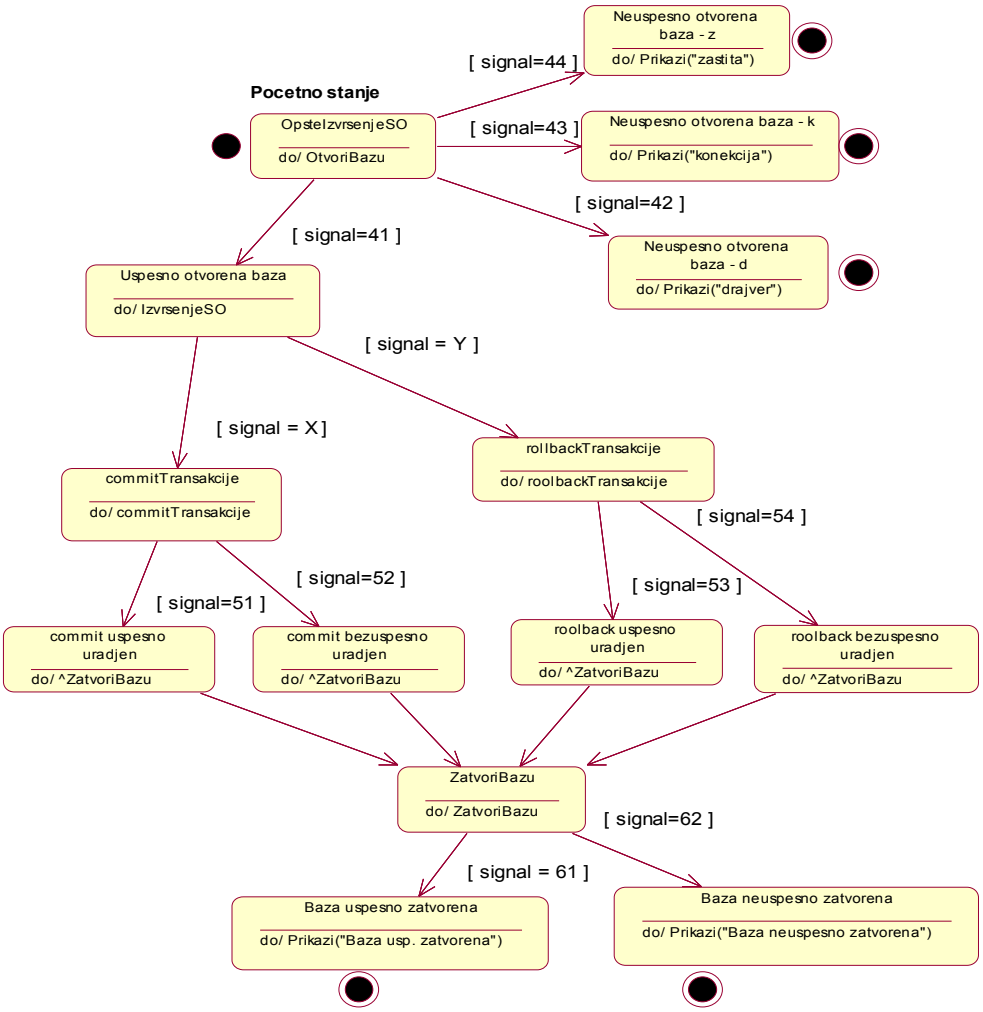


Sekvencni dijagram SO Storniraj

Nakon projektovanja svake od SO prelazi se na projektovanje klase koja je odgovorna za konekciju sa bazom i za kontrolu izvršenja transakcije. Metoda koja obezbeđuje navedene zahteve se zove *opsteIzvrsenjeSO()*. Pored objektnog koda ista je predstavljena preko dijagrama prelaza stanja(Slika DPSI).

```
class OpstaSO
{
 static DatabaseBroker DBBR;
 static int signal;
 static boolean BazaOtvorena = false;

 int opsteIzvrsenjeSO(Racun rac)
  {  signal = 0; // Pocetno stanje
     if (!otvoriBazu()) return signal;
     izvrsenjeSO(rac);
     proveraUspesnostiTransakcije(rac);
    zatvoriBazu();
    return signal;
  }

int izvrsenjeSO(Racun rac){//Ova metoda se implementira u okviru klasa SO
              return 0;}

boolean otvoriBazu()
  {     if (BazaOtvorena == false)
          { DBBR = new DatabaseBroker();
            signal = DBBR.otvoriBazu("RACUN"); // 41,42,43,44
            if (!stanjeOperacijeOpstaSO(signal)) return false;
          }
    BazaOtvorena = true;
    return true;
  }

 boolean zatvoriBazu()
    {  int signal1 = DBBR.zatvoriBazu(); // 61,62
       if (!stanjeOperacijeOpstaSO(signal1))   { signal = signal1;return false;}
       BazaOtvorena = false;
      return true;
    }

 boolean proveraUspesnostiTransakcije(Racun rac)
   { int signal1;
     if (stanjeOperacije(signal))
        { signal1 = DBBR.commitTransakcije(); // 51,52
          if (!stanjeOperacijeOpstaSO(signal1))
           {signal=signal1;
            return false;
           }
        }
  else
        { signal1 = DBBR.rollbackTransakcije(); // 53,54
          if (!stanjeOperacijeOpstaSO(signal1))
            { signal = signal1;
              return false;
            }
        }
   return true;
    }

 boolean stanjeOperacije(int signal){//Ova metoda se implementira u okviru klasa SO
                          return false;}
 boolean stanjeOperacijeOpstaSO(int signal)
  {   switch (signal)
              {   case 41: Prikazi(signal,"Uspesno otvorena baza"); return true;
                  case 42: Prikazi(signal,"Neuspesno otvorena baza - drajver");return false;
                  case 43: Prikazi(signal,"Neuspesno otvorena baza - konekcija");return false;
                 case 44: Prikazi(signal,"Neuspesno otvorena baza - zastita");return false;
                  case 51: Prikazi(signal,"Commit uspesno uradjen");return true;
                 case 52: Prikazi(signal,"Commit neuspesno uradjen");return false;
                 case 53: Prikazi(signal,"Rollback uspesno uradjen");return true;
                 case 54: Prikazi(signal,"Rollback neuspesno uradjen");return false;
                 case 61: Prikazi(signal,"Baza uspesno zatvorena");return true;
                 case 62: Prikazi(signal,"Baza neuspesno zatvorena");return false;
              }
    return false;
```

```
  }
 void Prikazi(int signal,String stanje)
  { System.out.println("SIGNAL: " + signal + ". Stanje: " + stanje);}

}
```



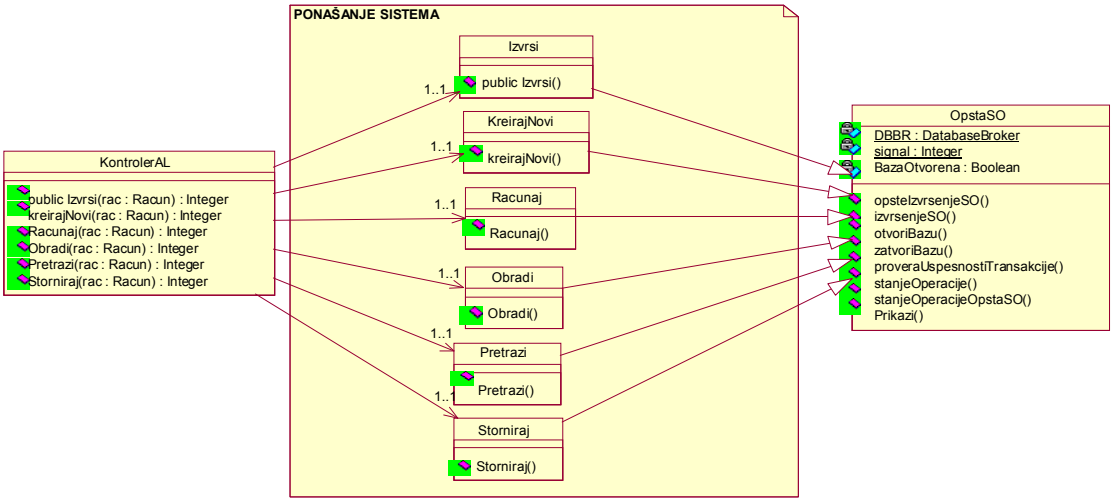
Slika DPSI: Dijagram prelaza stanja metode *opšteIzvršenjeSO()*

Svaka od SO treba da nasledi klasu *OpstaSO* kako bi mogla da se poveže sa bazom i kako bi se njeno izvršenje pratilo kao transakcija:

```
class Izvrsi extends OpstaSO
{

 public static int Izvrsi(Racun rac)
  {
      Izvrsi iz = new Izvrsi();

     return iz.opsteIzvrsenjeSO(rac);   // umeso return iz.IzvrsenjeSO(rac);
  }

...
}
```

Na sličan način se radi i sa ostalim klasama koje su odgovorne za izvršenje SO. Boldovali smo ono što se dodaje kod svake od klasa.

Klase koje su odgovorne za SO nasleđuju klasu OpstaSO (Slika OSO).



Slika OSO: Klase koje su odgovorne za SO nasleđuju klasu OpstaSO

1.3.5 PROJEKTOVANJE APLIKACIONE LOGIKE – DATABASE BROKER

Pre projektovanja database brokera navodi se njegova definicija i definicija svih onih koncepata koji su potrebni da bi se jasno shvatila komunikacija između database brokera i baze podataka.

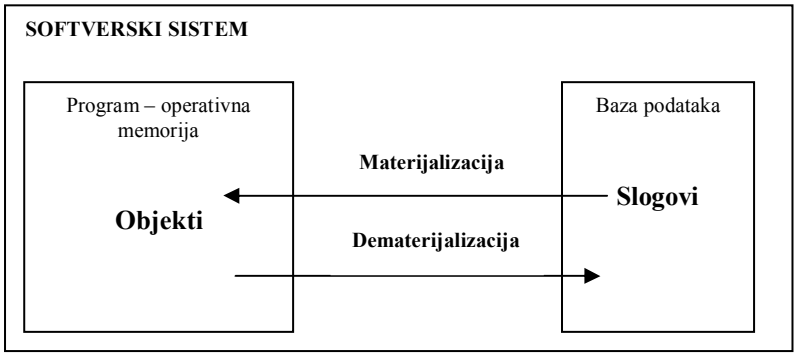
Definicije aplikacione logike – database broker

Def PRPO1(G. Booch): Objekat je **perzistentan** ukoliko nastavi da postoji i nakon prestanka rada programa koji ga je stvorio.

Def PRPO2: Objekat je **perzistentan** ukoliko se može **materijalizovati** i **dematerijalizovati** (Slika MD).

Def PRPO3: Materijalizacija predstavlja proces transformacije slogova iz baze podataka u objekte programa[19].

Def PRPO4: Dematerijalizacija (**Pasivizacija**) predstavlja proces transformacije objekata iz programa u slogove baze podataka.



Slika MD: Materijalizacija i dematerijalizacija objekta

Def PRPO5: Perzistentni okvir je skup interfejsa i klasa koji omogućava perzistentnost objektima različitih klasa (Perzistentni okvir omogućava **perzistentni servis**[20] objektima). On se može proširiti sa novim interfejsima i klasama.

Def PRPO6: Perzistentni okviri su zasnovani na **Holivudskom principu**: *"Don't call us, we'll call you"*. To znači da korisnički definisane klase prihvataju poruke od predefinisanih klasa okvira.

Def PRPO7: Ukoliko u okviru neke **transakcije**, operacije menjaju stanje perzistentnih objekata, one ne čuvaju to stanje odmah u bazi podataka. Ukoliko se želi zapamtiti stanje koje je nastalo kao rezultat izvršenih operacija nad perzistentnim objektima poziva se **commit operacija**. Ukoliko se ne želi zapamtiti stanje koje je nastalo kao rezultat izvršenih operacija nad perzistentnim objektima poziva se **roolback operacija**. Commit i rollback operacije predstavljaju **transakcione operacije**.

Primer Database brokera

U našem primeru mi smo projektovali perzistentni okvir (klasa DatabaseBroker[21]) koji će da realizuje sledeće metode:

1. *int otvoriBazu()*
2. *int zatvoriBazu()*
3. *int commitTransakcije()*
4. *int rollbackTransakcije()*
5. *int pamtiSlog(Objekat)*

[19] Termine materijalizacija i dematerijalizacija smo objasnili u užem smislu kao procese koji transformišu objekte programa u slogove baza podataka. U širem smislu bi se podrazumevalo da objekti mogu biti sačuvani u bilo kom perzistentnom skladištu podataka.

[20] Ukoliko **perzistentni servis** omogućava pamćenje objekata u relacionoj bazi podataka za njega se kaže da je on **Object-Relation servis preslikavanja (mapping service).**

[21] Database Broker [Larman] patern predstavlja jednu moguću realizaciju perzistentnog okvira. Database Broker [Larman] patern je odgovoran za materijalizaciju, dematerijalizaciju i keširanje objekata u memoriji. On se često naziva i **Database Mapper** patern.

6. *int brisiSlog(Objekat)*
7. *int brisiSlogove(Objekat)*
8. *int promeniSlog(Objekat)*
9. *int daDaLiPostojiSlog(Objekat)*
10. *int nadjiSlogiVratiGa(Objekat,Objekat)*
11. *int vratiZadnjiSlog(Objekat,Objekat)*
12. *int vratiBrojZadnjegSloga(Objekat)*
13. *int kreirajSlog(Objekat)*

Dajemo detaljno objašnjenje svake od navedenih metoda.

```
class DatabaseBroker
{
  static Connection con;
  static Statement st;
```

/*Ugovor DB1: **otvoriBazu**(String imeBaze) : integer
Postuslov: Uspostavljena je veza (konekcija) sa bazom podataka. Ukoliko je uspešno ostvarena veza sa bazom podataka metoda vraća vrednost 41, inače metoda vraća vrednost:

- 42 ako drajver nije učitan.
- 43 ako se desila greška kod konekcije
- 44 ako se desila greška kod zaštite baze podataka. */

```
public int otvoriBazu(String imeBaze)
   {  String Urlbaze;
      try {   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); // Učitavanje JdbcOdbc drajvera
              Urlbaze = "jdbc:odbc:" + imeBaze; // Određuje se adresa gde se nalazi baza
              con = DriverManager.getConnection(Urlbaze);  // Uspostavljanje veze sa bazom podataka.
              con.setAutoCommit(false); // Omogucava se roolback transakcije.
         } catch(ClassNotFoundException e) { System.out.println("Drajver nije ucitan:" + e);return 42;}
           catch(SQLException esql) { System.out.println("Greska kod konekcije:" + esql);return 43;}
           catch(SecurityException ese) {System.out.println("Greska zastite:" + ese);return 44;}
     return 41;
  }
```

/*Ugovor DB2: **zatvoriBazu**() : integer
Postuslov: Prekinuta je veza (konekciju) sa bazom podataka. Ukoliko je uspešno prekinuta veza sa bazom podataka metoda vraća vrednost 61, inače vraća vrednost 62. */

```
public int zatvoriBazu()  { try {  con.close();} catch(Exception e) {System.out.println(e);return 62;} return 61; }
```

/*Ugovor DB3: **commitTransakcije**() : integer
Postuslov: Sve operacije koje su menjale stanje baze, od zadnjeg poziva commit ili rollback transakcije, su uspešno izvršene (promene su uspešno zapamćene u bazi). U tom slučaju metoda vraća vrednost 51, inače vraća vrednost 52 ako commit nije uspešno izvršen. */

```
int commitTransakcije()
   {   try{ con.commit();
   }   catch(SQLException esql)  {  System.out.println("Nije uspesno uradjen commit. " + esql); return 52;}
     return 51;
   }
```

/*Ugovor DB4: **rollbackTransakcije**() : integer
Postuslov: Efekti svih operacija koje su menjale stanje baze, od zadnjeg poziva commit ili rollback transakcije, su poništeni (promene nisu zapamćene u bazi). U tom slučaju metoda vraća vrednost 53, inače vraća vrednost 54 ako rollback nije uspešno izvršen. */

```
  int rollbackTransakcije()
   {   try{ con.rollback();
         }  catch(SQLException esql)  { System.out.println("Nije uspesno uradjen rollback." +esql); return 54; }
     return 53;
   }
```

/*Ugovor DB5: **pamtiSlog(OpstiDomenskiObjekat)**: integer
Postuslov: Izvršeno je pamćenje objekta u bazu podataka (materijalizacija). Ukoliko je metoda uspešno izvršena vraća vrednost 31, inače vraća vrednost 32.
Napomena: Navedena metoda je genericka jer omogucava pamćenje objekata bilo koje klase u bazi, ukoliko te klase nasleđuje klasu *OpstiDomenskiObjekat.*

```
public int pamtiSlog(OpstiDomenskiObjekat odo)
  { String upit;
    try{ st = con.createStatement();
         upit ="INSERT INTO " + odo.vratiImeKlase()  +  " VALUES (" + odo.vratiVrednostiAtributa() + ")";
         st.executeUpdate(upit);
         st.close();
       }  catch(SQLException esql)
            {  System.out.println("Nije uspesno zapamcen slog u bazi: " + esql); return 32;  }
    return 31;
  }
```

/**

OBJAŠNJENJE POSTUPKA PROJEKTOVANJA GENERIČKE METODE
Ukoliko bi metoda bila specifično vezana za konkretnu domensku klasu npr. Račun ona bi imala sledeći izgled (boldovali smo zavisne delove metode od domenske klase) :

```
public int pamtiSlog(Racun rac)
  { String upit;
    try{ st = con.createStatement();
         upit ="INSERT INTO  " + rac.vratiImeKlaseRacun()  +  " VALUES (" +
                                 rac.vratiVrednostiAtributaRacuna() + ")";
         st.executeUpdate(upit);
         st.close();
       }  catch(SQLException esql)
            {  System.out.println("Nije uspesno zapamcen slog u bazi: " + esql); return 32;  }
    return 31;
  }

class Racun
{ ...
public String vratiVrednostiAtributaRacuna()
 { return ""+ BrojRacuna + "', '" + NazivPartnera + "', " +  UkupnaVrednost.doubleValue() + ", " + Obradjen + ", " +
   Storniran;}
public String vratiImeKlaseRacun() { return "Racun";}
}
```

Ukoliko bi koristili navedeni pristup morali bi za svaku domensku klasu da implementiramo metodu *pamtiSLog(),* kao i sve ostale metode koje ćemo niže navesti (*brisiSlog(),promeniSlog(),...)* u klasi DatabaseBroker. To bi bilo prilično nepraktično jer bi broj operacija klase DataBaseBroker rastao sa pojavom novih domenskih klasa. Zbog toga smo svaku od niže navedenih metoda projektovali kao generičku metodu. Na primeru metode *pamtiSlog()* objasnićemo opšti postupak za projektovanje bilo koje od niže navedenih generičkih metoda.

Postupak projektovanja generičke metode:
1. Odrediti specifične klase *(Racun)* u metodi (*PamtiSlog*) koja treba da postane generalna. Imenovati specifične klase u opštem smislu (npr. klasu Račun sa klasom OpstiDomenskiObjekat). Takođe metode specifičnih klasa imenovati u opštem smislu, ako je njihov naziv povezan sa nečim što je specifično za klasu kojoj one pripadaju (npr. naziv metode *vratiImeKlaseRacun()* sa nazivom *VratiImeKlase()).*

```
public int pamtiSlog(OpstiDomenskiObjekat odo)
  { String upit;
    try{ st = con.createStatement();
         upit ="INSERT INTO" + odo.vratiImeKlase() + "  VALUES (" + odo.vratiVrednostiAtributa() + ")";
         st.executeUpdate(upit);
         st.close();
       }  catch(SQLException esql)
            {  System.out.println("Nije uspesno zapamcen slog u bazi: " + esql); return 32;  }
    return 31;
  }
```

2. Napraviti generalnu klasu (apstraktnu klasu ili interfejs) za specifične klase.

```
interface OpstiDomenskiObjekat
{ vratiVrednostiAtributa();
  vratiImeKlase();}
```

Iz navedenog može da se zaključi da metoda *pamtiSlog(),* može da prihvati različite domenske objekte preko parametra, ako domenski objekti naslede klasu *OpstiDomenskiObjekat* i implementiraju njene metode *vratiIzraz() i vratiImeKlase().*

U tom smislu domenska klasa Racun i StavkaRačuna će dobiti sledeći izgled:

```
class Racun implements OpstiDomenskiObjekat
{...
public String vratiVrednostiAtributa()
  { return ""+ BrojRacuna + "', '" + NazivPartnera + "', " +  UkupnaVrednost.doubleValue() + ", " + Obradjen + ", " +
    Storniran;}
public String vratiImeKlase()
  { return "Racun";}
}

class StavkaRacuna implements OpstiDomenskiObjekat
{
 public String vratiVrednostiAtributa()
  { return "'" + rac.BrojRacuna + "' , " + RB.intValue() + ", '" + SifraProizvoda + "', " +  Kolicina.intValue() + ", " +
ProdajnaCena.doubleValue() + ", " + ProdajnaVrednost.doubleValue();}

public String vratiImeKlase() { return "StavkaRacuna";}
}
```

Kod domenskih klasa smo boldovali sve delove programa koji su promenjeni kako bi oni mogli da budu prihvaćeni kao parametar metode *pamtiSlog().*

KRAJ OBJAŠNJENJA POSTUPKA PROJEKTOVANJA GENERIČKE METODE

/**

/*Ugovor DB6: **brisiSlog**(OpstiDomenskiObjekat odo) : integer
Postuslov: Obrisan je objekat (slog) u bazi podataka (takav objekat se ne može više materijalizovati). Ukoliko je metoda uspešno izvršena vraća vrednost 33, inače vraća vrednost 34.
Napomena: Navedena metoda je genericka jer omogucava da se preko nje moze obrisati objekat bilo koje klase koja nasleđuje klasu *OpstiDomenskiObjekat.*
/*

```
public int brisiSlog(OpstiDomenskiObjekat odo)
    { String upit;
      try {   st = con.createStatement();
            upit ="DELETE *  FROM " + odo.vratiImeKlase() + " WHERE " + odo.vratiUslovZaNadjiSlog();
            st.executeUpdate(upit);
            st.close();
        }   catch(SQLException esql) { System.out.println("Nije uspesno obrisan slog u bazi: " + esql);  return 34; }
       return 33;
    }
```

/*Ugovor DB7: **brisiSlogove**(OpstiDomenskiObjekat odo) : integer
Postuslov: Obrisani su objekti (slogovi) u bazi podataka po zadatom uslovu. Ukoliko je metoda uspešno izvršena vraća vrednost 33, inače vraća vrednost 34.
Napomena: Navedena metoda je genericka jer omogucava da se preko nje moze obrisati više objekata, po zadatom uslovu, bilo koje klase koja nasleđuje klasu *OpstiDomenskiObjekat.*
/*

```
public int brisiSlogove(OpstiDomenskiObjekat odo)
    { String upit;
      try {    st = con.createStatement();
             upit ="DELETE *  FROM " + odo.vratiImeKlase() + " WHERE " + odo.vratiUslovZaNadjiSlogove();
          st.executeUpdate(upit);
         st.close();
         }   catch(SQLException esql) { System.out.println("Nije uspesno obrisan slog u bazi: " + esql);return 34;  }
       return 33;
    }
```

/*Ugovor DB8: **promeniSlog**(OpstiDomenskiObjekat odo) : integer
Postuslov: Promenjen je objekat u bazi podataka po zadatom uslovu. Ukoliko je metoda uspešno izvršena vraća vrednost 35, inače vraća vrednost 36.
Napomena: Navedena metoda je genericka jer omogucava promenu objekta bilo koje klase koja nasleđuje klasu *OpstiDomenskiObjekat.*

```
/*
  public int promeniSlog(OpstiDomenskiObjekat odo)
    { String upit;
      try {  st = con.createStatement();
            upit ="UPDATE " + odo.vratiImeKlase() +  " SET " + odo.postaviVrednostiAtributa() +
                  " WHERE " + odo.vratiUslovZaNadjiSlog();
             st.executeUpdate(upit);
             st.close();
         }   catch(SQLException esql) { System.out.println("Nije uspesno promenjen slog u bazu: " + esql); return 36;
}
      return 35;
    }
```

/*Ugovor DB9: **daLiPostojiSlog**(OpstiDomenskiObjekat odo) : integer
Postuslov: Ukoliko postoji objekat u bazi metoda vraća 37 inače vraća 38. Ispituje da li postoji objekat - ne vraća ga.
Napomena: Navedena metoda je genericka jer omogucava pretraživanje objekta bilo koje klase koja nasleđuje klasu *OpstiDomenskiObjekat*.

```
/*
  public int daLiPostojiSlog(OpstiDomenskiObjekat odo)
    { String upit;
      ResultSet RSslogovi;
      try { st = con.createStatement();
            upit ="SELECT *" +  " FROM " + odo.vratiImeKlase() +
              " WHERE " + odo.vratiUslovZaNadjiSlog();
           RSslogovi = st.executeQuery(upit);
           boolean signal = RSslogovi.next();
          RSslogovi.close();
          st.close();
         if (signal == false)  return 38; // Slog ne postoji u bazi.
          }   catch(SQLException esql) {  System.out.println("Nije uspesno pretrazena baza: " + esql); return 39;  }
      return 37; // Slog postoji u bazi.
    }
```

/*Ugovor DB10: **nadjiSlogiVratiGa**(OpstiDomenskiObjekat odo, OpstiDomenskiObjekat odo1) : integer
Postuslov: Pročitan je objekat (odo) sa njegovim podredjenim objektima (odo1). Ukoliko je metoda uspešno izvršena vraća 71 inače vraća 72. Ovo je materijalizacija objekta.
Napomena: Navedena metoda je genericka jer omogucava čitanje objekta i njegovih podređenih objekata bilo koje klase koja nasleđuje klasu *OpstiDomenskiObjekat*.

```
/*
  public  int nadjiSlogiVratiGa(OpstiDomenskiObjekat odo, OpstiDomenskiObjekat odo1 )
    { ResultSet RSslogovi;
      ResultSet RSstavkeSlogova;
      String upit,upit1;
     Statement st,st1;
       try                    {                                                                    st                  =
con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,ResultSet.CONCUR_READ_ONLY);
            st1                                                                                                      =
con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,ResultSet.CONCUR_READ_ONLY);
            upit ="SELECT *" +  " FROM " + odo.vratiImeKlase() +  " WHERE " + odo.vratiUslovZaNadjiSlog();
            upit1   ="SELECT   *"   +     "   FROM   "   +   odo1.vratiImeKlase()   +     "   WHERE   "   +
odo1.vratiUslovZaNadjiSlogove();
            RSslogovi = odo.executeQuery(upit);
            RSstavkeSlogova = odo1.executeQuery(upit1);
            if(!odo.Napuni(RSslogovi,RSstavkeSlogova))
                return 72;
           RSslogovi.close();
           RSstavkeSlogova.close();
           st.close();
           st1.close();
        }   catch(Exception e) { System.out.println("Greska kod citanja sloga iz baze" + e); return 72; }
     return 71; }
```

/*Ugovor DB11: **vratiZadnjiSlog**(OpstiDomenskiObjekat odo, OpstiDomenskiObjekat odo1) : integer
Postuslov: Pročitan je zadnji uneti objekat i njegovi podređeni objekti .Ukoliko je metoda uspešno izvršena vraća 75 inače vraća 76. Ovo je materijalizacija objekta.
Napomena: Navedena metoda je genericka jer čita zadnji objekat i njegove podređene objekte bilo koje klase koja nasleđuje klasu *OpstiDomenskiObjekat*.

```
/*
    int vratiZadnjiSlog(OpstiDomenskiObjekat odo, OpstiDomenskiObjekat odo1)
     {   int signal=0;
         String upit;
         ResultSet RSslogovi;
         try { st = con.createStatement();
```

```
            upit ="SELECT Max(" + odo.vratiAtributPretrazivanja()+ " )AS Max" +  " FROM " + odo.vratiImeKlase();
            RSslogovi = st.executeQuery(upit);
            if (!odo.Napuni(RSslogovi)) return 76;
            if (nadjiSlogiVratiGa(odo,odo1)!=71) return 76;
            RSslogovi.close();
            st.close();
           } catch(Exception e) {  System.out.println("Greska kod citanja zadnjeg unetog sloga u bazu" + e);  return
                                    76; }
      return 75;
    }
```

/*Ugovor DB12: **vratiBrojZadnjegSloga**(OpstiDomenskiObjekat odo) : integer
Postuslov: Pročitan je broj zadnjeg unetog objekta .Ukoliko je metoda uspešno izvršena vraća 171 inače vraća 173. Napomena: Navedena metoda je genericka jer čita broj zadnje unetog objekta bilo koje klase koja nasleđuje klasu *OpstiDomenskiObjekat*.

```
/*
int vratiBrojZadnjegSloga(OpstiDomenskiObjekat odo)
     {   int signal=0;
         String upit;
        ResultSet RSslogovi;
        try {  st = con.createStatement();
              upit ="SELECT Max(" + odo.vratiAtributPretrazivanja()+ " )AS Max" +
                    " FROM " + odo.vratiImeKlase();
             RSslogovi = st.executeQuery(upit);
            if(!odo.Napuni(RSslogovi)) { return 173;}
            RSslogovi.close();
            st.close();
          } catch(Exception e) { System.out.println("Greska kod citanja zadnjeg unetog sloga u bazu" + e);  return 173;
                                 }

      return 171;
  }
```

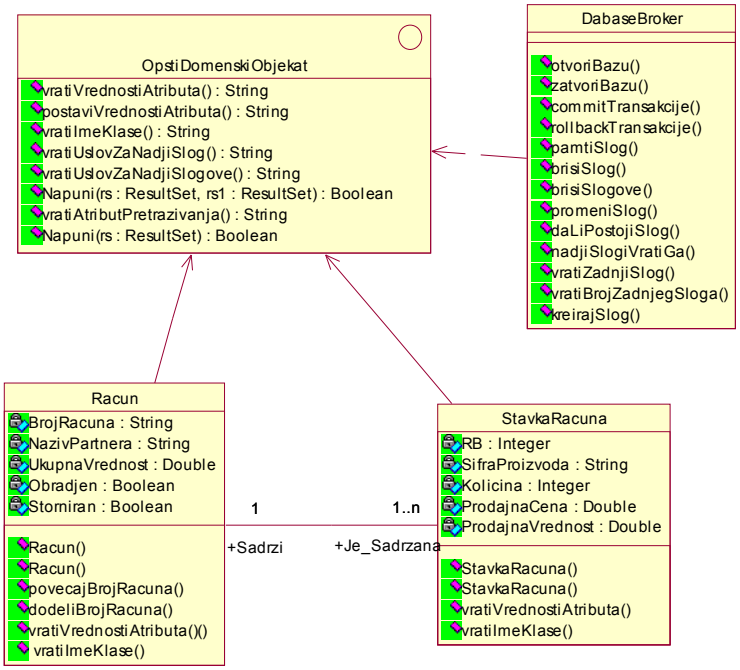
/*Ugovor DB13: **kreirajSlog**(OpstiDomenskiObjekat odo) : integer
Postuslov: Kreiran je novi objekat.Ukoliko je metoda uspešno izvršena vraća 45 inače vraća 46.
Napomena: Navedena metoda je genericka jer kreira objekat bilo koje klase koja nasleđuje klasu *OpstiDomenskiObjekat*.

```
/*
    int kreirajSlog(OpstiDomenskiObjekat odo)
      { String upit;
        try{  st = con.createStatement();
              upit ="INSERT INTO " + odo.vratiImeKlase()  +  " VALUES (" + odo.vratiVrednostiAtributa() + ")";
              st.executeUpdate(upit);
             st.close();
           }   catch(SQLException esql)  {  System.out.println("Nije uspesno kreiran slog u bazi: " + esql);return 46; }
      return 45;
    }

 }
```

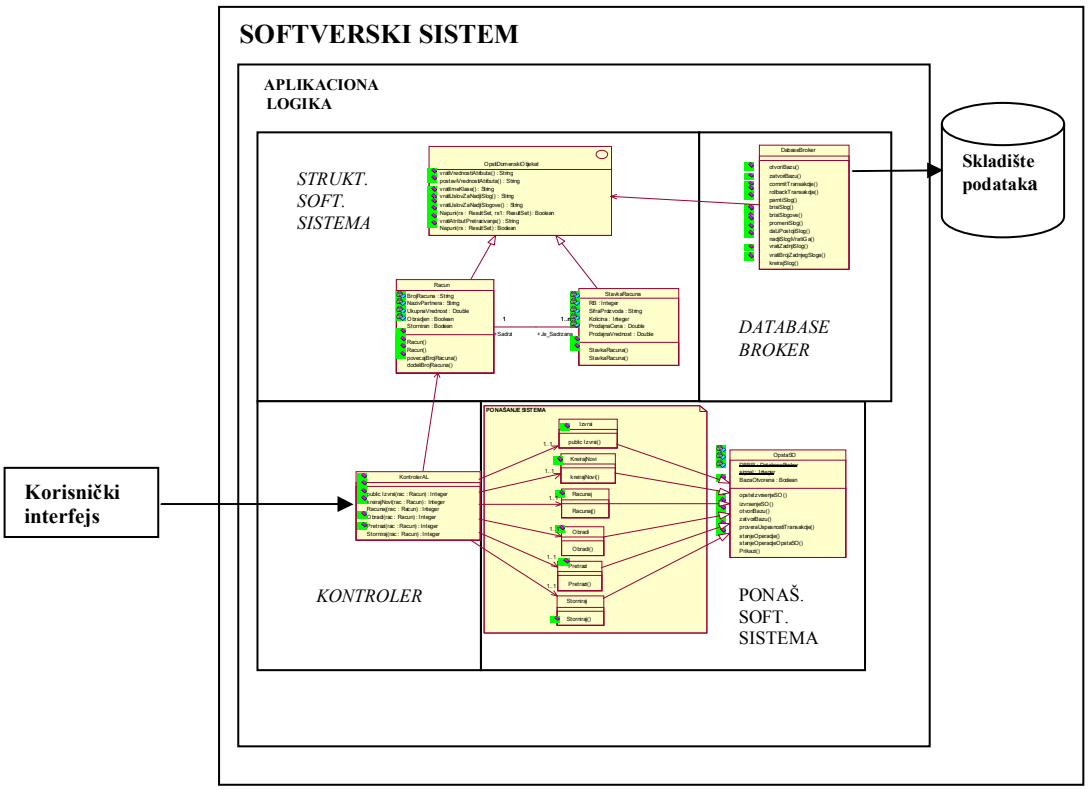
U procesu pravljenja generičkih metoda DatabaseBroker klase dobili smo metode interfejsa OpstiDomenskiObjekat:

```
// Operacije navedenog interfejsa je potrebno da implementira svaka od domenskih klasa,
// koja zeli da joj bude omogucena komunikacija sa Database broker klasom.

interface OpstiDomenskiObjekat
{ String vratiVrednostiAtributa();
 String postaviVrednostiAtributa();
 String vratiImeKlase();
 String vratiUslovZaNadjiSlog();
 String vratiUslovZaNadjiSlogove();
 boolean Napuni(ResultSet rs, ResultSet rs1);
 String vratiAtributPretrazivanja();
 boolean Napuni(ResultSet rs);
}
```

Kao rezultat projektovanja klase DatabaseBrokera i interfejsa OpstiDomenskiObjekat dobijaki se sledeći dijagrami klasa (Slika DBBR, ASSDBBR)



Slika DBBR: Database broker klasa se povezuje sa klasom OpstiDomenskiObjekat



Slika ASSDBBR: Arhitektura soft. sistema nakon projektovanja *DatabaseBroker* klase

Napomena: Povezivanje Database brokera sa softverskim klasama strukture je očigledan primer uzora (paterna), koji je sličan uzorima koje je dao Gamma [GOF].

1.3.6 PROJEKTOVANJE SKLADIŠTA PODATAKA

Na osnovu softverskih klasa strukture projektovali smo tabele (skladišta podataka) relacionog sistema za upravljanje bazom podataka (Slika ASSTBP).

Table: Racun

Columns

Name	Type	Size
BrojRacuna	Text	10
NazivPartnera	Text	50
UkupnaVrednost	Number (Double)	8
Obradjen	Yes/No	1
Storniran	Yes/No	1

Table Indexes

Name	Number of Fields
PrimaryKey	1
Fields:	BrojRacuna, Ascending

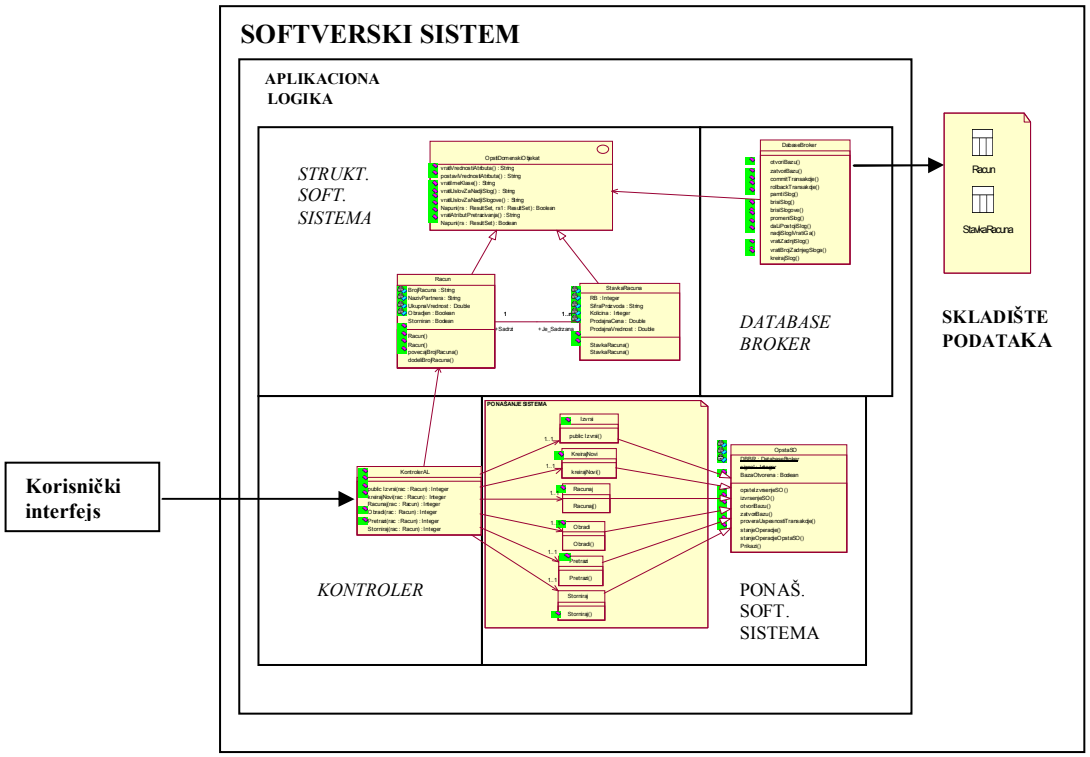
Table: StavkaRacuna

Columns

Name	Type	Size
BrojRacuna	Text	10
RB	Number (Integer)	2
SifraProizvoda	Text	20
Kolicina	Number (Integer)	2
ProdajnaCena	Number (Double)	8
ProdajnaVrednost	Number (Double)	8

Table Indexes

Name	Number of Fields
PrimaryKey	2
Fields:	BrojRacuna, Ascending RB, Ascending



Slika ASSTBP: Arhitektura soft. sistema nakon projektovanja tabela baze podataka

1.3.7 STRUKTURA KORISNIČKOG INTERFEJSA

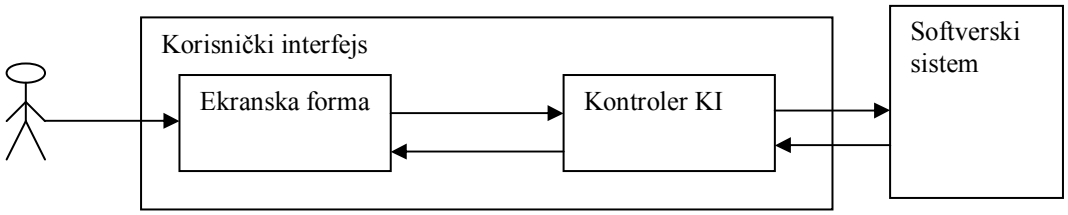
Korisnički interfejs shodno *Def. Laz1*, predstavlja realizaciju ulaza i/ili izlaza softverskog sistema. Pre projektovanja korisničkog interfejsa objasnićemo delove korisničkog interfejsa (njegovu strukturu), način njihove međusobne komunikacije i način komunikacije korisničkog interfejsa sa softverskim sistemom (Slika SKI).

Korisnički interfejs se sastoji od:

- Ekranske forme koja je odgovorna da:
    a) prihvata podatke koje unosi aktor,
    b) prihvata događaje koje pravi aktor,
    c) poziva kontrolera grafičkog interfejsa, prosleđujući mu prihvaćene podatke
    d) prikazuje podatke koje je dobio od kontrolera grafičkog interfejsa.

Svaku od navedenih odgovornosti ekranska forma obavlja pomoću grafičkih elemenata.

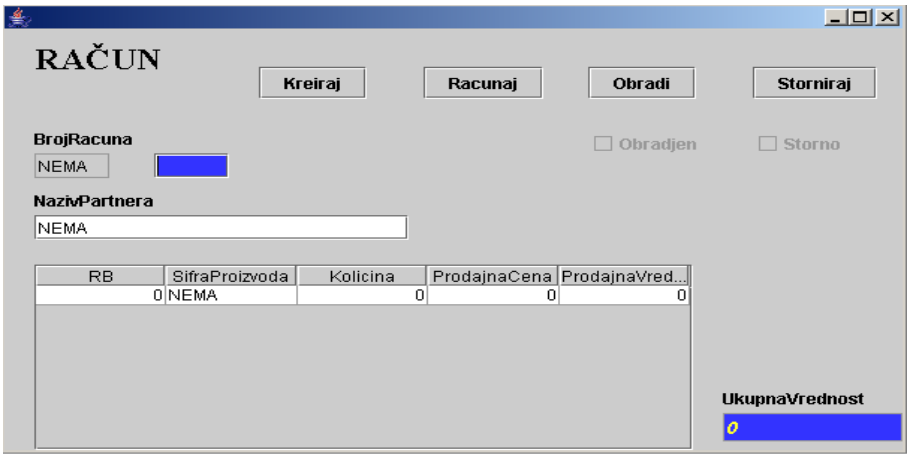
- Kontrolera korisničkog interfejsa koji je odgovoran da:
    a) prihvati podatke koje šalje ekranska forma,
    b) konvertuje podatke (koji se nalaze u grafičkim elementima) u objekat koji predstavlja ulazni argument SO koja će biti pozvana,
    a) šalje zahtev za izvršenje SO do aplikacionog servera (softverskog sistema),
    b) prihvata objekat (izlaz) softverskog sistema koji nastaje kao rezultat izvršenja SO i
    c) konvertuje objekat u podatke grafičkih elemenata.



Slika SKI: Struktura korisničkog interfejsa

1.3.8 PROJEKTOVANJE EKRANSKE FORME

Ekranska forma treba, za navedeni primer, da ima sledeći izgled:



Postoje dva aspekta projektovanja ekranske forme:

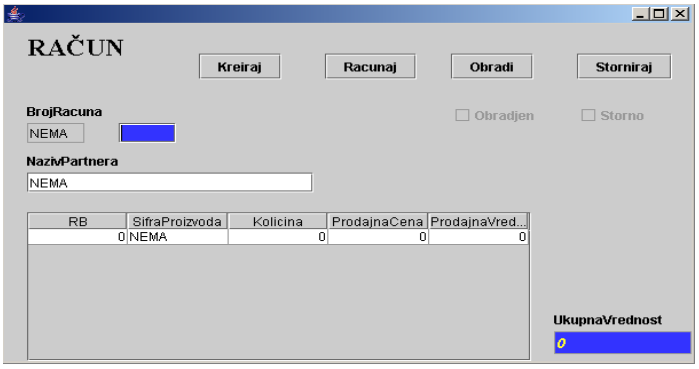
a) Projektovanje scenaria SK koji se izvode preko ekranske forme.
b) Projektovanje metoda ekranske forme.

Projektovanje scenaria SK

Svaki scenario SK koji se izvodi nad ekranskom formom se projektuje.

SKPZ1*: Slučaj korišćenja – Pravljenje novog računa*

***Preduslov**: Sistem je uključen i prodavac je ulogovan pod svojom šifrom. Sistem prikazuje formu za obradu računa (Kada korisnik pozove sistem da se izvrši (APSO), sistem inicijalno prikazuje formu na kojoj se nalazi zadnji uneti račun(IA)).*



***Napomena:** Mi polazimo od pretpostavke da je baza podataka u početku prazna. To znači da će ekranska forma na početku izvršenja programa, odnosno njena polja, biti ispunjena sa podrazumevanim vrednostima. U početku kada je baza prazna ekranska forma nije vezana ni za jedan račun, jer isti ne postoje u bazi podataka.*

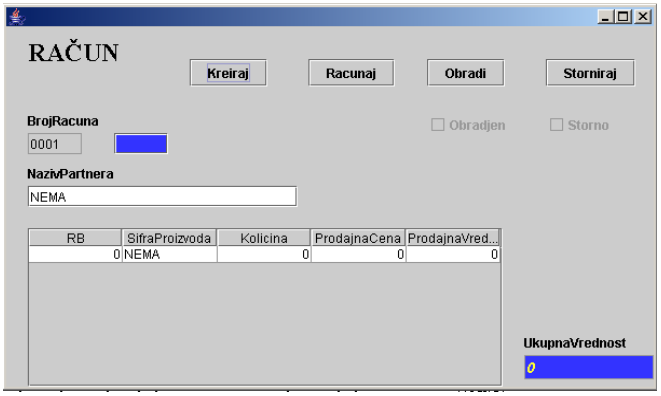
Osnovni scenarijo SK

1. *Prodavac poziva sistem da kreira novi račun. (APSO)*

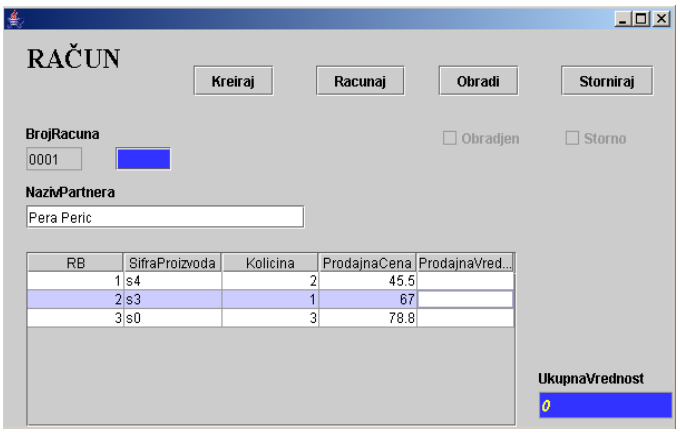
Opis akcije: Prodavac pritiska (jedan klik miša) dugme Kreiraj nakon čega se poziva sistemska operacija kreirajNovi() koja pravi novi račun.

2. *Sistem kreira novi račun. (SO)*
3. *Sistem prikazuje prodavcu novi račun. (IA)*

Pre nego što se na ekranskoj formi prikaže novi račun treba da bude prikazana poruka: Slog je uspešno kreiran i zapamćen u bazi ili slog nije uspesno kreiran i zapamćen u bazi u zavisnosti od uspešnosti izvršenja operacije.



4. *Prodavac unosi podatke u račun. (APUSO)*

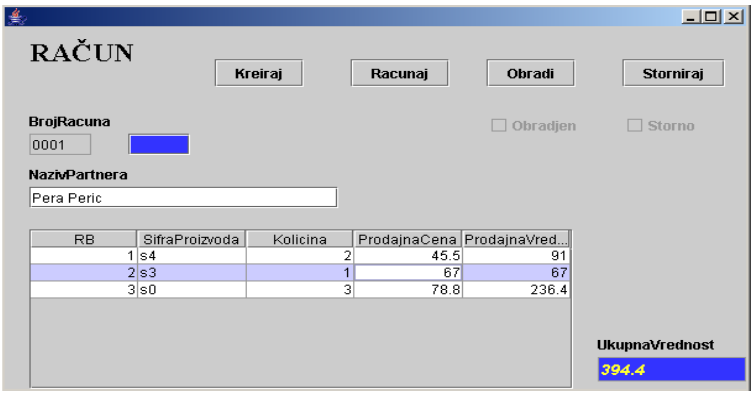


5. *Prodavac poziva sistem da izracuna iznose stavki racuna i ukupni iznos racuna.(APSO)*

Opis akcije: Prodavac pritiska dugme Racunaj ili pritiska tipku <Enter>, dok unosi podatke u poljima stavki računa, nakon čega se poziva sistemska operacija Racunaj() koja računa ProdajnuVrednost po svakoj stavci i UkupnuVrednost računa.

6. *Sistem računa iznose po svakoj stavci računa i ukupni iznos racuna.(SO)*
7. *Sistem prikazuje prodavcu izmenjen račun. (IA)*

Pre nego što se na ekranskoj formi prikaže račun sa promenjenim vrednostima navedenih polja treba da bude prikazana poruka: Slog je uspešno zapamćen u bazi ili slog nije uspesno zapamćen u bazi u zavisnosti od uspešnosti izvršenja operacije.

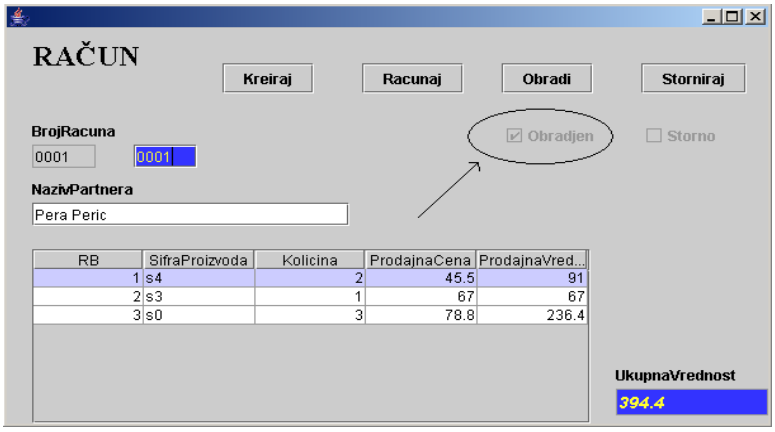


8. *Prodavac kontroliše da li je uneo sve potrebne podatke na računu. (ANSO)*
9. *Prodavac poziva sistem da izvrsi konacnu obradu računa. (APSO)*

Opis akcije: Prodavac pritiska dugme Obradi nakon čega se poziva sistemska operacija Obradi().

10. *Sistem obrađuje račun.(SO)*
11. *Sistem prikazuje prodavcu obrađen račun.(IA)*

Pre nego što se na ekranskoj formi prikaže račun sa promenjenim poljem Obradjen treba da bude prikazana poruka: Slog je uspešno zapamćen u bazi ili slog nije uspesno zapamćen u bazi u zavisnosti od uspešnosti izvršenja operacije.



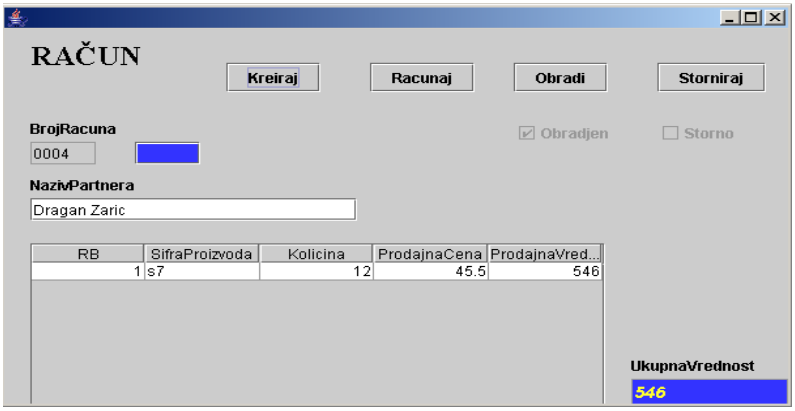
Polje Obradjen (zaokruženo na slici) biva označeno nakon uspešno izvršene obrade računa.

Alternativna scenarija

- *5.1 Ukoliko sistem ne može da računa iznose stavki računa i ukupni iznos računa on prikazuje prodavcu poruku da ne može da obradi račun.(IA) Prekida se izvršenje scenarija.*
- *9.1 Ukoliko sistem ne može da obradi račun on prikazuje prodavcu poruku da ne može da obradi račun.(IA) Prekida se izvršenje scenarija.***Napomena:** Na sličan način kao i kod kreiranja prvog računa, kreirali smo još tri računa kako bi smo mogli da demonstriramo slučajeve korišćenja koji slede.

SKPZ2*: Slučaj korišćenja – Provera postojanja računa*

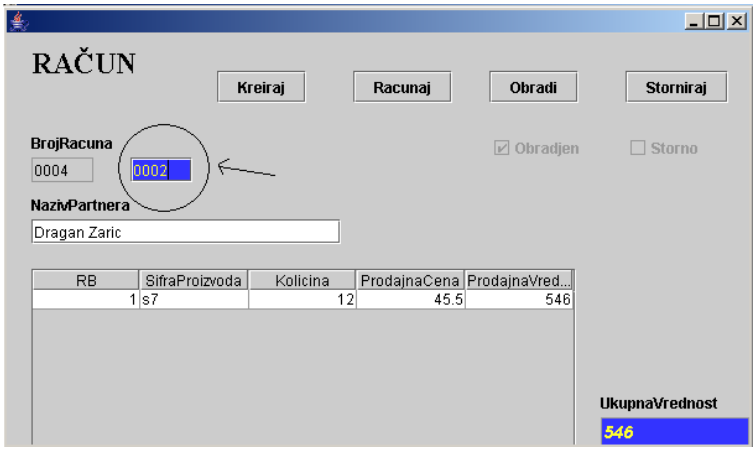
***Preduslov**: Sistem je uključen i prodavac je ulogovan pod svojom šifrom. Sistem prikazuje formu za obradu računa (IA).*



Osnovni scenarijo SK

1. Prodavac unosi broj računa koji želi da proveri. (APUSO)

Opis akcije: Korisnik unosi npr. račun broj 0002 u polje koje ima plavu pozadinu(na slici je to zaokruženo).

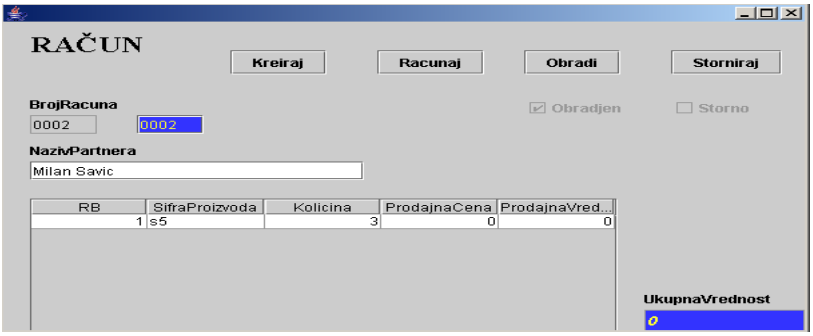


2. Prodavac poziva sistem da proveri da li račun sa zadatim brojem postoji.(APSO)

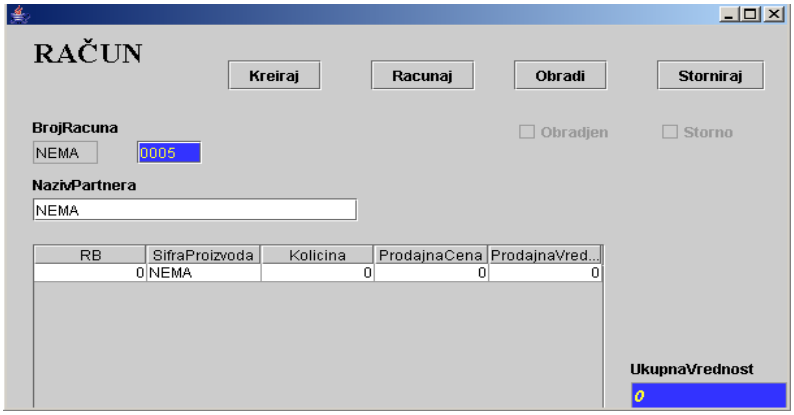
Opis akcije: Unos broja računa u polju pretraživanja treba završiti pritiskom na tipku <Enter> nakon čega se poziva sistemska operacija Pretrazi().

3. Sistem proverava postojanje računa.(SO)
4. Sistem prikazuje prodavcu račun. (IA)

Ukoliko je operacija uspešno izvršena na ekranskoj formi se prikazuje traženi račun:



U suprotnom, ako traženi račun ne postoji, prikazuje se ekranska forma sa podrazumevanim vrednostima:

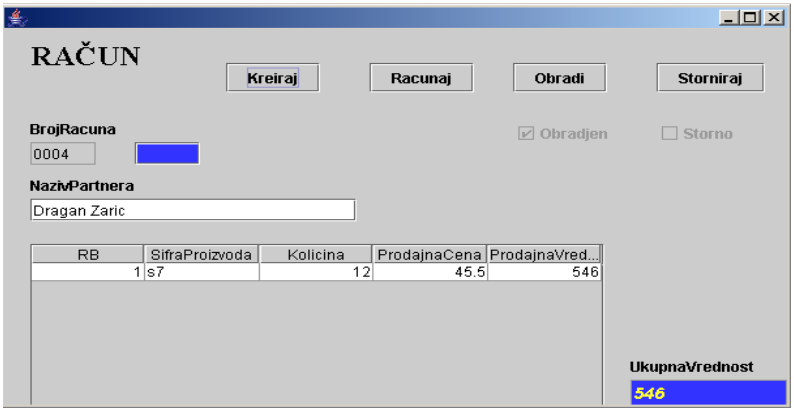


Alternativna scenarija

- *3.1 Ukoliko račun sa zadatim brojem ne postoji sistem prikazuje prodavcu poruku da račun ne postoji(IA). Prekida se izvršenje scenarija.*

SKPZ3*: Slučaj korišćenja – Storniranje računa*

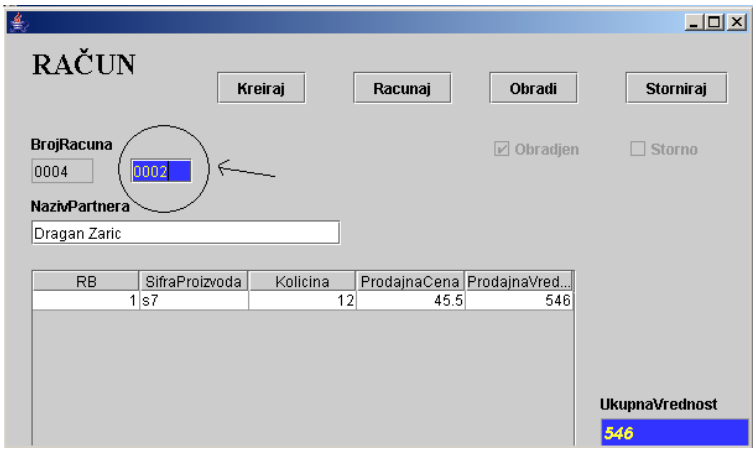
***Preduslov**: Sistem je uključen i prodavac je ulogovan pod svojom šifrom. Sistem prikazuje formu za obradu računa (IA).*



Osnovni scenarijo SK

1. Prodavac unosi broj računa koji želi da stornira. (APUSO)

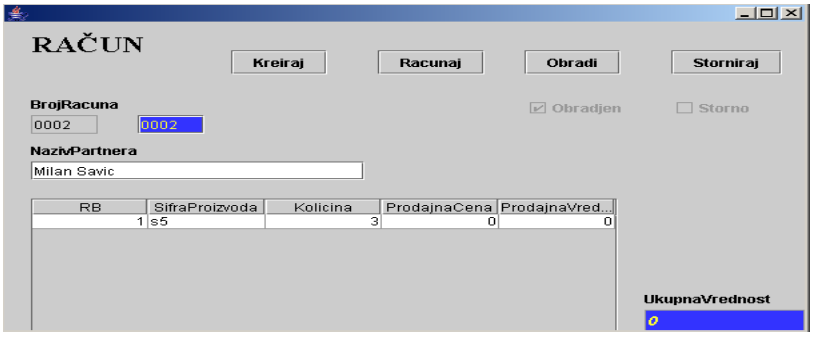
Opis akcije: Korisnik unosi npr. račun broj 0002 u polje pretraživanja koje ima plavu pozadinu (na slici je to zaokruženo).



2. Prodavac poziva sistem da proveri da li račun sa zadatim brojem postoji.(APSO)

Opis akcije: Unos broja računa u polju pretraživanja treba završiti pritiskom na tipku <Enter> nakon čega se poziva sistemska operacija Pretrazi().

3. Sistem proverava postojanje računa.(SO)
4. Sistem prikazuje prodavcu račun ukoliko postoji. (IA)

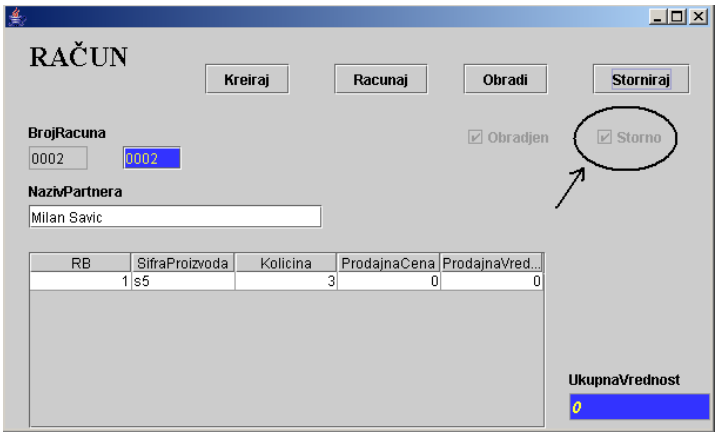


5. Prodavac poziva sistem da stornira zadati račun.(APSO)

Opis akcije: Prodavac pritiska dugme Storniraj nakon čega se poziva sistemska operacija Storniraj() koja vrši storniranje računa.

6. Sistem stornira račun. (SO)
7. Sistem prikazuje prodavcu storniran račun. (IA)

Pre nego što se na ekranskoj formi prikaže račun sa promenjenim poljem Storniran treba da bude prikazana poruka: Slog je uspešno promenjen u bazi ili slog nije uspesno promenjen u bazi u zavisnosti od uspešnosti izvršenja operacije.



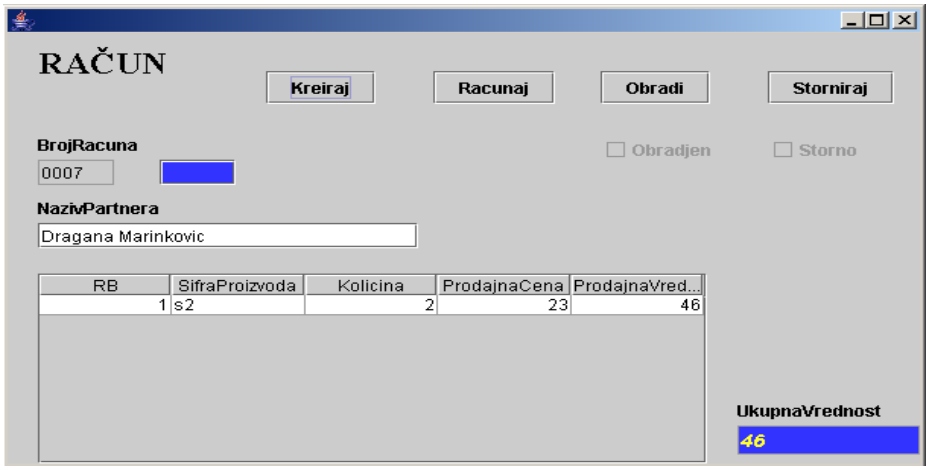
Polje Storniran (zaokruženo na slici) biva označeno nakon uspešno izvršenog storniranja računa.

Alternativna scenarija

- *2.1 Ukoliko račun sa zadatim brojem ne postoji sistem prikazuje prodavcu poruku da račun ne postoji(IA). Prekida se izvršenje scenarija.*
- *5.1 Ukoliko račun ne može da se stornira sistem prikazuje prodavcu poruku da ne može da stornira račun (IA). Prekida se izvršenje scenarija.*

SKPZ4*: Slučaj korišćenja – Dopuna računa*

***Preduslov**: Sistem je uključen i prodavac je ulogovan pod svojom šifrom. Sistem prikazuje formu za obradu računa (IA).*



Osnovni scenarijo SK

1. Prodavac unosi broj računa koji želi da dopuni. (APUSO)

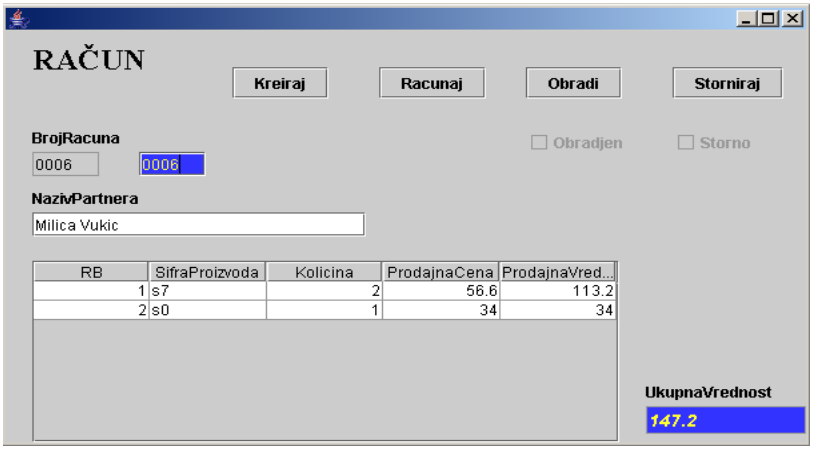
Opis akcije: Korisnik unosi npr. račun broj 0006 u polje pretraživanja.

3. Prodavac poziva sistem da proveri da li račun sa zadatim brojem postoji.(APSO)

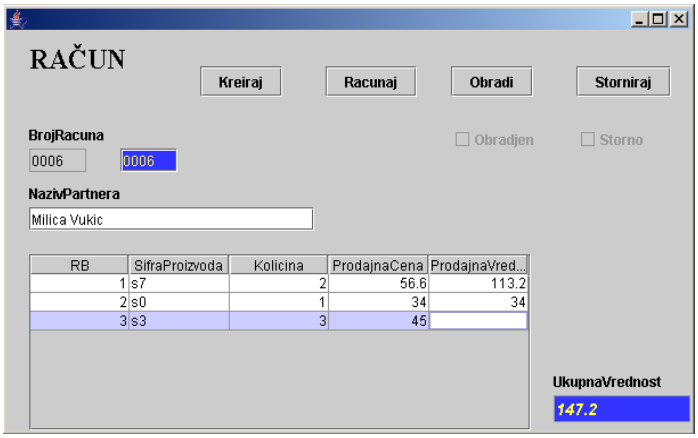
Opis akcije: Unos broja računa u polju pretraživanja treba završiti pritiskom na tipku <Enter> nakon čega se poziva sistemska operacija Pretrazi().

4. Sistem proverava postojanje računa.(SO)

5. Sistem prikazuje prodavcu račun. (IA)



6. Prodavac unosi dopunske podatke u račun. (APUSO)

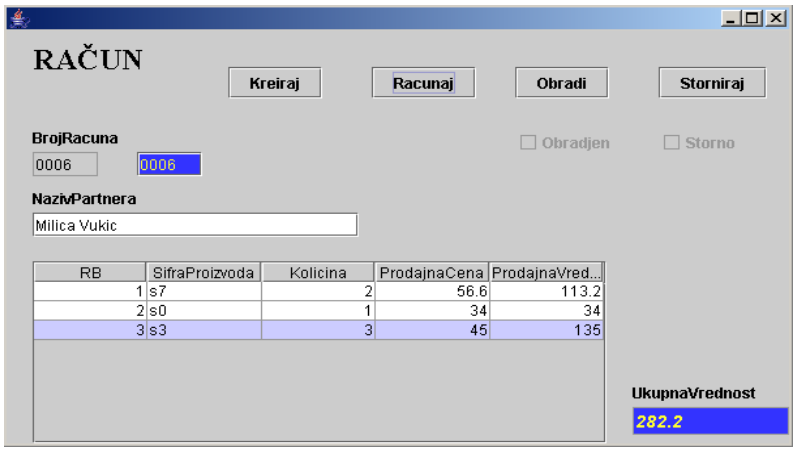


7. Prodavac poziva sistem da izracuna iznose stavki racuna i ukupni iznos racuna.(APSO)

Opis akcije: Prodavac pritiska dugme Racunaj ili pritiska tipku <Enter>, dok unosi podatke u poljima stavki računa, nakon čega se poziva sistemska operacija Racunaj() koja računa ProdajnuVrednost po svakoj stavci i UkupnuVrednost računa.

8. Sistem računa iznose po svakoj stavci računa i ukupni iznos racuna.(SO)

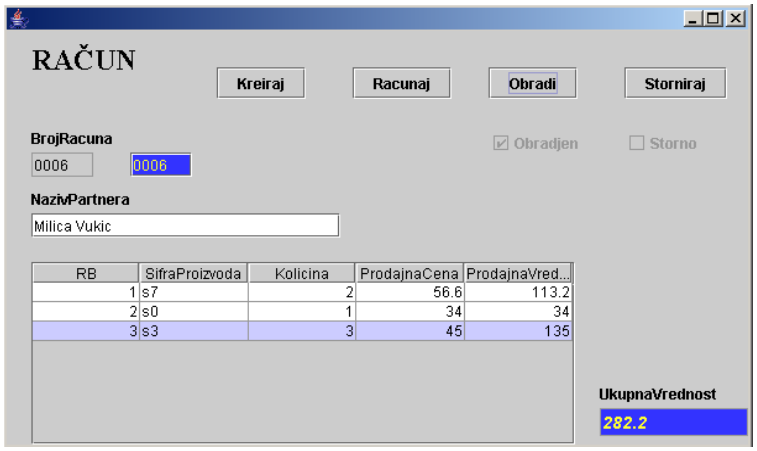
9. Sistem prikazuje prodavcu izmenjen račun. (IA)



10. *Prodavac kontroliše da li je uneo sve potrebne podatke na računu. (ANSO)*
11. *Prodavac poziva sistem da izvrsi konacnu obradu računa. (APSO)*

Opis akcije: Prodavac pritiska dugme Obradi nakon čega se poziva sistemska operacija Obradi().

12. *Sistem obrađuje račun.(SO)*
13. *Sistem javlja prodavcu da je obradio račun.(IA)*



Alternativna scenarija

- *3.1 Ukoliko račun ne postoji sistem prikazuje prodavcu poruku da račun ne postoji(IA). Prekida se izvršenje scenarija.*
- *6.1 Ukoliko sistem ne može da računa iznose stavki računa i ukupni iznos računa on prikazuje prodavcu poruku da ne može da obradi račun.(IA) Prekida se izvršenje scenarija.*
- *10.1 Ukoliko sistem ne može da obradi račun on prikazuje prodavcu poruku da ne može da obradi račun.(IA) Prekida se izvršenje scenarija.*

Projektovanje metoda ekranske forme

```
public class EkranskaForma extends JFrame {

// Glavni program
public static void main(String args[])
 {
   EkranskaForma EF    = new EkranskaForma();
   EF.show();

 }

// 1. Konstruktor kor. interfejsa
public EkranskaForma ()
  { KreirajKomponenteEkranskeForme();              // 1.1
    PokreniMenadzeraRasporedaKomponeti();          // 1.2
    PostaviImeDokumenta();                         // 1.3
    PostaviPoljeBrojRacuna();                      // 1.4
    PostaviPoljeNazivPartnera();                   // 1.5
    PostaviPoljeUkupnaVrednost();                  // 1.6
    PostaviKBObradjen();                           // 1.7
    PostaviKBStorniran();                          // 1.8
    PostaviTabelu();                               // 1.9
    PostaviLabeluBrojRacuna();                     // 1.10
    PostaviLabeluNazivPartnera();                  // 1.11
    PostaviLabeluUkupnaVrednost();                 // 1.12
    PostaviDugmeKreiraj();                         // 1.13
    PostaviDugmeObradi();                          // 1.14
    PostaviDugmeStorniraj();                       // 1.15
    PostaviDugmeRacunaj();                         // 1.16
    Izvrsavanje();                                 // 1.17
    PostaviPoljePretrazivanje();                   // 1.18
    pack();
    }


// 1.1 Kreiranje i inicijalizacija komponenti ekranske forme
void KreirajKomponenteEkranskeForme()
  {  LNazivDokumenta = new JLabel();
     BrojRacuna = new JFormattedTextField();
     NazivPartnera = new JFormattedTextField();
     UkupnaVrednost = new JFormattedTextField();
     Obradjen = new JCheckBox();
     Storniran = new JCheckBox();
     TabSlogoviRacuna = new JTable();
     LBrojRacuna = new JLabel();
     LNazivPartnera = new JLabel();
     LUkupnaVrednost = new JLabel();
     Kreiraj = new JButton();
     Racunaj = new JButton();
     Obradi = new JButton();
     Storniraj = new JButton();
     Pretrazivanje = new JFormattedTextField();
  }


// 1.2 Kreiranje menadjera rasporeda komponenti i njegovo dodeljivanje do kontejnera okvira(JFrame komponente).
void PokreniMenadzeraRasporedaKomponeti()
  { getContentPane().setLayout(new AbsoluteLayout());}

// 1.3 Odredivanje naslovnog teksta i njegovo dodeljivanje do kontejnera okvira.
void PostaviImeDokumenta()
{  LNazivDokumenta.setFont(new Font("Times New Roman", 1, 24));
   LNazivDokumenta.setText("RA\u010cUN");
   getContentPane().add(LNazivDokumenta, new AbsoluteConstraints(20, 10, -1, -1));
}
```

```
// POCETAK ODELJKA KOMPONENTI PREKO KOJIH SE PRIHVATAJU PODACI

// Komponente koje reprezentuju klasu Racun koja ima atribute: String BrojRacuna;
// String NazivPartnera; Double UkupnaVrednost; boolean Obradjen; boolean Storniran;
    private JFormattedTextField BrojRacuna;     // setValue(new String("nema");
    private JFormattedTextField NazivPartnera;  // setValue(new String("nema");
    private JFormattedTextField UkupnaVrednost; // setValue(new Double(0);
    private JCheckBox Obradjen; //  isSelected metoda vraca boolean
    private JCheckBox Storniran; // isSelected metoda vraca boolean

// 1.4
void PostaviPoljeBrojRacuna()
 { // Dodeljivanje pocetne vrednosti i formata polja.
   BrojRacuna.setValue(new String("nema"));
   // Vrednost polja ne moze da se menja.
   BrojRacuna.setEditable(false);
   // Polje se dodaje kontejneru okvira (JFrame).
    getContentPane().add(BrojRacuna, new AbsoluteConstraints(20, 100, 50, -1));
}

// 1.5
void PostaviPoljeNazivPartnera()
 {  // Dodeljivanje pocetne vrednosti i formata polja.
   NazivPartnera.setValue(new String("nema"));
    // Polje se dodaje kontejneru okvira (JFrame)
   getContentPane().add(NazivPartnera, new AbsoluteConstraints(20, 150, 250, -1));
 }

// 1.6
void PostaviPoljeUkupnaVrednost()
  { // Dodeljivanje pocetne vrednosti i formata polja.
    UkupnaVrednost.setValue(new Double(0));
   // Odredjivanje boje pozadina polja.
    UkupnaVrednost.setBackground(new Color(51, 51, 255));
    // Odredjivanje boje crtanja polja.
    UkupnaVrednost.setForeground(new Color(255, 255, 51));
    // Odredjivanje fonta koji ce biti koriscen kod unosa podataka u polje.
   UkupnaVrednost.setFont(new Font("Dialog", 3, 14));
    // Vrednost polja ne moze da se menja.
   UkupnaVrednost.setEditable(false);
    // Polje se dodaje kontejneru okvira (JFrame)
   getContentPane().add(UkupnaVrednost, new AbsoluteConstraints(480, 310, 120, -1));
  }

// 1.7
void PostaviKBObradjen()
 { // Dodeljivanje opisa kontrolnog(check) boksa.
   Obradjen.setText("Obradjen");
   //Obradjen.setFocusCycleRoot(true); IZBACI NAREDBU POSLE PROVERE
   // Kontrolni boks se ne moze fokusirati.
   Obradjen.setFocusable(false);
   // Kontrolni boks se ne moze menjati.
   Obradjen.setEnabled(false);
   // Polje se dodaje kontejneru okvira (JFrame)
   getContentPane().add(Obradjen, new AbsoluteConstraints(390, 80, -1, -1));
}

// 1.8
void  PostaviKBStorniran()
  { // Dodeljivanje opisa kontrolnog(check) boksa.
    Storniran.setText("Storno");
    // Kontrolni boks se ne moze fokusirati.
    Storniran.setFocusable(false);
    // Kontrolni boks se ne moze menjati.
    Storniran.setEnabled(false);
    // Polje se dodaje kontejneru okvira (JFrame)
   getContentPane().add(Storniran, new AbsoluteConstraints(500, 80, -1, -1));
  }

// Komponenete koje reprezentuju niz objakata klase StavkaRacuna koja ima atribute: Integer RB;
// String SifraProizvoda; Integer Kolicina;  Double ProdajnaCena;  Double ProdajnaVrednost;
    JTable TabSlogoviRacuna;

// Atributi potrebni za definisanje tabele.
    DefaultTableModel DTM;
    JComboBox SifrePr;
// 1.9
```

```
void PostaviTabelu()
{/// 1. - Kreiranje i inicijalizacija modela tabele.
 DTM =  new DefaultTableModel (StavkaRacuna.ZagSlogoviRacuna(),1)
      {  // LOGICKO POVEZIVANJE (PO TIPU) KOLONA TABELE SA ATRIBUTIMA KLASE StavkeRacun.
         // Inicilazacija KOLONA TABELE sa tipovima klasa koje su identicne tipovima
         // atributa klase StavkeRacuna ( Integer.class, String.class, Integer.class,
         // Double.class, Double.class).
         Class[] types = StavkaRacuna.vratiTipove();
         // Ova metoda onemogucava unos podataka neodgovarajuceg tipa u polja (celije) tabele.
         // npr: Ukoliko je kolona vezana za Double klasu u nju se ne mogu uneti String podaci.
        public Class getColumnClass(int columnIndex){return types [columnIndex];}
      };
// 2. - Povezivanje tabele sa modelom tabele.
TabSlogoviRacuna.setModel(DTM);

// 3. - Tabela postaje slusalac dogadjaja, koji ce se desiti nakon pritiska tipke
// na bilo kom polju tabele.
TabSlogoviRacuna.addKeyListener(new KeyAdapter()
  {  /*********************************************************/
              // DOGADJAJ KOJI INICIRA POZIV SISTEMSKE OPERACIJE
               public void keyPressed(KeyEvent evt)
        {int signal = KontrolerKl.PritisakTipke (evt,BrojRacuna,NazivPartnera, UkupnaVrednost,
                                                 Obradjen,Storniran,TabSlogoviRacuna,DTM);

          // Prikaz poruke o uspesnosti izvrsenja operacije
          PrikazPorukeUspesnosti(signal);
         }
    /*********************************************************/
  });

// 4. - Tabela postaje slusalac dogadjaja, koji ce se desiti nakon otpustanja tipke
// na bilo kom polju tabele.
TabSlogoviRacuna.addKeyListener(new KeyAdapter()
  {  /*********************************************************/
              // DOGADJAJ KOJI INICIRA POZIV SISTEMSKE OPERACIJE
              public void keyReleased(KeyEvent evt)
        {int signal = KontrolerKl.PustanjeTipke (evt,BrojRacuna,NazivPartnera,UkupnaVrednost,
                                                 Obradjen,Storniran,TabSlogoviRacuna,DTM);

          // Prikaz poruke o uspesnosti izvrsenja operacije
          PrikazPorukeUspesnosti(signal);
         }
    /*********************************************************/
  });

// 5.Kreiranje i inicijalizacija kombo boksa, koji ce da sadrzi sifre proizvoda.
SifrePr = new JComboBox();
for(int i=0;i<=20;i++)
   SifrePr.addItem(new String("s" + i));
// Povezivanje kombo boksa sa kolonom tabele.
TableCellEditor tce = new DefaultCellEditor(SifrePr);
TableColumnModel columnModel = TabSlogoviRacuna.getColumnModel();
TableColumn tk = columnModel.getColumn(1); // 1 oznacava drugu kolonu tabele
tk.setCellEditor(tce);
// 6.Kreiranje i inicijalizacija objekta koji pomaze da se tabela skroluje hotizontalno
// i vertikalno.
int vsb = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
int hsb = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
JScrollPane skrol = new JScrollPane(TabSlogoviRacuna,vsb,hsb);
// Skrol objekat se dodaje kontejneru okvira (JFrame)
getContentPane().add(skrol, new AbsoluteConstraints(20, 190, 440,150));

}

// KRAJ ODELJKA KOMPONENTI PREKO KOJIH SE PRIHVATAJU PODACI


// POCETAK ODELJKA KOMPONENTI - LABELA KOJE OPISUJU POLJA ZA PRIHVAT PODATAKA
   private JLabel LNazivDokumenta;
   private JLabel LBrojRacuna;
   private JLabel LNazivPartnera;
   private JLabel LUkupnaVrednost;


// 1.10
```

```
void PostaviLabeluBrojRacuna()
{ LBrojRacuna.setText("BrojRacuna"); getContentPane().add(LBrojRacuna, new AbsoluteConstraints(20, 80, -1, -1)); }

// 1.11
void  PostaviLabeluNazivPartnera()
 { LNazivPartnera.setText("NazivPartnera");
   getContentPane().add(LNazivPartnera, new AbsoluteConstraints(20, 130, -1, -1)); }

// 1.12
void  PostaviLabeluUkupnaVrednost()
{ LUkupnaVrednost.setText("UkupnaVrednost");
  getContentPane().add(LUkupnaVrednost, new AbsoluteConstraints(480, 290, -1, -1));}

// KRAJ ODELJKA KOMPONENTI - LABELA KOJE OPISUJU POLJA ZA PRIHVAT PODATAKA

// POCETAK ODELJKA KOMPONENTI (DUGMADI) PREKO KOJIH SE POZIVAJU SISTEMSKE OPERACIJE.
    private JButton Kreiraj;    private JButton Racunaj;    private JButton Obradi;    private JButton Storniraj;

// 1.13
void PostaviDugmeKreiraj()
 { Kreiraj.setText("Kreiraj");
   Kreiraj.addActionListener(new ActionListener() {
   /*************************************************************/
   // DOGADJAJ KOJI INICIRA POZIV SISTEMSKE OPERACIJE
   public void actionPerformed(ActionEvent evt)
         { int signal = KontrolerKI.SOKreirajNovi (BrojRacuna,NazivPartnera,UkupnaVrednost,
                                                   Obradjen,Storniran,TabSlogoviRacuna,DTM);

           // Prikaz poruke o uspesnosti izvrsenja operacije
           PrikazPorukeUspesnosti(signal);
          }
     });
   /*************************************************************/
  getContentPane().add(Kreiraj, new AbsoluteConstraints(170, 30, -1, -1));
}

// 1.14
void PostaviDugmeRacunaj()
 { Racunaj.setText("Racunaj");
   Racunaj.addActionListener(new ActionListener() {
   /*************************************************************/
   // DOGADJAJ KOJI INICIRA POZIV SISTEMSKE OPERACIJE
   public void actionPerformed(ActionEvent evt)
         { int signal = KontrolerKI.SORacunaj (BrojRacuna,NazivPartnera,UkupnaVrednost,
                                               Obradjen,Storniran,TabSlogoviRacuna,DTM);
           // Prikaz poruke o uspesnosti izvrsenja operacije
           PrikazPorukeUspesnosti(signal);
          }
     });
   /*************************************************************/
  getContentPane().add(Racunaj, new AbsoluteConstraints(280, 30, -1, -1));
}

// 1.15
void PostaviDugmeObradi()
 {
   Obradi.setText("Obradi");
   Obradi.addActionListener(new ActionListener() {
     // DOGADJAJ KOJI INICIRA POZIV SISTEMSKE OPERACIJE
     /*************************************************************/
     public void actionPerformed(ActionEvent evt)
           { int signal = KontrolerKI.SOObradi (BrojRacuna,NazivPartnera,UkupnaVrednost,
                                                Obradjen,Storniran,TabSlogoviRacuna,DTM);

                    // Prikaz poruke o uspesnosti izvrsenja operacije
             PrikazPorukeUspesnosti(signal);
            }
     /*************************************************************/
      });
   getContentPane().add(Obradi, new AbsoluteConstraints(390, 30, -1, -1));
 }


// 1.16
void  PostaviDugmeStorniraj()
```

```
{ Storniraj.setText("Storniraj");
  Storniraj.addActionListener(new ActionListener() {
                // DOGADJAJ KOJI INICIRA POZIV SISTEMSKE OPERACIJE
                /*********************************************************/
        public void actionPerformed(ActionEvent evt)
        { int signal = KontrolerKI.SOStorniraj (BrojRacuna,NazivPartnera,UkupnaVrednost,
                                                Obradjen,Storniran,TabSlogoviRacuna,DTM);

          // Prikaz poruke o uspesnosti izvrsenja operacije
          PrikazPorukeUspesnosti(signal);
          }
     /*********************************************************/
    });

  getContentPane().add(Storniraj, new AbsoluteConstraints(500, 30, -1, -1));
}

// 1.17
void Izvrsavanje()
{  // POZIV SISTEMSKE OPERACIJE
            int signal = KontrolerKI.SOIzvrsi(BrojRacuna,NazivPartnera,UkupnaVrednost,Obradjen,Storniran,
TabSlogoviRacuna,DTM);
   // Prikaz poruke o uspesnosti izvrsenja operacije
    PrikazPorukeUspesnosti(signal);
}

// POCETAK ODELJKA KOMPONENTI (POLJA) PREKO KOJIH SE PRETRAZUJU RACUNU
// 1.18

private JFormattedTextField Pretrazivanje;

void PostaviPoljePretrazivanje()
 { // Dodeljivanje pocetne vrednosti i formata polja.
   Pretrazivanje.setValue(new String(""));
   // Polje postaje slusalac dogadjaja,koji ce se desiti nakon unosa podataka u polje.
   Pretrazivanje.setBackground(new Color(51, 51, 255));
      // Odredjivanje boje crtanja polja.
   Pretrazivanje.setForeground(new Color(255, 255, 51));
   Pretrazivanje.addActionListener
     (new ActionListener()
        // DOGADJAJ KOJI INICIRA POZIV SISTEMSKE OPERACIJE
      { /*********************************************************/
                public void actionPerformed(ActionEvent evt)
        { int signal = KontrolerKI.SOPretrazi
(Pretrazivanje,BrojRacuna,NazivPartnera,UkupnaVrednost,
                                                Obradjen,Storniran,TabSlogoviRacuna,DTM);

        // Prikaz poruke o uspesnosti izvrsenja operacije
           PrikazPorukeUspesnosti(signal);

        }
        /*********************************************************/
       }
              );
             // Polje se dodaje kontejneru okvira (JFrame).
    getContentPane().add(Pretrazivanje, new AbsoluteConstraints(100, 100, 50, -1));
}

// KRAJ ODELJKA KOMPONENTI (POLJA) PREKO KOJIH SE PRETRAZUJU RACUNU
// KRAJ ODELJKA KOMPONENTI (DUGMADI) PREKO KOJIH SE POZIVAJU SISTEMSKE OPERACIJE.

// Nakon izvrsenja sistemskih operacija poziva se ova metoda koja prikazuje
// dijalog boks sa porukom o uspesnosti izvrsenja sistemske operacije.
void PrikazPorukeUspesnosti(int signal)
{ Boolean prikazi = new Boolean(true);;
  DijalogBoksPrikazPoruke DBPP = new DijalogBoksPrikazPoruke(this);
  int signal1 = DBPP.PrikazPoruke(signal);
  if (signal1 == 1)
    { DBPP.show();
      pack();
    }
  }
}
class DijalogBoksPrikazPoruke extends JDialog
{ public DijalogBoksPrikazPoruke(JFrame roditelj)
   { super(roditelj,"-Prikaz poruka o izvrsenim operacijama-", true);} // true definise modalni dijalog
```

```
int PrikazPoruke(int signal)
{ String poruka;
  Box b = Box.createVerticalBox();
  b.add(Box.createGlue());
 System.out.println("Signal na dijalogu:" + signal);
        switch (signal)
         {      case 21: poruka = new String("Unet novi red u kolonu");break;
                case 22: poruka = new String("Red ne moze da se brise ako nije selektovan");break;
                case 23: poruka = new String("Pritisnuta je tipka koja se ne obradjuje");return 0;
                case 31: poruka = new String("Slog je uspesno zapamcen");break;
                case 32: poruka = new String("Neuspesno pamcenje sloga u bazi");break;
                case 33: poruka = new String("Slog ili slogovi uspesno obrisani u bazi");break;
                case 34: poruka = new String("Neuspesno brisanje sloga u bazi");break;
                case 35: poruka = new String("Slog je uspesno promenjen");break;
                case 36: poruka = new String("Neuspesna promena sloga u bazi");break;
                case 41: poruka = new String("Baza je uspesno otvorena");return 0;
                case 42: poruka = new String("Baza je neuspesno otvorena (drajver).");break;
                case 43: poruka = new String("Baza je neuspesno otvorena (konekcija).");break;
                case 44: poruka = new String("Baza je neuspesno otvorena (zastita).");break;
                case 45: poruka = new String("Slog je uspesno kreiran i zapamcen u bazi.");break;
                case 46: poruka = new String("Slog nije uspesno kreiran i zapamcen u bazi.");break;
                case 47: poruka = new String("Uspesno povecan broj racuna.");return 0;
                case 48: poruka = new String("Neuspesno povecan broj racuna");break;
                case 51: poruka = new String("Uspesno je uradjen commit transakcije.");break;
                case 52: poruka = new String("Neuspesno je uradjen commit transakcije.");break;
                case 53: poruka = new String("Uspesno je uradjen rollback transakcije.");break;
                case 54: poruka = new String("Neuspesno je uradjen rollback transakcije.");break;
                case 55: poruka = new String("Uspesno dodeljen broj racuna.");return 0;
                case 56: poruka = new String("Neuspesno dodeljen broj racuna.");break;
                case 57: poruka = new String("Uspesno obradjen racun.");break;
                case 58: poruka = new String("Neuspesno obradjen racun.");break;
                case 61: poruka = new String("Uspesno zatvorena baza.");return 0;
                case 62: poruka = new String("Neuspesno zatvorena baza.");break;
                case 71: poruka = new String("Uspesno pretrazivanje slogova");return 0;
                case 72: poruka = new String("Neuspesno pretrazivanje slogova");break;
                case 75: poruka = new String("Uspesno procitan zadnji slog iz baze, ukoliko postoji");return 0;
                case 76: poruka = new String("Neuspesno procitan zadnji slog iz baze");return 0;
                case 77: poruka = new String("Zadovoljen preduslov SO");return 0;
                case 78: poruka = new String("Nije dobro izracunata ukupna vrednost racuna");break;
                case 79: poruka = new String("Uspesno izracunata ukupna vrednost");return 0;
                case 93: poruka = new String("Racun je vec storniran");break;
                case 94: poruka = new String("Pokusaj da se radi sa obradjenim ili storniranim racunom");break;
                case 157: poruka = new String("Uspesno storniran racun");break;
                case 158: poruka = new String("Neuspesno storniran racun");break;
                case 171: poruka = new String("Uspesno vracen broj zadnjeg sloga");break;
                case 173: poruka = new String("Neuspesno vracen broj zadnjeg sloga");break;
                default: poruka = null;return 0;
         }
  b.add(new JLabel(poruka));
  getContentPane().add(b,"Center");
  setSize(450,70);
  return 1;
 }

}
```

1.3.9 PROJEKTOVANJE KONTROLERA KI

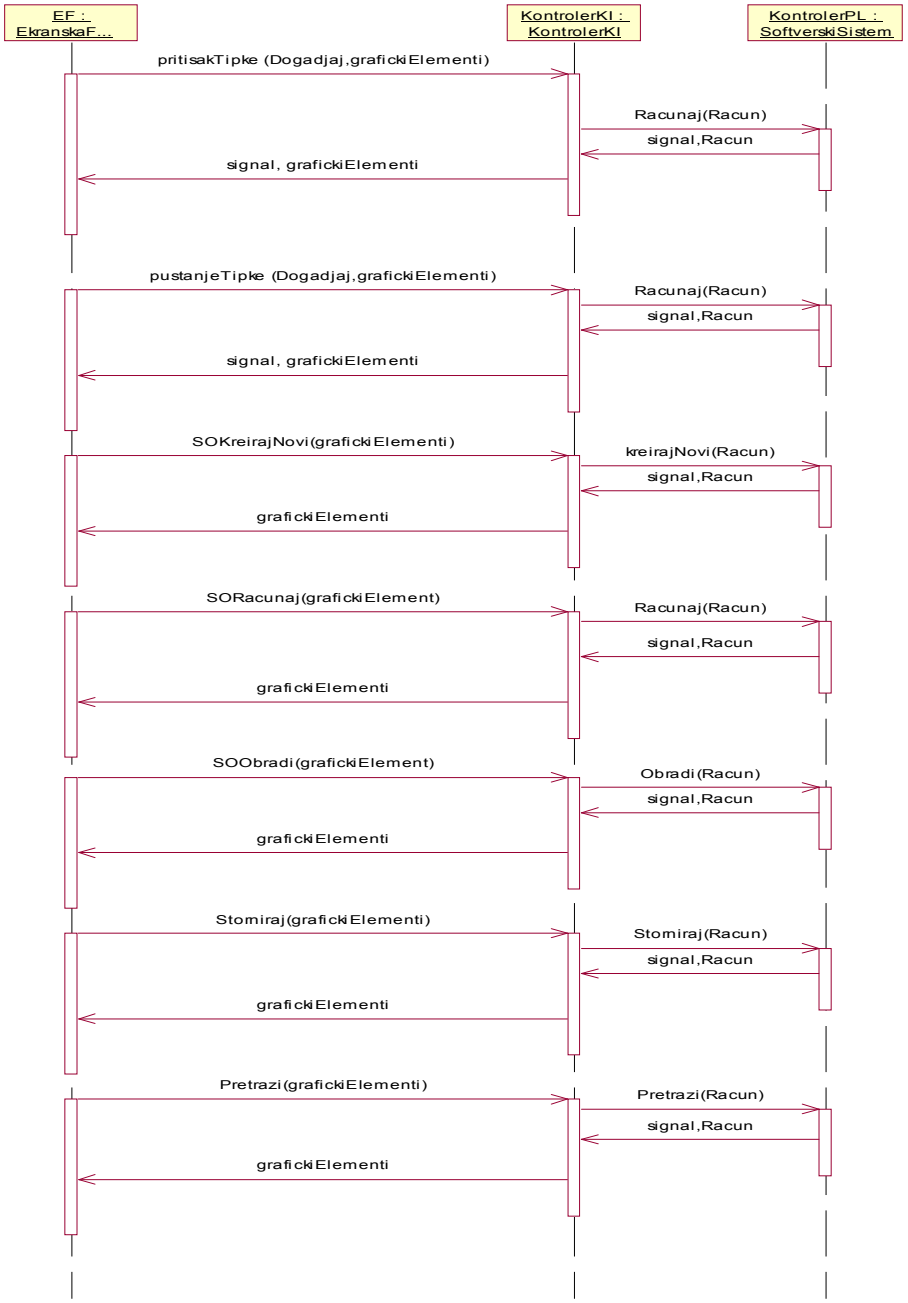
```
class KontrolerKI
 { static public int PritisakTipke(KeyEvent evt,JFormattedTextField BrojRacuna,
JFormattedTextField  NazivPartnera,JFormattedTextField UkupnaVrednost,JCheckBox
Obradjen,JCheckBox Storniran,JTable TabSlogoviRacuna, DefaultTableModel DTM)
    {  if (evt.getKeyCode() == KeyEvent.VK_INSERT)
        { DTM.addRow(StavkaRacuna.vratiPocetneVrednosti());  return 21; // Unet novi red u kolonu
        }
       if (evt.getKeyCode() == KeyEvent.VK_F1)
        { int selRed = TabSlogoviRacuna.getSelectedRow();
          if ( selRed>=0)
             { DTM.removeRow(selRed); // Obrisan novi red u koloni
           return SORacunaj(BrojRacuna,NazivPartnera,UkupnaVrednost,Obradjen,Storniran,
                            TabSlogoviRacuna,DTM);
             }
           else
             return 22; // Red ne moze da se brise ako nije selektovan.
           }
        return 23; // Uneta je tipka koja se ne obradjuje.
    }

static public int PustanjeTipke(KeyEvent evt,JFormattedTextField BrojRacuna,
JFormattedTextField NazivPartnera,JFormattedTextField UkupnaVrednost, JCheckBox Obradjen,
JCheckBox Storniran,JTable TabSlogoviRacuna,DefaultTableModel DTM)
    {  if (evt.getKeyCode() == KeyEvent.VK_ENTER)
        { return SORacunaj(BrojRacuna,NazivPartnera,UkupnaVrednost,Obradjen,Storniran,
                           TabSlogoviRacuna,DTM);
        }
       return 23; // Uneta je tipka koja se ne obradjuje.
    }


static public int SOPretrazi(JFormattedTextField Pretrazivanje, JFormattedTextField
BrojRacuna,JFormattedTextField NazivPartnera, JFormattedTextField UkupnaVrednost,
JCheckBox Obradjen,JCheckBox Storniran,JTable TabSlogoviRacuna, DefaultTableModel DTM)
{   Racun rac = new Racun();
KonvertujGrafickeKomponenteUObjekatRacun(rac,Pretrazivanje,NazivPartnera,UkupnaVrednost,Obradjen,Storniran,
TabSlogoviRacuna);
          /******** POZIVA SE KONTROLER POSL. LOGIKE DA IZVRSI SISTEMSKU OPERACIJU *********/
                              int signal = KontrolerPL.Pretrazi(rac);
                    /*******************************************************************************/
KonvertujObjekatRacunUGrafickeKomponente(rac,BrojRacuna,NazivPartnera,UkupnaVrednost,Obradjen,Storniran,
TabSlogoviRacuna, DTM);

return signal;
  }


static public int SOIzvrsi(JFormattedTextField BrojRacuna,JFormattedTextField
NazivPartnera,JFormattedTextField UkupnaVrednost,JCheckBox Obradjen,JCheckBox
Storniran,JTable TabSlogoviRacuna,DefaultTableModel DTM)
{   Racun rac = new Racun();
KonvertujGrafickeKomponenteUObjekatRacun(rac,BrojRacuna,NazivPartnera,UkupnaVrednost,Obradjen,Storniran,T
abSlogoviRacuna);
          /******** POZIVA SE KONTROLER POSL. LOGIKE DA IZVRSI SISTEMSKU OPERACIJU *********/
                              int signal = KontrolerPL.Izvrsi(rac);
                    /*******************************************************************************/
KonvertujObjekatRacunUGrafickeKomponente(rac,BrojRacuna,NazivPartnera,UkupnaVrednost,Obradjen,Storniran,T
abSlogoviRacuna,DTM);

return signal;
 }
```

```
static public int SOKreirajNovi(JFormattedTextField BrojRacuna,JFormattedTextField
NazivPartnera,JFormattedTextField UkupnaVrednost,JCheckBox Obradjen,JCheckBox
Storniran, JTable TabSlogoviRacuna,DefaultTableModel DTM)
{  Racun rac = new Racun();

      /******** POZIVA SE KONTROLER POSL. LOGIKE DA IZVRSI SISTEMSKU OPERACIJU *********/
                        int signal = KontrolerPL.kreirajNovi(rac);
     /*****************************************************************************/

KonvertujObjekatRacunUGrafickeKomponente(rac,BrojRacuna,NazivPartnera,UkupnaVrednost,Obradjen,Storniran,T
abSlogoviRacuna,DTM);

return signal;
}

static public int SORacunaj(JFormattedTextField BrojRacuna,JFormattedTextField
NazivPartnera, JFormattedTextField UkupnaVrednost,JCheckBox Obradjen, JCheckBox
Storniran, JTable TabSlogoviRacuna,DefaultTableModel DTM)
 { Racun rac = new Racun();
KonvertujGrafickeKomponenteUObjekatRacun(rac,BrojRacuna,NazivPartnera,UkupnaVrednost,Obradjen,Storniran,T
abSlogoviRacuna);
      /******** POZIVA SE KONTROLER POSL. LOGIKE DA IZVRSI SISTEMSKU OPERACIJU *********/
                        int signal = KontrolerPL.Racunaj(rac);
      /*****************************************************************************/

KonvertujObjekatRacunUGrafickeKomponente(rac,BrojRacuna,NazivPartnera,UkupnaVrednost,Obradjen,Storniran,T
abSlogoviRacuna,DTM);
return signal;
}

static public int SOStorniraj(JFormattedTextField BrojRacuna, JFormattedTextField
NazivPartnera, JFormattedTextField UkupnaVrednost,JCheckBox Obradjen,JCheckBox
Storniran, JTable TabSlogoviRacuna,DefaultTableModel DTM)
{   Racun rac = new Racun();
KonvertujGrafickeKomponenteUObjekatRacun(rac,BrojRacuna,NazivPartnera,UkupnaVrednost,Obradjen,Storniran,T
abSlogoviRacuna);
      /******** POZIVA SE KONTROLER POSL. LOGIKE DA IZVRSI SISTEMSKU OPERACIJU *********/
                        int signal =  KontrolerPL.Storniraj(rac);
     /*****************************************************************************/

KonvertujObjekatRacunUGrafickeKomponente(rac,BrojRacuna,NazivPartnera,UkupnaVrednost,Obradjen,Storniran,T
abSlogoviRacuna,DTM);
return signal;
}

static public int SOObradi(JFormattedTextField BrojRacuna,JFormattedTextField NazivPartnera,
JFormattedTextField UkupnaVrednost,JCheckBox Obradjen,JCheckBox Storniran, JTable
TabSlogoviRacuna,DefaultTableModel DTM)
{   Racun rac = new Racun();
KonvertujGrafickeKomponenteUObjekatRacun(rac,BrojRacuna,NazivPartnera,UkupnaVrednost,Obradjen,Storniran,T
abSlogoviRacuna);
          /******** POZIVA SE KONTROLER POSL. LOGIKE DA IZVRSI SISTEMSKU OPERACIJU *********/
                        int signal = KontrolerPL.Obradi(rac);
          /*****************************************************************************/

          KonvertujObjekatRacunUGrafickeKomponente(rac,BrojRacuna,NazivPartnera,UkupnaVrednost,Obradjen,
                                                   Storniran,TabSlogoviRacuna,DTM);
          return signal;
          }

 static void KonvertujGrafickeKomponenteUObjekatRacun(Racun rac,JFormattedTextField BrojRacuna,JFormattedTextField
NazivPartnera,JFormattedTextField UkupnaVrednost,JCheckBox Obradjen,JCheckBox Storniran,JTable TabSlogoviRacuna)
 {   rac.Napuni(BrojRacuna,NazivPartnera,UkupnaVrednost,Obradjen,Storniran);
     rac.sracun = new StavkaRacuna[TabSlogoviRacuna.getRowCount()];
      for(int i = 0; i< TabSlogoviRacuna.getRowCount(); i++)
        { rac.sracun[i] = new StavkaRacuna(rac);
         for (int j=0; j<TabSlogoviRacuna.getColumnCount(); j++)
          {  Object ob = TabSlogoviRacuna.getValueAt(i,j);
               rac.sracun[i].Napuni(j,ob);
          }
      } }
```

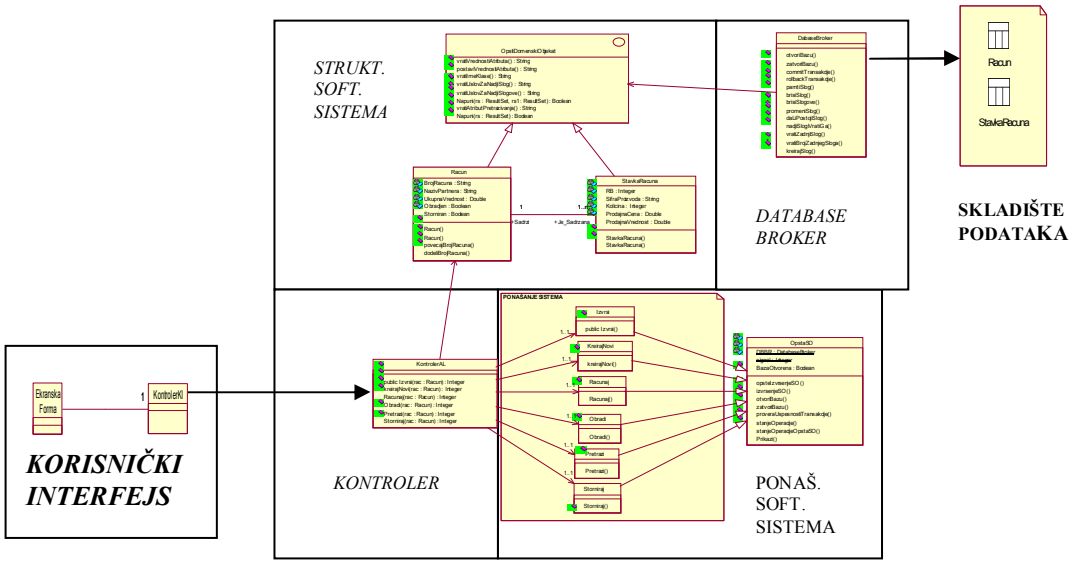
```
static void KonvertujObjekatRacunUGrafickeKomponente(Racun rac,JFormattedTextField BrojRacuna,JFormattedTextField
NazivPartnera,JFormattedTextField UkupnaVrednost,JCheckBox Obradjen,JCheckBox Storniran,JTable
TabSlogoviRacuna,DefaultTableModel DTM)
 { rac.Vrati(BrojRacuna,NazivPartnera,UkupnaVrednost,Obradjen,Storniran);
   DTM.setRowCount(rac.sracun.length);
   for(int i = 0; i< rac.sracun.length; i++)
    { for (int j=0; j<TabSlogoviRacuna.getColumnCount(); j++)
       {  System.out.println("Vrednost tabele: " + rac.sracun[i].Vrati(j));
         TabSlogoviRacuna.setValueAt(rac.sracun[i].Vrati(j),i,j);}
    }
 }
}
```

Nakon projektovanja korisničkog interfejsa dobija se sledeći dijagram klasa.

i sekvencni dijagram:



Korisnički interfejs u kontekstu arhitekture softverskog sistema se može predstaviti na sledeći način (Slika ASSKI):



Slika ASSKI: Arhitektura soft. sistema – korisnički interfejs

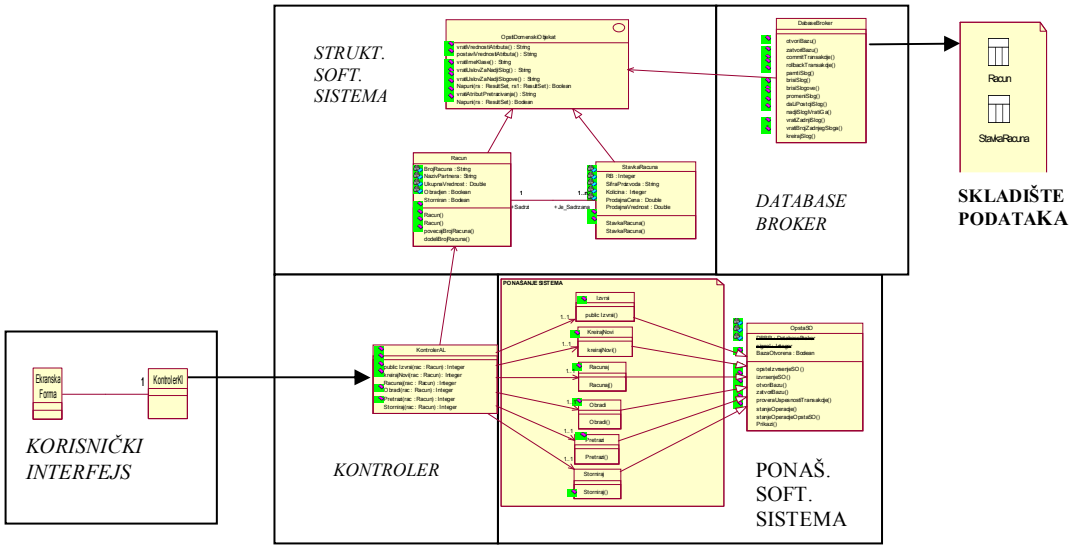
1.4 IMPLEMENTACIJA

U studijskom primeru prikazaćemo komponente koje smo dobili iz arhitekture softverskog sistema i redosled njihovog kompajliranja [JPRS].

SPIM5.1: Komponente

Komponente možemo dobiti iz klasa i interfejsa ili podsistema arhitekture softverskog sistema (Slika KM). Podsistemi arhitekture soft. sistema su:

1. Korisnički interfejs
2. Kontroler poslovne logike
3. Ponašanje softverskog sistema
4. Struktura softverskog sistema
5. Database broker
6. Skladište podataka



Slika KM: Komponente aplikacije

Na osnovu arhitekture soft. sistema dobili smo sledeće komponente:

- K1 – naziv komponente: *EkranskaForma.java*;
  Komponenta je izvedena iz klasa *EkranskaForma* i *DijalogBoksPrikazPoruke.*

- K2 – naziv komponente: *KontrolerKI.java*
  Komponenta je izvedena iz klase *KontrolerKI.*

- K3 – naziv komponente: *KontrolerPL.java*
  Komponenta je izvedena iz podsistema (klase) *KontrolerPL.*

- K4 – naziv komponente: *OpstaSO.java*
  Komponenta je izvedena iz klase *OpstaSO.*

- K5 – naziv komponente: *Pretrazi.java*
  Komponenta je izvedena iz klase *Pretrazi.*

- K6 – naziv komponente: *Izvrsi.java*
  Komponenta je izvedena iz klase *Izvrsi.*

- K7 – naziv komponente: *KreirajNovi.java*
  Komponenta je izvedena iz klase *KreirajNovi.*

- K8 – naziv komponente: *Racunaj.java*
  Komponenta je izvedena iz klase *Racunaj.*

- K9 – naziv komponente: *Storniraj.java*
  Komponenta je izvedena iz klase *Storniraj.*

- K10 – naziv komponente: *Obradi.java*
  Komponenta je izvedena iz klase *Obradi.*

- K11 – naziv komponente: *OpstiDomenskiObjekat.java*
  Komponenta je izvedena iz interfejsa *OpstiDomenskiObjekat.*

- K12 – naziv komponente: *DomenskiObjekti.java*
  Komponenta je izvedena iz klasa *Racun i StavkeRacuna.*

- K13 – naziv komponente: *DatabaseBroker.java*
  Komponenta je izvedena iz podsistema (klase) *DatabaseBroker.*

- K14 – naziv komponente: Racun.mdb
  Komponenta je izvedena iz tabela *Racun, StavkaRacuna*

Klase, interfejsi i podsistemi, iz kojih su napravljene komponente, se nakon pravljenja komponenti čuvaju u njima.

SPIM5.2: Redosled kompajliranja komponenti
Redosled kompajliranja može da se predstavi preko datoteke *start.bat* :

```
javac OpstiDomenskiObjekat.java
javac DatabaseBroker.java
javac DomenskiObjekti.java
javac OpstaSO.java
javac KreirajNovi.java
javac Izvrsi.java
javac Obradi.java
javac Pretrazi.java
javac Racunaj.java
javac Storniraj.java
javac KontrolerPL.java
javac KontrolerKI.java
javac EkranskaForma.java
```

Na kraju se izvršava komponenta EkranskaForma, koja pokreće izvršenje programa:
```
java EkranskaForma
```

1.5 TESTIRANJE

Testiranje, shodno arhitekturi softverskog sistema, može da se podeli u nekoliko nezavisnih jedinica testiranja. Nezavisne jedinice testiranja su softverske komponente koje smo dobili u fazi implementacije softverskog sistema.
Testiranje svake softverske komponente podrazumeva pravljenje:

a) Test slučajeva koji opisuju šta test treba da proveri,
b) test procedura koje opisuju kako će se izvršiti test i
c) test komponente koje treba da automatizuju test procedure ukoliko je to moguće.

Nakon testiranja softverskih komponenti vrši se njihova integracija. U tom smislu se prave testovi integracije softverskih komponenti.

Zadatak: Izvršiti testiranje nekih od postojećih softverskih komponenti i nakon toga izvšiti test njihove integracije.

2. Uzori (Paterni)

U ovoj glavi ćemo navesti osnovne definicije kojima se objašnjavaju uzori (paterni), zatim ćemo navesti poznate oblike predstavljanja uzora i na kraju ćemo dati klasifikaciju uzora.

2.1 Šta su uzori

Koreni uzora se nalaze u radovima [AC1-AC3] Krisofer Aleksandera (Christopher Alexander), koji se odnose na projektovanje arhitekture u građevinarstvu.
Mnogi termini, koji se koriste kod softverskih uzora, kao što su forces (sile), uzor (pattern), jezici uzora (pattern languages), i drugi dolaze od K. Aleksandera.
Aleksander je ukazao na primenu uzora u razvoju građevinske arhitekture. Uzori se u tom razvoju koriste u monotonom, progresivnom redosledu.
Aleksander je dao sledeću definiciju uzora [AC3]:

Svaki uzor je trodelno pravilo, koje uspostavlja relaciju između nekog problema, njegovog rešenja i njihovog konteksta.

*Uzor je u isto vreme i **stvar**, koja se dešava u stvarnosti, i **pravilo** koje govori **kako** se kreira navedena stvar i **kada** se ona može kreirati.*

Coplien za uzor kaže nešto slično[Cop1]:

Uzor je pravilo za građenje stvari, ali je ono istovremeno i sama stvar.

Aleksander je takođe pokušao da objasni uzore iz perspektive ljudkog mišljenja [AC3]: *Uzori u našem mišljenju su manje više, mentalne slike uzora u stvarnosti. Oni su apstraktna reprezentacija morfoloških pravila koja definišu uzore u stvarnosti.*
U našem mišljenju uzori su dinamički jer imaju neku silu. Pored toga uzori imaju osobinu generalnosti.

Ward Cunningham, kada u metafori objašnjava uzor, kaže da je uzor nešto što više liči na recept nego na plan [Cop1]: *Ja želim da napravim razliku između recepta i plana. Plan se može dobiti, inverznim postupkom, iz postojeće građevine, ali recept se ne može dobiti, inverznim postupkom, iz kolača.*

Aleksander je uočio uzore na osnovu struktura gradova i njihovih građevina. On je smatrao da dokumentovanje ovih uzora može pomoći ljudima u svakodnevnom životu. Uzor treba da prati odgovarajuća dokumentacija, kako bi se olakšalo nalaženje rešenje u sličnoj problemskoj situaciji. U tom smislu on metaforički kaže da je odnos između uzora i dokumenta isti kao odnos između poezije i literature.
Coplien kaže da uzori ne predstavljaju metodu projektovanja. Oni obuhvataju praksu koja se razvijala iz postojećih metoda razvoja softvera. U tom smislu on kaže da uzori nisu CASE alat koji po automatizmu može izvršiti neku transformaciju. Uzori su pre svega rezultat ljudske aktivnosti i upravo oni prave razliku između ljudske i kompjuterske inteligencije.
Proces nastajanja uzora, po Coplienu, ostao je prilično nejasan. On je rekao da mnogi uzori imaju svoju osnovu u radovima adaptera novih tehnologija ili sistema. Navedeni uzori su davali dobra rešenja za definisane probleme, ali ta rešenja nisu mogla da se povežu sa prvobitnim tehnološkim ili sistemskim rešenjima.
Dobro rešenje kod uzora ima dovoljno detalja koji govore šta uzor radi, ali to rešenje je istovremeno i dovoljno generalno da ukaže na širok kontekst mogućih problema koje ono zadovoljava.

2.2 Oblici predstavljanja uzora

Oblik predstavljanja uzora pomaže da se shvati: a) problem koji se razmatra, b) kontekst u kome se problem nalazi i c) objašnjenje rešenja problema.
Uzori se obično predstavljaju preko skupa sekcija. Zajedničke sekcije za različite načine predstavljanja uzora su:

- **Ime (Name)** – Veoma je važno da projektant izabere odgovarajuće ime za uzor iz dva razloga: a) ime treba da ukaže na suštinu samoga uzora i b) ime postaje sredstvo komunikacije (deo rečnika) između različitih projektanata i/ili razvojnih timova. Često se javlja dilema oko toga šta treba da bude ime uzora: imenica, glagol, imenična fraza,...itd kao i dilema da li ime treba da ukaže na problem, kontekst problema ili na rešenje. U tom smislu imena uzora su nastajala kao različite kombinacije navedenih dilema. Navešćemo neke od njih:

- Uzori koji su imeničke fraze a ukazuju na rešenje - Bridge (Most); GOF-ov uzor [GOF].
- Uzori koji su imeničke fraze a ukazuju na kontekst problema - A Place to Wait (Mesto za čekanje) - Aleksanderov uzor [AC2].
- Uzori koji su glagolske fraze a ukazuju na rešenje - Developing in Pairs (Razvoj u parovima) – Coplienov uzor [Cop2].
- Uzori koji su glagolske fraze a ukazuju na problem - Identify the Nouns (Identifikovanje imenica) – DeBruler-ov uzor [DeBr].

- **Svrha (Intent) -** Ova sekcija sadrži rečenicu ili frazu koja sumarno objašnjava šta uzor radi. Ona je prvi put upotrebljena kod GOF uzora [GOF].
- **Problem (Problem)** – Ova sekcija opisuje problem koji treba da bude rešen. Opis problema u većini slučajeva je prikazan u opštem obliku.
- **Kontekst (Context)** – Kontekst uzora ukazuje na okruženje u kome se dešava uzora. On treba da ukaže na promene u okruženju, koje mogu negativno da utiču na uzor u smislu njegove primene.
- **Sile (Forces) -** Coplien kaže: "Ako razumete sile u uzoru tada vi razumete problem (zato što vi razumete međudejstvo (sile) između elemenata problema) i rešenje (zato što vi razumete kako da izbalansirate te sile) …". Iz tog razloga sile su u središtu razmatranja uzora.
  Termin sile je nasleđen iz uzora vezanih za građevinsku arhitekturu. Kada se prave lukovi na zidu, tada treba da se balansiraju sile gravitacije sa silama strukture koje sačinjavaju zid. Balansiranjem sila se postiže snažna struktuiranost takvog sistema. U softverskom inženjerstvu termin sila se koristi pre svega figurativno, zato što kod programa ne postoje fizičke sile koje treba balansirati[22].
  Sile treba da prošire shvatanje problema ili preciznije rečeno, preko sila treba da se u potpunosti shvati (oceni) razmatrani problem. Aleksander kaže da :"Sile predstavljaju deo problema a ne izdvojenu celinu."
  Sile ukazuju na teškoće i ograničenja koje postoje u problemu.

- **Rešenje (Solution) -** Rešenje odgovarajućeg problema je dobro ako ima: a) dovoljno detalja koji objašnjavaju projektantu šta treba da radi i b) dovoljnu generalnu širinu da ukaže na širok kontekst problema. Neki uzori obezbeđuju samo parcijalno rešenje problema, ne balansirajući sile uzora, što predstavlja osnovu za pojavu novih uzora, koji treba da obezbede balansiranje sila uzora. Na ovaj način se dolazi do anti-uzora (anti-paterna). Kod anti-uzora rešenje nekog problema ima: a) veoma uzak kontekst problema i b) sile uzora koje nisu na pravi način balansirane. To praktično znači da male promene kod definicije problema mogu da učine da uzor bude neupotrebljiv.

- **Skica (Sketch) -** Aleksander je govorio da skica[23] uzora predstavlja suštinu (esenciju) uzora. U tom smislu on kaže [AC1]:
  "Ovi dijagrami, koje sam ja u mojim najnovijim radovima nazvao uzori, predstavljaju ključ procesa kreiranja oblika…, najveća snaga onoga što sam pisao se nalazi u snazi ovih dijagrama."

  Poinkar je ukazao na važnost skica kada se razmatraju metode ali i problem kod shvatanja metoda:
  "Sociolozi se bave sociološkim metodama. Psiholozi se bave psihološkim metodama… Osobe koje se bave metodama često ne mogu da shvate značaj dijagrama (skica) zato što su oni obsednuti detaljima metoda…"

  Jaku vezu između uzora i skice Aleksander je objasnio na sledeći način [AC3]:
  " Ako ne možete da nacrtate dijagram od nečega, onda vi nemate uzor."
  U softverskom inženjerstvu skice kod uzora pomažu projektantu da shvati vezu između delova razmatrane celine. Gamma i drugi autori [GOF] koriste OMT dijagrame, kojima predstavljaju strukturu rešenja za svaki uzor.

[22] Smatramo da sila u softverskom inženjerstvu ustvari predstavlja ponanašanje sistema. U tom smislu uzor treba da balansira strukturu i ponašanje sistema. Pod ponašanjem podrazumevamo sve dinamičke aspekte sistema, predstavljene preko operacija koje se izvršavaju nad strukturom sistema.
[23] To je po analogiji slično konceptualnom modelu koji se pravi u toku razvoja softverskog proizvoda.

- **Rezultujući kontekst (Resulting Context) -** Svaki uzor vrši transformaciju sistema iz jednog konteksta u novi kontekst. Rezultujući kontekst predstavlja okruženje rešenja i on predstavlja ulaz za uzore koji slede. Takođe rezultujući kontekst ukazuje na novi odnos između sila u uzoru kao i na probleme koji iz toga mogu da proisteknu.

Postoji za sada tri poznata oblika predstavljanja uzora[24]:

a) Aleksanderov oblik
b) GOF oblik
c) Coplienov oblik

2.2.1 Aleksanderov oblik

U radovima Aleksandera su dati izvorni oblici uzora. Sekcije (Ime,Svrha,Problem,Kontekst,…) kod Aleksandera nisu posebno naglašene.
Aleksander započinje opis uzora od slike koja prikazuje primer arhitekture razmatranog uzora. Nakon toga, svaki uzor sadrži paragraf koji opisuje kontekst problema. Zatim se razmatra problem, koji se sastoji iz glavne linije (headline) koja daje suštinu problema u jednoj ili dve rečenice i tela problema, u kome se detaljno objašnjava problem. Daju se iskustva vezana za razmatrani uzor, opisuje se validnost uzora, ukazuje se na različite načine primene uzora, itd.
Tada se navodi rešenje kao srce(glavni deo) uzora. Rešenje daje odgovor na postavljeni problem u datom kontekstu. Ovo rešenje je dato u obliku instrukcije koja govori šta treba da se radi da bi se izgradio uzor. Nakon toga daje se dijagram rešenja na kome su jasno naznačene glavne komponente rešenja. Na kraju se daje veza razmatranog uzora sa svim manjim uzorima koji ga sačinjavaju.

2.2.2 GOF oblik

GOF (Gang of Four) oblik *[GOF]* predstavljanja uzora sastoji se iz sledećih sekcija:

- *Ime uzora i njegova klasifikacija (Pattern name and classification)* – ime uzora opisuje suštinu uzora. Dobar naziv je veoma važan, jer postaje deo rečnika koji se koristi u razvoju programa.
- *Svrha (Intent)* – to je kratka rečenica koja daje odgovor na sledeće pitanja: Šta uzor projektovanja (design pattern) treba da radi? Koja je svrha uzora?
- *Takođe poznat kao (Also Known As)* – navedena sekcija daje nazive postojećih uzora, koji su u suštini isti ili veoma slični kao razmatrani uzor.
- *Motivacija (Motivation)* – problemi u projektovanju doveli su do pojave uzora projektovanja. U ovoj sekciji navodi se konkretna primena razmatranog uzora, odnosno motivi koji su doveli do njegove pojave.
- *Primenjivost (Applicability)* – ova sekcija objašnjava u kojim situacijama se uzor može primeniti. Ona takođe ukazuje na primere lošeg projektovanja (gde bi trebalo uzor primeniti) i objašnjava kako se prepoznaju mesta gde se javlja loše projektovanje.
- *Struktura (Structure)* – ova sekcija daje dijagram klasa koji predstavlja strukturu rešenja. Navedeni dijagram koristi notaciju zasnovanu na OMT-u (Object Modeling Technique) *[Rum]*.
- *Učesnici (Participants)* – daje se opis klasa i/ili objekata (učesnici rešenja) iz dijagrama klasa, odnosno opisuju se njihove obaveze i odgovornosti.
- *Saradnja (Collaboration)* – daje se objašnjenje kako učesnici međusobno sarađuju i kako izvršavaju svoje obaveze.
- *Posledice (Consequences)* – navedena sekcija opisuje sve posledice primene uzora. Ona takođe pokazuje koji elementi strukture uzora mogu da se menjaju nezavisno.
- *Implementacija (Implementation)* – ovde se navode predlozi tehnika koje treba koristiti kada se implementiraju uzori.
- *Jednostavan program (Sample code)* – dat je programski kod i njegovo objašnjenje. Navedeni program predstavlja rešenje problema. Programski jezici koji se koriste kod implementacije programa su C++ i SmallTalk[25].
- *Poznata korišćenja (Known uses)* – navode se primeri uzora koji postoje u realnim sistemima.
- Povezani sa uzorima (Related Patterns) – daju se svi oni uzori koji su na neki način povezani sa razmatranim uzorom.

[24] U toku rada pokazaćemo da postoji i naš oblik predstavljanja uzora koji ćemo nazvati BG-FON oblik predstavljanja uzora.
[25] U primerima koje ćemo dati u daljem tekstu, kada se detaljno budu objašnjavali GOF uzori, biće korišćen Java programski jezik.

2.2.3 Coplienov oblik

Coplienov oblik je veoma sličan Aleksanderevom obliku predstavljanja uzora. On se sastoji iz sledećih sekcija:

- *Ime uzora* (The pattern name) – uglavnom se koriste imenice za imena uzora, ali se u nekim slučajevima koriste i glagolske fraze.
- *Problem* (The problem) – obično se problem opisuje preko nekog pitanja.
- *Kontekst* (The context) – opisuje se kontekst u kome se može desiti problem i daje se predlog mogućih rešenja.
- *Sile* (The forces) – sile opisuju međudejstva između elemenata problema. One ukazuju na vezu između problema i različitih rešenja.
- *Rešenje* (The solution) – objašnjava se način rešavanja problema.
- *Obrazloženje* (A rationale) – u ovoj sekciji se objašnjava šta uzor radi i njegova istorija
- *Rezultujući kontekst* (Resulting context) – ovde se navode sile u uzoru koje su balansirane. Takođe se navode sile koje nisu balansirane (one predstavljaju preduslov za pojavu novih uzora).

2.3 Klasifikacija uzora

Uzori *[Cop1]* se mogu klasifikovati na sledeći način:

2.3.1 Tronivojski uzori

Postoje tri nivoa apstrakcije kojima može biti pridružen neki uzor. Za najniži nivo apstrakcije se vezuju idiomi(idioms). Za srednji nivo apstrakcije se vezuju uzori projektovanja (design patterns). Dok se za najviši nivo apstrakcije vezuju okviri (framework).

- **Idiomi** su uzori najnižeg nivoa koji zavise od specifične implementacione tehnologije, kao što je npr. programski jezik. Coplien je u tom smislu dao poseban doprinos shavatanju idioma *[Cop3]*.
- **Uzori projektovanja** su uzori koji su nezavisni od konkrente implementacione tehnologije. Oni su mikroarhitektura: oni sadrže strukturu koja može biti složena ali ne i dovoljno velika da bi se za nju reklo da je podsistem[26]. Glavni doprinos u shavataju uzora projektovanja dao je E. Gamma i drugi u knjizi Design Patterns *[GOF]*.
- **Okviri**[27] su uzori sistemskog nivoa. Oni su projektovani tako da sadrže kompletan kod[28] jedne od osnovnih funkcija sistema ili celog sistema, koji može biti proširen za konkretnu aplikaciju. To znači da okvir obezbeđuje opšte rešenje a različite aplikacije ga koriste i proširuju svojim specifičnostima. Jedan okvir može da sadrži druge uzore različitih nivoa apstrakcije (idiome, uzore projektovanja i okvire).

2.3.2 Aleksanderevo skaliranje

Pojam skaliranja kod Aleksandera *[AC3]* se odnosi na pravljenje generalnog jezika uzora iz specifičnih jezika uzora. U tom smislu on je ukazao na iste i slične uzore iz više jezika uzora koji bi mogli da predstave jedan zajednički generalni jezik uzora.

2.3.3 Drugi skalirajući pristupi

Osobe koje su se bavile organizacijom uzora, koristile su različite tehnike kako bi to postigle. Ono što je zajedničko za sve njih je korišćenje zajedničkih (kišobran) uzora (umbrella patterns). U toj organizaciji manji uzori su bili razdvojeni između sebe, ali su preko zajedničkih uzora mogli da uspostave vezu.

2.3.4 Anti-uzori

Oni uzori koji su se u praksi pokazali kao loši (ili ne rade ili delimično rade) nazivaju se anti-uzori. Na anti-uzore ukazali su u nezavisnim istraživanjima Sam Adams i Andrew Koenig *[RKo]*. U tim istraživanjima navedeni autori su obrazlagali i objašnjavali posledice loših rešenja kod anti-uzora.

2.3.5 Meta-uzori

U istraživanjima Wolfgang Pree-a *[Pree]* se prvi put uvodi pojam meta-uzori. On je generalizovao ključne strukture mnogih GOF uzora. Navedene strukture su predstavljale blokove (meta-uzore) iz kojih se mogu napraviti drugi uzori.

---

[26] Kada kažemo podsistem mislimo na jednu od osnovnih funkcija koje sačinjavaju jedan sistem(program).

[27] Panu Viljama za okvire kaže: " Okviri ukazuju na kolekciju konkretnih klasa, koje zajedno rade kako bi izvršili postavljeni parametrizirani zadatak".

[28] Navedeni kod pokriva okvir problema.

2.4. GOF uzori projektovanja [GOF]

Pregled

GOF uzori se koriste u fazi projektovanja softverskog proizvoda. Oni pomažu u imenovanju i opisu **generičkih rešenja**, koja mogu biti primenjena u različitim problemskim situacijama.

Kod razvoja softvera, uopšteno gledajući, prvo treba da se shvati i razume razmatrani problem (sistem). Zatim se vrši njegova analiza, da bi se na kraju vršilo njegovo projektovanje i implementacija. U toku projektovanja uočavaju se klase projektovanje. Ukoliko se želi da navedene klase budu **fleksibilne** (u smislu njihovog jednostavnog održavanja i nadogradnje), njih treba organizovati pomoću uzora projektovanja. Uzore treba koristiti i onda kada se želi **dinamička izmena funkcionalnosti** programa u toku njegovog izvršavanja.
Uzori projektovanja omogućavaju fleksibilnost klasa i dinimičku izmenu funkcionalnosti ali istovremeno oni povećavaju **složenost** sistema[29].
Kada projektujemo objektno-orijentisan sistem, uzori nam pomažu da lakše i brže shvatimo (koncipiramo) organizaciju klasa, koje mogu ostvariti funkcionalnost sistema. U tom smislu uzori nam pomažu da shvatimo **konceptualni** aspekt željene funkcionalnosti sistema **.**
Uzori na apstraktnom nivou koriste opštu, zajedniču sintaksu, odnosno jezik[30] koji omogućava projektantima softvera lakšu **komunikaciju**.

Uzori su podeljeni u 3 grupe:
- **Kreacioni** uzori pomažu da se izgradi sistem nezavisno od toga kako su objekti, kreirani, komponovani i reprezentovani.
- **Strukturni** uzori opisuju složene strukture međusobno povezanih klasa i objekata.
- **Uzori ponašanja** opisuju način na koji klase ili objekti sarađuju i raspoređuju odgovornosti.

U daljem tekstu ćemo objasniti neke od paterna sve tri grupe uzora.

2.4.1 Uzori za kreiranje objekata

Uzori za kreiranje objekata pomažu da se izgradi system nezavisno od toga kako su objekti, kreirani, komponovani i reprezentovani. Postoje sledeći uzori za kreiranje objekata:

1. ***Abstract Factory -*** Obezbeđuje interfejs za kreiranje familije povezanih ili zavisnih objekata bez specifikacije njihovih konkretnih klasa. (Te klase su poznate tek u vreme izvršavanja programa.)
2. ***Builder -*** Deli konstrukciju kompleksnog objekta od njegove reprezentacije, tako da isti konstrukcioni proces može da kreira različite reprezentacije.
3. ***Factory Method -*** Definiše interfejs za kreiranje objekata , ali omogućava podklasama da one donesu odluku kako će se kreirati objekti. Prenosi se nadležnost instanciranja objekata sa klase na podklase.
4. ***Prototype -*** Specificira vrste objekata koje će biti kreirane korišćenjem prototipske instance i kreira nove objekte kopiranjem navedenog prototipa.
5. ***Singleton -*** Obezbeđuje klasi samo jedno pojavljivanje i globalni pristup do nje.

U niže navedenom tekstu daćemo detalje vezane za navedene uzore

1 BUILDER UZOR

2

3 Definicija

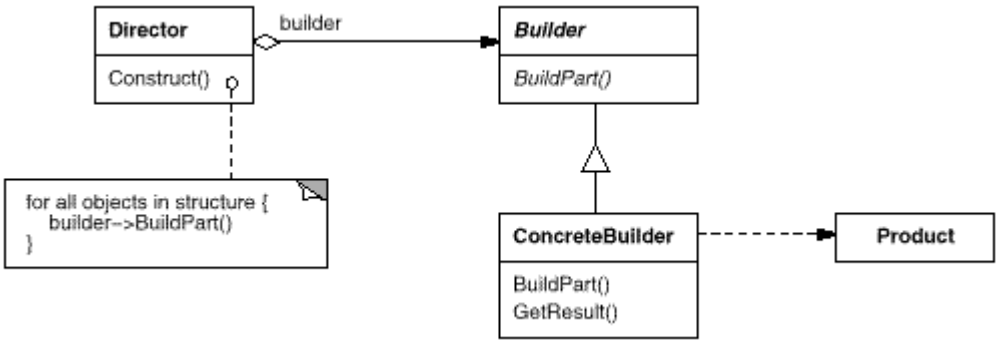
Deli konstrukciju kompleksnog objekta od njegove reprezentacije, tako da isti konstrukcioni proces može da kreira različite reprezentacije.

Primena

Koristite Builder uzor kada:
- *Algoritmi za kreiranje kompleksnih objekata treba da budu nezavisni od delova koji čine objekat i načina njihovog grupisanja.*
- *Konstrukcioni proces mora da dopusti različite reprezentacije objekta koji je konstruisan.*

[29] Mi ćemo kasnije formalno objasniti kada treba koristi uzore, u kontekstu navedene rečenice da oni povećavaju složenost sistema.
[30] Danas je taj jezik UML (Unified Modeling Language), koga ćemo detaljnije objasniti u prilogu teze.

Struktura



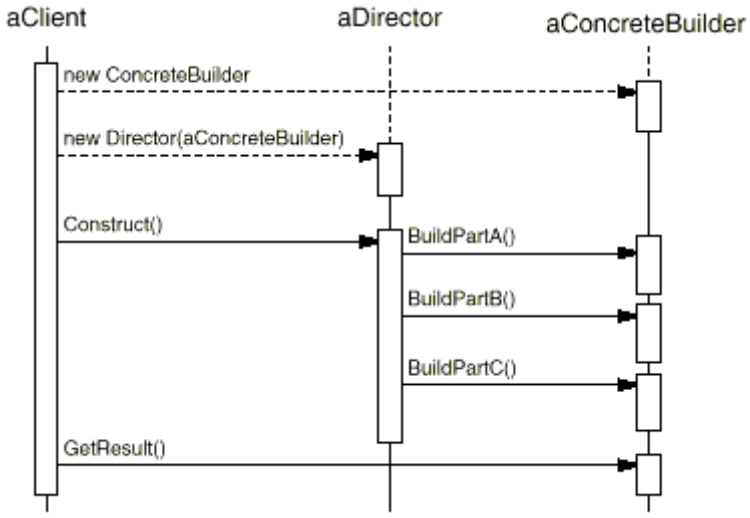
Učesnici

- ***Builder***
  Specificira apstraktni interfejs za kreiranje delova Proizvod.
- ***ConcreteBuilder*** *(CB1,CB2)*
  Konstruiše i grupiše delove proizvoda implementirajući Builder interfejs.
  Obezbeđuje interfejs za prihvatanje proizvoda.
- ***Director***
  Konstruiše objekat korišćenjem Builder interfejsa.
- ***Product***
  Reprezentuje kompleksan objekat koji se konstruiše. ConcreteBuilder izgrađuje internu reprezentaciju proizvoda i definiše proces kojim se grupišu delovi proizvoda.
  Uključuje klase koje definišu delove proizvoda, uključujući interfejs za grupisanje delova u finalni rezultat (proizvod).

Saradnja

- *Klijent kreira Director objekat i povezuje (konfiguriše) ga sa željenim Builder objektom.*
- *Direktor obaveštava builder objekat svaki put kada delovi proizvoda treba da budu izgrađeni.*
- *Builder obrađuje zahteve od direktora i dodaje delove do proizvoda.*
- *Klijent prihvata proizvod od builder objekta.*

Sledeći kolaboracioni dijagram ilustruje kako Builder i Director sarađuju sa klijentom.



Posledice
Navodimo neke od kljućnih posledica Builder uzora:

- *Moguće je menjati internu reprezentaciju proizvoda.*
- *Programski kod, kojim se vrši konstruisanje i reprezentacija proizvoda, je izolovan.*
- *Omogućava se dobra kontrola nad konstrukcionim procesom. Builder uzor konstruiše proizvod korak po korak pod direktnom kontrolom Director objekta.*

Implementacija

Obično, postoji jedna apstraktna Builder klasa koja definiše operacije za svaku komponentu (deo), koju Director klasa može da poželi da kreira. Operacije ne rade ništa podrazumevano. Svaka ConcreteBuilder klasa prekriva (overrides) operacije iz apstraktne Builder klase kojima se kreiraju komponente proizvoda.

Jednostavan program (Urađen u C++)

```
Class Builder
{ public:
   virtual BuildPartA();
   virtual BuildPartB();
   virtual CreateProduct();
   virtual Product *GetResult();
};

Class CB1 : public Builder
{  Product * pr;
   public:
   CreateProduct() {  pr= new Product;}
   BuildPartA(){PA1 *pa1 = new PA1; pr->AddA(pa1);}
   BuildPartB(){PB1 *pb1 = new PB1;pr->AddB(pb1);}
   Proizvod* GetResult() { return pr;}
};

Class CB2 : public Builder
{ Product * pr;
   public:
   CreateProduct () {  pr= new Proizvod;}
   BuildPartA(){PA2 *pa2 = new PA2; pr->AddA(pa2);}
   BuildPartB(){PB2 *pb2 = new PB2;pr->AddB(pb2);}
   Proizvod* GetResult() { return pr;}
};

Class Product
{ AbstractProductA * apa;  AbstractProductB * apb;
   public:
   AddA(AbstractProductA *apa1){ apa = apa1;}
   AddB(AbstractProductB * apb1){ apb =  apb1}};
Class Director   // konstruise objekat koristeci builder interfejs.
 { Builder * bd;
    Public:
   Director(Builder *bd1)
      { bd=bd1; }
   Proizvod* Construct()
       { bd->CreateProduct();
         bd->BuildPartA();   bd->BuildPartB();
         return bd->GetResult();
       }
};

main()
{   CB1 * kb1 = new CB1;     CB2 * kb2 = new CB2;
     Director *dr;     Product *pr;

    dr = new Director(kb1);
    dr.Construct();
    pr = kb1 ->GetResult(); // Klijent prihvata proizvod od konkretnog Builder objekta

     dr = new Director(kb2);
    dr.Construct();
     pr = kb2 ->GetResult();
}
```

2.4.2 Strukturni uzori

Strukturni paterni opisuju složene strukture međusobno povezanih klasa i objekata. Postoje sledeći strukturni uzori:

*1. **Adapter -** Adapter patern konvertuje interfejs neke klase u drugi interfejs koji klijent očekuje. Drugim rečima, isti prilagođava nekompatibilne interfejse.*

*2. **Bridge -** Odvaja (dekupluje) apstrakciju od njene implementacije tako da se one mogu menjati nezavisno.*

3. **Composite -** Objekti se komponuju u strukturu drveta kako bi predstavili deo-celina hijerarhiju. Composite uzor omogućava da se jednostavni objekti i komponovani objekti tretiraju jedinstveno.

*4. **Decorator -** Pridružuje odgovornost do objekta dinamički. Dekorator proširuje funkcionalnost objekta dinamički dodavanjem funkcionalnosti drugih objekata.*

*5. **Facade(fasada) -** Obezbeđuje jedinstven interfejs za skup interfejsa u podsistemu. Facade definiše interfejs visokog nivoa koji omogućava da se podsistem lakše koristi.*

*6. **Flyweight -** Koristi deljenje da efikasno podrži veliki broj objekata.*

*7. **Proxy -** Obezbeđuje surogat(**placeholder)** interfejsa drugog objekta u cilju kontrolisanog pristupa do njega.*

U niže navedenom tekstu daćemo detalje vezane za neke od navedenih uzora

BRIDGE UZOR *(Primer: Broker -> OpstaDomenskaKlasa)*

Definicija

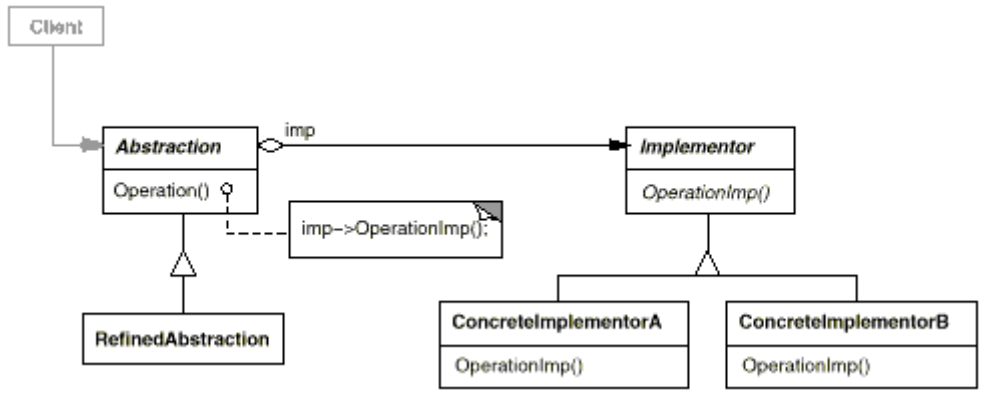
Odvaja (dekupluje) apstrakciju od njene implementacije tako da se one mogu menjati nezavisno.

Primena

Bridge uzor se koristi:

- *Kada se želi izbeći jaka veza između apstrakcije i njenih implementacija. To može biti u slučaju kada se implementacija bira u vreme izvršenja programa.*
- *Apstrakcije i njene implementacije mogu da se proširuju preko podklasa. U tom slučaju Bridge uzor omogućava kombinovanje različitih apstrakcija i implementacija koje se mogu proširivati nezavisno.*
- *Promene u implementaciji apstrakcije nemaju uticaj na klijente. To znači da programski kod klijenta ne mora da se rekompajlira kada se menja implementacija apstrakcije.*
- *Želi se sakriti od klijenta u potpunosti implementacija apstrakcije.*

Struktura



Učesnici

- ***Abstraction***
  Definiše interfejs apstrakcije.
  Čuva referencu na objekat tipa Implementor.
- ***RefinedAbstraction***
  Proširuje interfejs definisan klasom Abstraction.

- ***Implementor***
  Definiše interfejs za implementacione klase. Ovaj interfejs ne mora da odgovara interfejsu klase Abstraction. Obično Implementor interfejs obezbeđuje samo primitivne operacije a klasa Abstraction definiše operacije visokog nivoa koje su zasnovane na navedenim primitivnim operacijama.

- ***ConcreteImplementor (Cimp1,Cimp2)***
  Implementira interfejs klase Implementor .

Posledice

- *Razdvajanje interfejsa od implementacije.Implementacija se može izabrati u vreme izvršenja programa. To znači da je izbegnuta zavisnost između vremena kompajliranja i izbora klase koja će da implementira primitivu operaciju.*
- Povećanje proširivosti programa.
- *Apstrakcije i implementacije se mogu proširivati nezavisno.*
- *Sakrivanje implementacionih detalja od korisnika.*
- *Korisnik može da poziva primitivne operacije ali ne može da utiče na telo primitivnih operacija (implementacione detalje).*

Jednostavan program

```
Class Client
{ Abstraction * ap;
   public:
   Client(Abstraction * ap1) { ap = ap1;}
   mcl(){ap->Operation();}
};

Class Abstraction
{ Implementor * imp;
   public:
  Abstraction (Implementor * imp1) { imp = imp1;}
   Operation() { imp->OperationImp();}
};

Class Implemenor
{ public:
   virtual OperationImp();
};

Class Cimp1:public Implementor
{ public:
   OperationImp();
};

Class Cimp2:public Implementor
{ public:
   OperationImp();
};

main()
{ Client * cl;  Apstraction * ap;  Implementor imp;
  imp = new Cimp1;
  ap = new Abstrakcion(imp);
  cl = new Client(ap);
  cl->mcl();
  imp = new Cimp2;
  ap = new Abstraction(imp);
  cl = new Client(ap);
  cl->mcl();
}
}
```

Prvo se izabere konkretna implementacija. Zatim se implementacija poveže sa apstrakcijom. Na kraju se klijent povezuje sa apstrakcijom.
Ovaj patern podseća na top down metodu pomoću koje se korišćenjem koncepata apstrakcije i dekompozicije složenost jednog problema pojednostavljuje.

DECORATOR UZOR *(primer: UI klase kod Jave)*

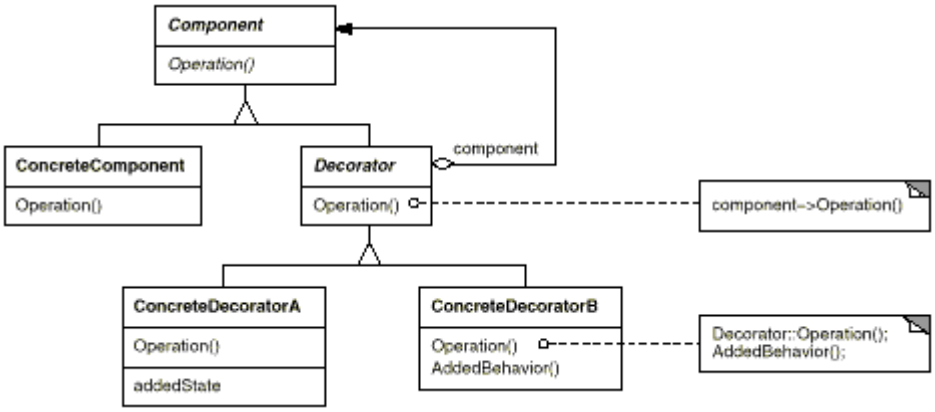
Definicija

Pridružuje odgovornost do objekta dinamički. Dekorator proširuje funkcionalnost objekta dinamički dodavanjem funkcionalnosti drugih objekata.

Primena

- *Kada se dodaje odgovornost do individualnog objekta dinamički i transparentno, bez uticaja na druge objekte.*
- *Kada se neke odgovornosti trebaju oduzeti objektu.*
- *Kada je proširivanje nasleđivanjem nepraktično, jer može dovesti do nekontrolisanog porasta broja klasa ili je nasleđivanje iz nekog razloga onemogućeno.*

Struktura



Učesnici

- ***Component***
  Definiše interfejs za objekte kojima se odgovornost dodaje dinamički.
- ***ConcreteComponent (CC)***
  Definiše konkretan objekat kome će biti dodata odgovornost dinamički.
- ***Decorator***
  Čuva referencu na konkretnu komponentu i omogućava dodavanje odgovornosti.
- ***ConcreteDecorator (Cdec1,Cdec2)***
  Dodaje odgovornost do komponente.

Jednostavan program

```
Class Component  // Interfejs za objekte kojima ce biti dodata dopunska odgovornost.
{
   public:
   virtual Operacija();
};

Class CC : public Component   // Konkretna komp. kojoj ce biti dodate dopunske odg.
{  public:
    Operacija();
};

Class Decorator : public Component
{ Component * cm;
   public:
   Decorator(Component * cm1) { cm = cm1;}
   Operacija(){cm->Operacija();}};
Class Cdec1 : public Decorator
{ public:
    Operacija(){Decorator::Operacija();
                  // operacije koje se pridruzuju konkretnoj komponenti
                  mkdec1(); …}
```

```
  mcdec1();
};

Class Cdec2 : public Decorator
{ public:
   Operacija(){Decorator::Operacija();
                  // operacije koje se pridruzuju konkretnoj komponenti
                   mcdec2(); …}
  mcdec2();
};

main()
{ CC * cc = new CC; //objekat kome se dodaju dopunska ponasanja
  Component * cm; // objekat koji ce sadrzavati ponasanje od kk + dopunska ponasanja.
  Cdec1 * cd1 = new Cdec1(cc);
  Cdec2 * cd2 = new Cdec2(cd1);
  // moglo je ovako da se napise: cd2 = new Cdec2(new Cdec1(new CC))
  cd2 -> Operacija();
 // Operacije se pozivaju rekurzivno po principu staka, prvo se poziva CC::Operacija pa
 // Cdec1::Operacija() i na kraju Cdec2::Operacija();
 // cd2 pokazuje na cd1 dok cd1 pokazuje na cc.
 }
```

FACADE UZOR (primer Kontroler kod realizacije Aplikacione logike)

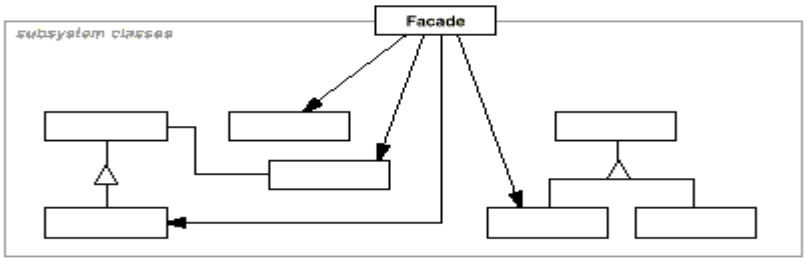
Definicija

Obezbeđuje jedinstven interfejs za skup interfejsa u podsistemu. Facade definiše interfejs visokog nivoa koji omogućava da se podsistem lakše koristi.

Primena

- *Kada treba obezbediti jednostavan interfejs za kompleksni podsistem.*
- *Kada postoji mnogo zavisnosti između klijenta i implementacionih klasa apstrakcije. Facade uzor razdvaja (dekupluje) podsistem od klijenta i drugih podsistema.*
- *Kada se žele podeliti podsistemi po slojevima.*

Struktura



Učesnici

- ***Facade***

  Zna koje su klase odgovorne za svaki od zahteva koji mu se prosleđuje.

  Prenosi odgovornost na klijenta da izvrši određeni zahtev.

- ***Podsistemske klase***

  Implementiraju podsistemske funkcionalnosti.

  Nemaju informacije ko je facade klasa.

Jednostavan program

```
Class Facade
 { A * a;
   B * b;
   C * c;
   Public:
   Facade (A * a1,B*b1, C*c1) { a = a1; b = b1; c = c1;}
   MF(){ a ->ma(); b->mb(); c->mc(); …}
```

```
   ma(){a->ma();}
   mb() {b->mb();}
   mc() {c->mc();}
}

Class A
 { public:
   ma();
   …};

Class B
 { public:
   mb();
   …};

Class C
 { public:
   mc();
   …};

main()
 { Facase * f;
   f = new Façade(new A,new B,new C);
   f->MF();    f->ma();    f->mb();    f->mc(); }
```

2.4.3 Uzori ponašanja

Uzori ponašanja opisuju način na koji klase ili objekti sarađuju i raspoređuju odgovornosti. Navodimo
sledeće uzore ponašanja:

1. ***Chain of Responsibility*** **-** Izbegava čvrsto povezivanje između pošiljaoca zahteva i njegovog primaoca, obezbeđujući lanac povezanih objekata, koji će da obrađuju zahtev sve dok se on ne obradi.
2. ***Command*** - Zahtev se učauruje kao objekat, što omogućava klijentima da parametrizuju različite zahteve. Navedenim pristupom podržava se izvršenje povratnih(undoable) operacija kao i oporavak podataka usled nasilnog prekida programa.
3. ***Interpreter*** **-** Dati jezik definiše reprezentaciju njegove gramatike, zajedno sa interpreterom koji koristi tu reprezentaciju za interpretaciju rečenica jezika.
4. ***Iterator*** - Obezbeđuje način da se pristupi elementima agregiranog objekta sekvencijalno bez navođenja njegove reprezentacije.
5. ***Mediator*** **-** Definiše objekat koji sadrži skup objekata koji su u međusobnoj interakciji. Interakcija između objekata može se nezavisno menjati u odnosu na druge interakcije. Pomoću medijatora se uspostavlja slaba veza između objekata.
6. ***Memento*** - Bez narušavanja učaurenja memento patern čuva interno stanje objekta tako da objekat može biti vraćen u to stanje kasnije.
7. ***Observer*** **-** Definiše jedan-više zavisnost između objekata, tako da promena stanja nekog objekta utiče automatski na promenu stanja svih drugih objekata koji su povezani sa njim.
8. ***State*** - Objekat može da menja svoje ponašanje usled promene njegovog internog stanja.
9. ***Strategy*** **-** Definiše familiju algoritama i obezbeđuje njihovu međuzavisnot. Strategy uzor omogućava promenu algoritma nezavisno od klijenta koji ga koristi.
10. ***Template Method*** **-** Definiše skelet algoritma neke operacije, prenoseći odgovornost izvršenja pojedinih koraka algoritma do podklasa. Template method omogućava podklasama da redefinišu neke od koraka algoritma bez promene algoritamske strukture.
11. ***Visitor*** **-** Predstavlja neku operaciju koja se izvršava na elementima objektne strukture. Visitor omogućava da se definiše nova operacija bez promene klasa onih elemenata nad kojima se izvršava operacija.

ITERATOR UZOR *(Iterator kod kolekcija u Javi)*

Definicija

Obezbeđuje način da se pristupi elementima agregiranog objekta sekvencijalno bez navođenja njegove reprezentacije.

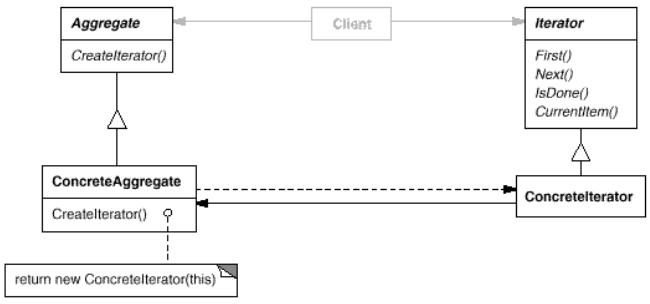
Takođe poznat kao

Cursor

Primena

- Kada se želi pristupiti sadržaju agregatnog objekta bez izlaganja njegove interne reprezentacije.
- Kada se želi obezbediti jedinstveni interfejs za pristup različitim agregatnim strukturama (podrška polimorfnoj iteraciji).

Struktura



Učesnici

- **Iterator**
  Definiše interfejs za pristupanje i kretanje kroz elemente agregiranog objekta.
- **ConcreteIterator**
  Implementira iterator interfejs.
  Čuva tekuću poziciju elementa u agregiranom objektu.
- **Aggregate**
  Definiše intefejs za kreiranje Iterator objekta.
- **ConcreteAggregate**
  Implementira CreateIterator() metodu kako bi vratio pojavljivanje ConcreteIterator klase.

Saradnja

- ConcreteIterator čuva tekući element (objekat) agregacije i može da odredi sledeći element tekućeg elementa.

Posledice

Korišćenje Iterator uzora ima tri važne posledice:

1. *On podržava različite varijante kretanja kroz agregirani objekat.* Kroz složeni agregatni objekat se može kretati na mnogo načina. Na primer, kroz drvo (agregirani objekat) koji se sastoji iz čvorova, može se kretati na više načina (prefiksni, sufiksni i infiksni prolaz). Iterator uzor olakšava promenu algoritma kretanja kroz agregirani objekat, tako što se menja jedan iterator pojavljivanja sa drugim iterator pojavljivanjem. Takođe se mogu definisati Iterator podklase, kako bi se podržala nova kretanja kroz agregirani objekat.

2. *Iterator uzor pojednostavljuje interfejs agregiranog objekta.* Iteratorov interfejs kretanja treba da bude sličan za različita kretanja kroz agregirani objekat.

3. *Više načina kretanja može biti pridruženo agregatnom objektu.* Iterator čuva tekuće kretanje kroz agregirani objekat. Zbog toga može imati više kretanja kroz agregirani objekat odjednom.

Jednostavan program

```
Class Aggregate
{  virtual CreateIterator();
    virtual Count();
    virtual Append();
    virtual Remove();
};

Class Iterator
{ virtual First();
   virtual Next();
   virtual isDone();
  virtual CurentItem();
};

Class ConcreteAggregate : public Aggregate
{ int a[30]; int numberelem;
  public:
  ConcreteAggregate () { numberelem = 0;}
  Iterator* CreateIterator() { return new ConcreteIterator(this);}
 Count() { return numberelem;}
  Append() { int temp; cin<<temp; aŠnumbereleme++] = temp;}
  Remove() { numberelem--;}
};

Class ConcreteIterator
{ ConcreteAggregate * ag;
   int counter;
   public:
   friend ConcreteAggregate;
```

```
  ConcreteIterator(ConcreteAggregate * ag1){ag=ag1;}
  First() { counter = 0}
  Next() { counter ++;}
  isDone() { return counter < ag.Count();}
 CurentItem(){if (isDone()) return ag.aScounter]; }
};



main()
{ ConcreteAggregate * ca = new ConcreteAggregate();
  ConcreteIterator * ci = ca.CreateIterator();
  ca.Append();
  ca.Append();
  ci.First();
  cout<<ci.CurrentItem();
  ci.Next();
  cout<<ci.CurrentItem();
   ca.Remove();
}
```

STATE UZOR

Definicija

Objekat može da menja svoje ponašanje usled promene njegovog internog stanja.

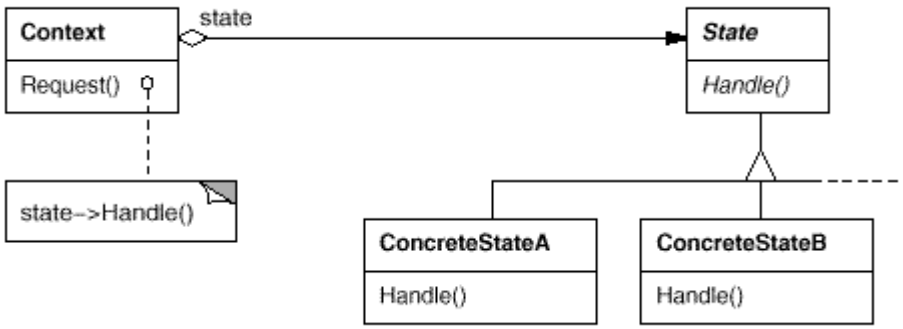
Poznat kao

Objekti stanja

Primena

State uzori se koriste u jednom od sledećih slučajeva:

- Ponašanje objekta zavisi od njegovog stanja. Objekat menja svoje ponašanje u realnom vremenu u zavisnosti od stanja u kome se nalazi.
- Operacije mogu da imaju dosta uslovnih izraza koji zavise od stanja objekta. Ova stanja se obično predstavljaju sa jednom ili više nabrajajućih konstanti. State uzor za svaku granu uslova (koja reaguje na neko stanje) pravi posebnu klasu. Ovim se omogućava da se stanje objekta tretira kao objekat koji može da se menja nezavisno od drugih objekata.



Učesnici

- **Context**
  - Definiše interfejs za klijenta.
  - Sadrži pojavljivanje ConcreteState podklase koja definiše tekuće stanje.
- **State**
  - Definiše interfejs za Context objekat, odnosno sadrži ponašanje koje se menja u zavisnosti od promene stanja kontekst objekta.
- **ConcreteState subclasses** (CSA,CSB)
  - Svaka podklasa (koja predstavlja stanja kontekst objekta) implementira ponašanje State klase.

Saradnja

- Context objekat delegira zahteve (zavisne od stanja) do odgovoarajućeg ConcreteState objekta.
- Context objekat može da prosledi sebe kao argument do State objekta kad obrađuje postavljeni zahtev. Time se omogućava State objektu pristup do Context objekta ako je to potrebno.
- Context objekat je osnovni interfejs za klijente. Klijent može da konfiguriše Context objekat sa State objektima. Pošto je Context objekat konfigurisan, njegovi klijenti ne mogu da rade direktno sa State objektima.
- Context ili ConcreteState objekti mogu da odrede kako se stanja menjaju i pod kojim uslovima (koji događaj utiče na tranziciju stanja).

Posledice

State uzor ima sledeće posledice:

1. On lokalizuje ponašanja koja zavise od stanja i deli ponašanje za različita stanja.
2. Tranzicije stanja se vide ekplicitno, kada state atribut Context objekta dobije neko konrektno stanje.
3. State objekti mogu biti deljeni, jer oni nemaju interna stanja koja se mogu menjati u vremenu već imaju samo ponašanje.

Implementacija

Kod implementacije State uzor javljaju se sledeće činjenice i dileme:

1. Ko definiše tranziciju stanja? State uzor ne definiše kriterijum za tranzicije stanja.Ako su kriterijumi nepromenjivi oni se u celini mogu implementirati u Context objektu. Generalno gledajući, fleksibilniji pristup je u omogućavanju State podklasama da same specificiraju njihovo sledeće stanje i događaje koji iniciraju tranzicije stanja. Ovo zahteva dodavanje interfejsa do Context objekta koji omogućava State objektima da postave stanje Context objekta eksplicitno.

Decentralizacija tranzicione logike omogućava laku promenu i proširivanje logike definisanjem novih State podklasa. Nedostatak decentralizacije se ogleda u neposrednoj povezananosti State podklasa, što utiče na implementacionu povezanost klasa.

2. *Tabelarni pristup upravljanja stanjima.* Cargill [TC] je opisao drugi način upravlja stanjima pomoću tabela. On je koristio tabele da preslika ulaze (događaje) i tranzicije stanja. Za svako stanje, tabela preslikava svaki mogući događaj u sledeće stanje. Ovakav pristup konvertuje programski kod uslovne logike (i pristup preko virtulanih metoda) u pristup upravljanja stanjima preko tabele.

Glavna prednost ovakvog pristupa se ogleda u sledećem: Promena tranzicionih kriterijuma (koji događaj prevodi objekat u novo stanje) se vrši promenom podataka u tabeli umesto promenom programskog koda.

Postoje sledeći nedostaci ovakvog pristupa:

- Poziv virtuelnih metoda je efikasniji od obrade podataka u tabeli.
- Postavljanjem tranzicione logike u uniformni tabelarni format tranzicioni kriterijumi su manje eksplicitni i zbog toga su teži za razumevanje.
- Tabelarni pristup upravljanja stanjima obuhvata stanja i njihove tranzicije, ali je on veoma nepogodan da izvrši određene dopunske radnje (npr., računanje nekog izraza) kod tranzicija.

Ključna razlika između state mašina kojima upravlja tabela i State uzora je sledeća: State uzori projektuju ponašanja koja su zavisna od specifičnog stanja, dok je tabelarni pristup pre svega usmeren na definisanje tranzicija stanja.

Jednostavan program

```
Class Context
{ State * st;
  public:
  Request(){st->Handle();}
};

Class State
{ public:
virtual Request();
};

Class CSA: public State
{
 public: Request();
};

Class CSB: public State
{
 public: Request();
};

main()
{ Context *con;
   if (uslov)
   con = new Context(new CSA());
else
  con = new Context(new CSB());
con->Request();}
```

STRATEGY UZOR

Definicija

Definiše familiju algoritama i obezbeđuje njihovu međuzavisnot. Strategy uzor omogućava promenu algoritma nezavisno od klijenta koji ga koristi.

Takođe poznat kao

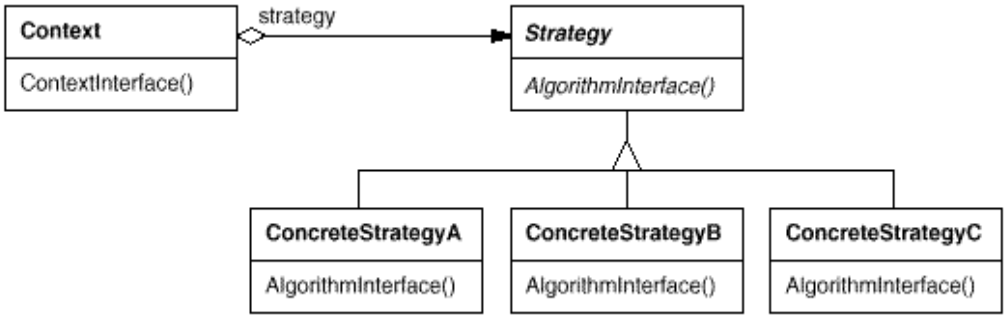
Policy

Primena

Strategy uzor se koristi kada:

- Mnogo povezanih klasa se razlikuje u njihovom ponašanju. Strategy obezbeđuje način da se klasa konfiguriše na jedno od tih ponašanja.
- Javi se potreba za različitim varijantama algoritma. Strategy uzor može se koristiti kada se navedene varijante implementiraju kao algoritamska hijerarhija klasa.
- Algoritam se koristi sa podacima koje klijent ne bi trebao da zna. Strategy uzor se koristi kako bi se izbegla složenost, algoritamsko-specifičnih struktura podataka.
- Klasa se definiše sa više ponašanja, koja se pojavljuju kao više uslovnih izraza u njihovim operacijama. Umesto mnogo uslova, povezani uslovi (conditional branches) se pomeraju u njihovu sopstvenu Strategy klasu.

Struktura



Učesnici

- **Strategy**
  Deklariše interfejs koji je zajednički za sve algoritme koji ga podržavaju. Context koristi ovaj interfejs da pozove algoritam definisan sa ConcreteStrategy klasom.
- **ConcreteStrategy**
  Implementira algoritam koji koristi Strategy interfejs.
- **Context**
  Context objekat je konfigurisana sa ConcretyStrategy objektom.
  On sadrži referencu na Strategy objekat.
  Može da definiše interfejs, koji omogućava Strategy objektu pristup, do njegovih podataka.

Saradnja

- Strategy i Context objekti sarađuju u implementaciji izabranog algoritma. Kada se algoritam pozove Context objekat može da prosledi sve podatke, koje zahteva algoritam, do strategy objekta. Takođe context može da prosledi samog sebe kao argument do Strategy objekta. Ovo omogućava povratni poziv context objekta od strategy objekta.
- Context objekat preusmerava zahteve njegovih klijenata do njegovog strategy objekta. Klijenti obično kreiraju i prosleđuju ConcreteStrategy objekat do Context objekta. Klijenti sarađuju isključuju sa Context objektom. Postoji familija ConcreteStrategy klasa koje klijent bira.

Jednostavan program

```
Class Context
 {Strategy * st;
   public:
   Context(Strategy *st) { st1=st;}
   ContextIterface() {st->Algorithm();}};

 Class Strategy
  { public:
     virtual Algorithm();…}
  };

Class ConcreteStrategyA : public Strategy  { Algorithm(); …};

Class ConcreteStrategyB : public Strategy  { Algorithm(); …};

Class ConcreteStrategyC : public Strategy  { Algorithm(); …};
 main()
{ ConcreteStrategyA  * csa = new ConcreteStrategyA;  ConcreteStrategyB  * csb = new ConcreteStrategyB;
ConcreteStrategyC  * csc = new ConcreteStrategyC;
  Strategy * st;
   // Izbor konkretne strategije
   …
   case 1 :…st =  csa;
```

```
  case 2: … st = csb;
  case 3: … st = csc;
  …
 Context * con = new Context (st);
}
```

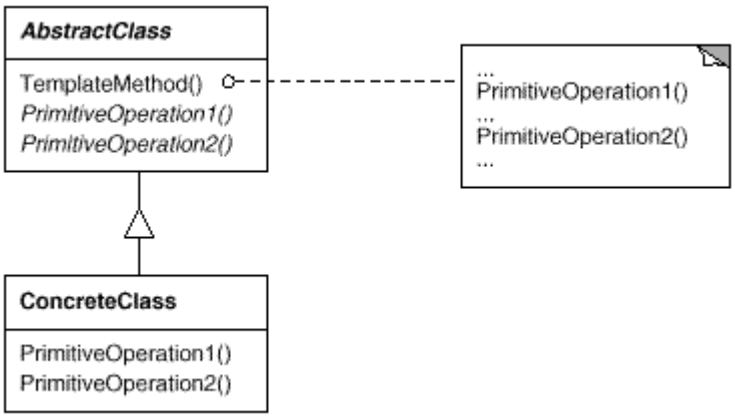
TEMPLATE METHOD UZOR ***(Primer: OpstaSO: opsteIzvrsenjeSo())***

Definicija
Definiše skelet algoritma neke operacije, prenoseći odgovornost izvršenja pojedinih koraka algoritma do podklasa. Template method omogućava podklasama da redefinišu neke od koraka algoritma bez promene algoritamske strukture.

Primena:
Template method uzor se koristi:

- da implementira generalno ponašanje algoritma, pri čemu podklase mogu da implementiraju specifična ponašanja algoritma.
- kada zajedničko ponašanje između klasa treba biti faktorizovano i lokalizovano u zajedničkoj klasi, kako bi se izbeglo dupliciranje koda. Dobar primer za to su dali Opdyke i Johnson objašnjavajuć koncept "refactoring to generaliza". Suština navedenog koncepta se ogleda u uočavanju razlika u ponašanju iste operacije u različitim situacijama (kontekstima). Za različita ponašanja se prave posebne operacije koje će Template method-a pozvati (jednu od njih) u zavisnosti od situacije.
- da kontroliše proširenje podklasa. Može se definisati template method-a koja poziva "hook" ("udica") operacije u specifičnim tačkama, čime se omogućava kontrolisano proširenje podklasa, samo u tim tačkama.

Struktura



Učesnici

- **AbstractClass**
  Definiše abstraktne primitivne operacije koje konkretna podklasa implementira.
  Implementira algoritam template method-e. Template method poziva primitivne operacije.
- **ConcreteClass**
  Implementira primitivne operacije koje opisuju specifična ponašanja podklasa.

Saradnja

ConcreteClass se oslanja na AbstractClass pri implementaciji apstraktnih operacija template method-e.

Posledice
Template method-e predstavljaju osnovnu tehniku kod ponovnog korišćenja raspoloživog koda. One su posebno važne kod biblioteka klasa, zbog toga što su one značajne za "faktorisanje[31]" ponašanja u bibliotekama klasa.
Template method je zasnovan na tkzv. "Holivudskom principu", koji glasi: "Don't call us, we'll call you[32]". Navedeni princip ukazuje na to da roditeljska klasa poziva operacije podklasa a ne obrnuto.

Template method-e pozivaju sledeće vrste operacija:

- Konkretne operacije od ConcreteClass.
- Konkretne operacije od AbstractClass (koje koriste podklase).
- Primitivne operacije, odnosno apstrakne operacije od AbstractClass.
- Factory metode (pogledati Faktory method uzor)
- "Hook" operacije, koje obezbeđuju podrazumevano ponašanje koje podklase mogu proširiti. Obično su "hook operacije" u apstraktnoj klasi prazne.

Veoma je važno da se zna, kod template method-a, koje su operacije "hook" (mogu biti prekrivene) a koje su operacije apstraktne (moraju biti prekrivene). Na osnovu toga podklase znaju koje operacije se mogu a koje se operacije moraju prekriti. Podklasa može da proširi ponašanje operacije roditeljske klase prekrivanjem operacije i pozivanjem roditeljske operacije eksplicitno.

```
Void DerivedClass::Operation()   // prekrivanje roditeljske operacije
{ ParentClass::Operation(); //  eksplicitno pozivanje roditeljske operacije
// DerivedClass prosiruje ponasanje roditeljske klase
}
```

Međutim u slučaju, kada ima više izvedenih klasa koje treba da prošire ponašanje, navedena operacija DerivedClass::Operation() mora da za svaku izvedenu klasu poziva ParentClass::Operation() pre proširenja ponašanja. Zbog toga ParentClass::Operation() postaje template method-a koja kontroliše kako se vrši proširenje podklasa. Osnovna ideja ovakvog pristupa je u tome da se iz navedene template method-e pozove hook operacija. Tada podklase mogu da prekriju ovu hook operaciju, odnosno da prošire ponašanje od ParentClass::Operation().

```
void ParentClass::Operation()
{ // Ponasanje roditeljske klase
    HookOperation();
}
```

HookOperation je prazna u ParentClass.

```
Void ParentClass::HookOperation() {}
```

Podklase prekrivaju HookOperation i proširuju ponašanje od ParentClass::Operation().

```
Void DerivedClass::HookOperation()
{ // prosirenje ponasanja izvedene klase}
```

Jednostavan kod:

```
Class AbstractClass
{ TemplateMethod()
   { …    PrimitiveOperation1();   PrimitiveOperation2();    …   }
     protected:
     PrimitiveOperation1(){…}
     PrimitiveOperation2(){…}
};

Class ConcreteClass1 : public AbstractClass
{    PrimitiveOperation1(){…}
     PrimitiveOperation2(){…}
};

Class ConcreteClass2: public AbstractClass
{    PrimitiveOperation1(){…}
     PrimitiveOperation2(){…}
};
```

[31] Faktorisanje predstavlja proces kojim se nalazi zajednički činilac od nekoliko ponašanja.
[32] "Nemojte zvati nas, mi ćemo pozvati vas"

```
Class ConcreteClass3 :  public AbstractClass
{      PrimitiveOperation1(){…}
       PrimitiveOperation2(){…}
};

main()
{ ConcreteClass1 * cc1 = new ConcreteClass1;
  ConcreteClass2 * cc2 = new ConcreteClass2;
  ConcreteClass3 * cc3 = new ConcreteClass3;
  // Za svaki od navedenih objekata (cc1,cc2,cc3) poziva se ista metoda
 // (TemplateMethod()) cije ponasanje zavisi od implementacije primitivnih operacija
// navedenih objekata.
   cc1->TemplateMethod();    cc2->TemplateMethod();   cc3->TemplateMethod();
}
```

2.5. GRASP[33] uzori projektovanja

Pregled
Kod Larmanove metode razvoja softvera [Larman] uzori se primenjuju u fazi projektovanja, u toku kreiranja dijagrama saradnje, kada se odgovornosti dodeljuju do objekata i kada se projektuje saradnja između objekata. Inače dijagrami saradnje se prave na osnovu sistemskih operacija, odnosno ugovora o sistemskim operacijama koji su definisani u fazi analize.

Larman na sledeći način definiše uzore:
Uzori su imenovani parovi problem/rešenja, koji definišu savete i principe kod dodeljivanja odgovornosti objektima.
U kontekstu navedene definicije navešćemo neke od GRASP uzora:

a) Kreator uzor
b) Uzor visoke kohezije (*High Cohesion*)
c) Uzor slabe povezanosti (*Low Coupling*)
d) Kontroler uzor
e) Uzor podele domena od prezentacije (Model-View Separation Pattern)
f) Izdavač-Pretplatnik (Publish-Subscribe) uzor
g) Uzor polimorfizma
h) Pure Fabrication uzor
i) Uzor indirekcije
j) Don't Talk to Strangers uzor

Sekcije koje prate svaki od navedenih uzora su sledeće:
- *Problem koji treba da reši uzor.*
- *Rešenja datog problema.*
- *Primer koji ilustruje rešenje problema.*

U nekim slučajevima je data veza razmatranog uzora sa postojećim uzorima.

KREATOR UZOR
Kreiranje objekata je jedna od najznačajnih aktivnosti u objektno-orijentisanom razvoju. Shodno tome, korisno je da postoje generalni principi za dodeljivanje odgovornosti za kreiranje. Oni pomažu da faza projektovanja razvije slabe veze između objekata, jasnoću, učurenje i ponovno korišćenje postojećeg softvera.

Problem: Ko treba biti odgovoran za kreiranje novih pojavljivanja klasa?

Rešenje: Klasi B se dodeljuje odgovornost za kreiranje objekta klase A, ako je zadovoljen jedan od sledećih uslova:
a) B agregira A objekat
b) B sadrži A objekat
c) B pamti A objekat
d) B koristi A objekat
e) B ima inicijalne podatke koji će biti prosleđeni do A kada se A kreira.

UZOR SLABE POVEZANOSTI ***(Primer: Klase koje realizuju SO su između sebe slabo povezane)***
Kuplovanje je mera vezanosti klase sa drugim klasama, kojom se utvrđuje koliko je jako klasa povezana sa drugim klasama, upoznata sa njima ili zavisna od njih.

Problem: Kako da se omogući slaba zavisnost između klasa kako bi se povećala mogućnost njihovog ponovnog korišćenja?
Rešenje: Dodeliti odgovornosti do klasa tako da povezanost između klasa bude što manja.

Klasa sa niskom (*low*) ili slabom (*weak*) povezanošću ne zavisi "previše mnogo" od drugih klasa. Termin "previše mnogo" biće objašnjen u kontekstu zavisnosti klasa:

[33] GRASP – General Responsibility Assignment Software Patterns

Klasa sa visokom (*high*) ili snažnom (*strong*) povezanošću zavisi mnogo od drugih klasa. Takva klasa nije poželjna, jer dovodi do sledećih problema:

- Promene povezanih klasa utiču na lokalnu promenu posmatrane klase.
- Teško je takvu klasu posmatrati izolovanu.
- Teško je ponovo koristiti takvu klasu jer ona zahteva prisustvo drugih klasa od kojih zavisi.

UZOR VISOKE KOHEZIVNOSTI *(Primer: Klase koje realizuju SO imaju veliku koheziju)*
Kohezija (preciznije rečeno funkcionalna kohezija) je mera kojom se utvrđuje koliko su jako odgovornosti klasa međusobno povezane. Klasa sa jako povezanim odgovornostima, koje izvršavaju jednostavne poslove, ima visoku koheziju. *Idealno bi bilo da se za jednu operaciju pravi jedna klasa.*

Problem: Kako upravljati složenošću klase?

Rešenje: Klasi se dodeljuju odgovornosti tako da kohezija klase ostaje visoka.

KONTROLER UZOR *(Primer: Kontroler kod aplikacione logike)*
Problem: Ko je odgovoran za upravljanje sistemskim događajima?

Rešenje: Klasa koja predstavlja :

- celokupni sistem (facade controller)
- celokupno poslovanje ili organizaciju (facade controller)
- nešto u stvarnosti što je aktivno (npr. uloga osobe) i što učestvuje u izvršenju nekog zadatka (role controller) ili klasi koja
- veštački upravlja svim sistemskim događajima jednog slučaja korišćenja (use case controller)

dodeljuju se odgovornosti za upravljanje sistemskim događajima .

Naglašavamo da jedna klasa kontroler upravlja svim sistemskim događajima jednog slučaja korišćenja. Takođe, navodimo da objekti iz korisničkog interfejsa (nivo prezentacije) prihvataju sistemske događaje, ali iste šalju do kontrolera da ih obradi. Kontroler koristi Command uzor iz GoF-a za obradu sistemskih događaja.
Sistemski događaj je događaj koji generiše spoljni učesnik. Ovim događajima su dodeljene sistemske operacije koje se izvršavaju kao odgovor na njih.. Kontroler nije objekat nivoa prezentacije. On je odgovoran za upravljanje sistemskim događajem. Isti definiše metode za sistemske operacije.

Prilikom projektovanja, navedene sistemske operacije ne izvršava Sistem već klase koje njemu pripadaju. Tako klase *Kasa, Račun* i *Kasirka* zadovoljavaju navedene uslove da mogu biti kontroleri. Prilikom analize i projektovanja pokazaće se da je *Kasa* najbolji kontroler. Bitno je da svi sistemski događaji budu u okviru jednog kontrolera za isti slučaj korišćenja zbog praćenja njegovog stanja.

UZOR PODELE DOMENA OD PREZENTACIJE (Model – View sep. pattern) *(Primer: Odnos između korisničkog interfejsa i kontrolera)*
Problem: Između domenskih i prezentacionih objekata treba da postoji što je moguće slabija veza, kako bi se domenski objekti mogli ponovo koristiti u drugim aplikacijama i kako bi se smanjio uticaj interfejsa na domenske objekte. Šta treba učiniti?

Rešenje: Definišu se domenske (model) klase koje nemaju direktnu povezanost ili vidljivost ka prezentacionim (pogled) klasama. Obrada podataka aplikacije i funkcionalnost se zatim razvijaju u domenskim a ne u prezentacionim klasama.

UZOR POLIMORFIZMA (*Primer: Interfejs OpstiDomenskiObjekat*)
Problem: Kako da se obrade alternative u zavisnosti od promene tipa? Kako da se kreiraju pluggable softverske komponente?
Alternative zasnovane na tipu. Uslovni izrazi u programa su podložni promenama. Ako je program projektovan korišćenjem if-then-else ili pomoću case uslovnog izraza, tada pojava novog ponašanja u

programu zahteva da se uslovna logika menja. Takav program je težak za održavanje zato što uvođenje novog ponašanja zahteva promene u programu na svim mestima gde postoji uslovna logika.
Pluggable softverske komponente. Posmatrajući komponente u klijent-server vezi može se primetiti da klijent koristi i menja (poziva drugu) server komponente bez promene njegove strukture i ponašanja?

Rešenje: Kada se povezane alternative ili ponašanja menjaju po tipu (klasi), tada treba dodeliti odgovornost za ponašanje do tipova korišćenjem polimorfnih operacija.

*UZOR INDIREKCIJE (*Primer: Komunikacija između poslovne logike i skladište podataka preko database brokera*)*

Problem

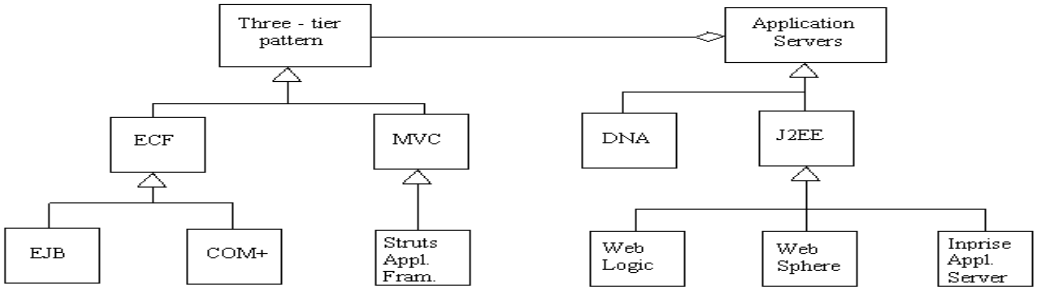
Kako da se izbegne direktna povezanost ili ostvari slaba povezanost između objekata u cilju njihovog ponovnog i jednostavnog korišćenja?

Rešenje

Dodeljuje se odgovornost do objekta (broker objekat) koji posreduje između drugih komponenti ili servisa tako da oni nisu direktno povezani. Objekat posrednik kreira indirekcije (posredne veze) između drugih komponenti ili servisa.

3. Arhitekture[34] distribucije objekata

Navodimo neke od postojećih arhitektura za razvoj distribuiranih aplikacija[Cir4], koje su zasnovane na tronivojskom uzoru (Slika ADSO).



Slika ADSO: Pregled tronivojskih arhitektura

Iz tronivojskog uzora izvedeni su:
a) ECF (Enterprise Component Framework) i
b) MVC (Model-View-Controller) uzori arhitektura za razvoj distribuiranih aplikacija.

Tronivojski uzor takođe predstavlja logičku osnovu iz koje su izvedeni savremeni aplikacioni serveri[35] (application servers).
Aplikacioni serveri su zasnovani na 2 osnovne arhitekture:
1. Microsoft-ovoj Windows DNA (*Distributed interNet Applications – Microsoft Transaction Service i Microsoft Internet Information Server*) arhitekturi.
2. Sun-ovoj J2EE (*Java 2 Platform Enterprise Edition*) arhitekturi.

Windows DNA predstavlja softverski paket, koji se koristi u razvoju distribuiranih aplikacija. Sa druge strane J2EE predstavlja specifikaciju, odnosno upustvo za razvijanje Enterprise aplikacionih servera. Navešćemo neke implementacije (aplikacione servere) navedene specifikacije:
a) Web Logic
b) Web Sphere-u
c) Inprise Application Server

3.1 ECF uzor

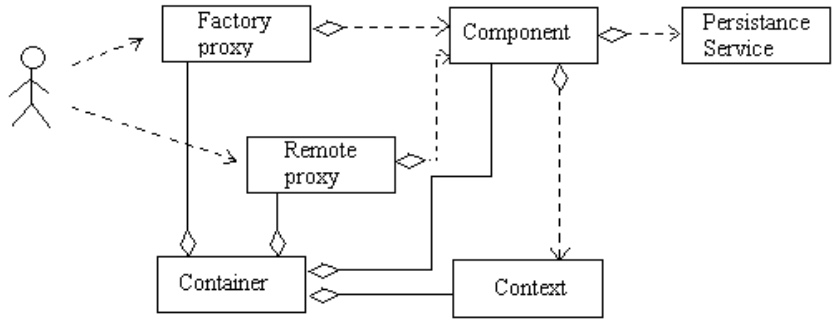
ECF uzor (Slika ADSO) je okvir (Framework[36]) za razvoj složenih (Enterprise) distribuiranih aplikacija koje su zasnovane na softverskim komponentama (Component), koje se mogu ponovo koristiti u novim problemskim situacijama.
ECF uzor (Slika ECF) sadrži sledeće elemente: Client, FactoryProxy, RemoteProxy, Context, Component, Container i PersistenceService.
Navedeni elementi imaju sledeće uloge:
Client postavlja zahtev za izvršenje operacije nad Component elementom.
Proxy element, presreće klijentski zahtev i i izvršava operaciju nad Component elementom. FactoryProxy je zadužen da obezbedi sledeće operacije: create(), find() i remove(), dok je RemoteProxy zadužen da obezbedi ostale operacije koje poziva klijent.
Context element postoji za svaki component element i on čuva njegov kontekst, kao što je: stanje transakcije, perzistentnost, sigurnost,…

[34] Kada prof. Lazarević, u radu [Laz5], objašnjava realizaciju sistema on kaže:
Realizacija sistema je značajan korak u projektovanju softvera koji treba da odgovori na pitanje kako izabrati model (ili čak i teoriju) za prikaz ponašanja sistema u prostoru stanja i kako prikazati konkretan sistem u datom modelu.
Smatramo da arhitekture distribucije objekata predstavljaju model za prikaz ponašanja sistema, što će detaljno biti prikazano u studijskom primeru koji prati projektovanje.

[35] Aplikacioni serveri su sredstva za realizaciju arhitektura distribucije objekata.
Svaki aplikacioni server sastoji se iz tri osnovna dela:
a) deo za komunikaciju sa klijentom
b) deo za komunikaciju sa nekim skladištem podataka (baza podataka ili datotečni sistem)
c) deo koji sadržu poslovnu logiku.

[36] Okvir se definiše kao skelet neke aplikacije koju može kastomizirati projektant softvera. Pod kastomizacijom se podrazumeva ugrađivanje specifičnosti aplikacije u dati okvir.

Container element obuhvata (agregira) sve elemente ECF uzora osim Client i PersistenceService elemenata. On obezbeđuje run-time okruženje na kome se izvršavaju razni servisi distribuirane obrade aplikacija, kao što su: međuprocesna komunikacija, sigurnost, perzistentnost i transakcije.
Component element, kada želi da obezbedi svoju perzistentnost[37] poziva PersistanceService element koji je za to zadužen.



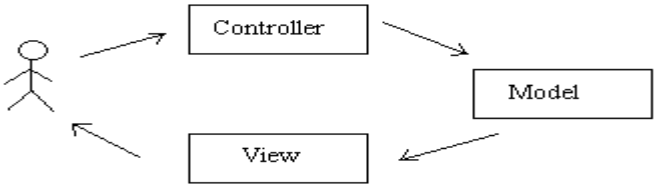
Slika ECF: ECF uzor

Iz ECF uzora su izvedene EJB (Enterprise JavaBeans) i COM+ arhitektura.

3.2 MVC uzor

MVC uzor (Slika MVC) je okvir za razvoj složenih Web aplikacija.
On sadrži sledeće elemente:

- View – obezbeđuje korisniku interfejs (ekransku formu) preko koje će korisnik da komunicira sa aplikacijom. Takođe on šalje korisniku razne izveštaje, koji se dobijaju iz modela.
- Controller – prihvata zahtev od klijenta za izvršenje operacije. Nakon toga poziva operaciju koja je definisana u modelu i kontroliše njeno izvršavanje. Rezultat izvršene operacije, on dalje preko view-a nazad šalje do klijenta.
- Model – sadrži strukturu poslovnog sistema i njene operacije[38].



Slika MVC: MVC uzor

Jedna od realizacija MVC uzora jeste *Struts Application Framework,* kod koga elementi MVC uzora imaju sledeće uloge:

- **View** – Obično sadrži JSP strane koje se procesiraju na nekom Web serveru (npr. Tomcat 3.2 Web server). Rezultat tog procesa su HTML dokumenti koji se šalju korisniku (browser-u).

- **Controller** – Obično se realizuje preko Java servleta ili Java bean-ova koji su zaduženi da obezbede kontrolu i kordinaciju izvršenja postavljenih zahteva. Oni dalje prosleđuju zahtev do EJB (Enterprise Java Beans-a) objekata[39]. Rezultati izvršenja operacija nad EJB objektima se preko Web servera nazad prosleđuju do korisnika.

- **Model** – Struktura se obično realizuje preko Entity Bean-a, dok se operacije odnosno poslovna pravila realizuju ili preko SessionBean-a ili preko Entity Bean-a.

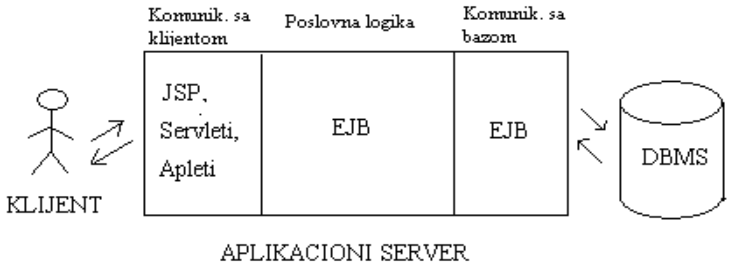
[37] Pošto je Component objekat predstavljen u opštem smislu, podrazumeva se da je za njegove sastavne delove (objekte) potrebno obezbediti perzistentnost.
[38] To je ustvari dijagram klasa strukture, koji je izveden iz domenskog modela.
[39] Postoje dve vrste EJB-a: Session Bean i EntityBean.

3.3 J2EE arhitektura

Kao što smo već rekli J2EE predstavlja specifikaciju, odnosno upustvo za razvijanje Enterprise aplikacionih servera.
Osnovu J2EE arhitekture čine:

a) Servleti *(servlets)*
b) JSP strane *(Java Server Pages) i*
c) EJB-ovi *(Enterprise Java Beans)*

Uzimajući u obzir tronivojski uzor, JSP, servleti i apleti su zaduženi za komunikaciju sa klijentom (prezentacioni nivo), dok su EJB-ovi zaduženi za onaj deo poslovne logike aplikacije, koja nije ugrađena u aplikacioni server, kao i za komunikaciju sa bazom (logiku pristupa podacima) (Slika J2EE).

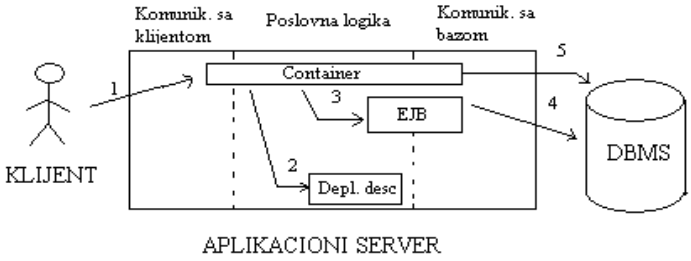


Slika J2EE : Raspodela odgovornosti elemenata J2EE arhitekture

Aplikacioni server sastoji se iz sledećih elemenata:

a) Home i Remote interface-a,
b) Context-a,
c) EJB-ova i
d) Container-a, koji agregira navedene elemente (EJB Home i Remote interface, Context i EJB elemente).

Scenarijo komunikacije (Slika KAS) između klijenta i aplikacionog servera je sledeći: Klijent poziva metodu Remote ili Home interface-a EJB-a. Container element presreće poziv (1) klijenta i proverava (2) da li je poziv validan (da li je u skladu sa semantikom koju je definisao deployment descriptor[40]). Ukoliko je poziv validan on ga dalje usmerava ka odgovarajućem EJB-u (3), koji kasnije komunicira sa bazom podataka. Komunikacija sa bazom može biti izvršena i preko container elementa (5), o čemu ćemo kasnije pričati kada budemo objašnjavali perzistentnost EJB-ova.



Slika KAS: Scenarijo komunikacije klijenta i aplikacionog servera

Container element obezbeđuje okruženje na kome se izvršavaju EJB-ovi. On kontroliše sve aspekte izvršenja jednog EJB-a. U tom smislu on obezbeđuje sledeće servise:

a) Registrovanje EJB-ova prilikom njihovog kreiranja i punjenja.
b) Upravljanje zivotnim ciklusom EJB-a.
c) Provera zastite i kontrola pristupa EJB-ovima.
d) Serijalizacija poziva metoda EJB-ova.
e) Kordinacija transakcija EJB-ova.
f) Upravljanje perzistentnoscu EJB-ova.
g) Pracenje kontekstnih informacija svakog EJB-a u vreme izvršenja programa.

[40] Deployment descriptor je XML datoteka koja specificira osobine EJB-a koje se odnose na rad sa transakcijama, zaštitu, perzistentnost,…

Context element se pravi za svaki EJB. On u sebi čuva informacije o klijentu, kontejneru i o samom EJB-u. EJB-ovi koriste ove informacije u obradi zahteva klijenta.

EJB (Enterprise Java Bean) – je u širem smislu arhitektura za razvoj složenih (Enterprise) distribuiranih aplikacija koje su zasnovane na softverskim komponentama (Java Bean-sima)[41]. Shodno tome EJB u užem smislu predstavlja softversku komponentu:

- koja se sastoji iz tri dela:

1. EJBHome interfejsa koji je odgovoran da specificira operacije: create(), find() i remove(),
2. EJBObject interfejsa koji je odgovoran da specificira ostale operacije koje zahteva klijent,
3. Klase (bean-a)[42] koja implementira EJB interfejse.

- i koja je odgovorna da obezbedi transakcione i perzistentne servise.

Postoje dve vrste EJB-ova:

1) Session bean i
2) Entity bean

Session bean je odgovoran:

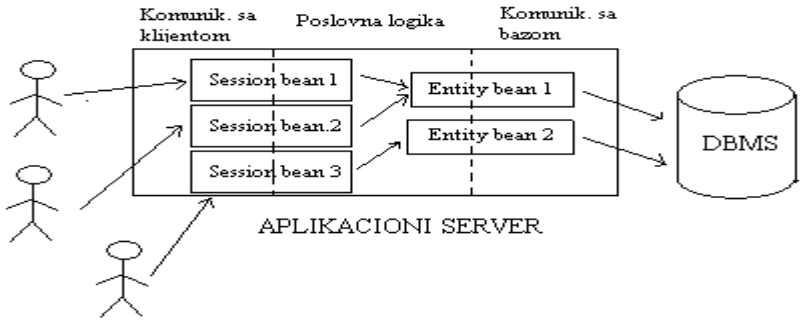
a) za komunikaciju između klijenta i servera i za
b) poslovna pravila (ponašanje) aplikacije

Entity bean je odgovoran:

a) da predstavi poslovne podatke (strukturu) aplikacije
b) i da komunicira sa bazom podataka.

Postoji sledeće poređenje između session i entity bean-a (Slika SEB):

	Session bean	**Entity bean**
Odgovornost	Izvršava zadatak ili proces za klijenta	Predstavlja poslovne objekte
Deljeni pristup	Jedna instanca po klijentu	Deljena instanca za više klijenata
Perzistentnost	Nije perzistentan – kada klijent završi rad sa binom isti nije više na raspolaganju	Perzistentan je – po završetku rada EJB stanje objekta je sačuvano u bazi.



Slika SEB: Odnos između sesion i entity bean-a

Iz navedenog izlaganja možemo da primetimo da su za transakcione i perzistentne servise zaduženi i Container i Entity bean. U tom smislu postoji dva pristupa vezanih za perzistentnost Entity bean-a:

1. *BMP (Bean Managed Persistent) Entity Bean* – perzistentnost Entity bean-a se implementira unutar Entity Bean klasa.
2. *CMP (Container managed Persistent) Entity Bean* – perzistentnost Entity bean-a se implementira unutar Container elemenata.

Navešćemo detaljnu specifikaciju interfejsa za session i entity bean-ove , jer ćemo se na iste pozivati kada budemo objašnjavali pristup koji smo koristili u fazi projektovanja i implementacije našeg studijskog primera.

[41] Podsećamo da je EJB arhitektura izvedena iz ECF arhitekture.

[42] Klijenti nikada ne komuniciraju sa klasama koje implementiraju EJB interfejs, već to isključivo rade preko EJB interfejsa.

Svaki session bean, kod EJB arhitekture, mora da implementira interfejs *java.ejb.SessionBean*.

```
public interface javax.ejb.SessionBean extends javax.ejb.EnterpriseBean
{ // Inicijalizuje sesion bean.
   public abstract void  ejbCreate(...) throws java.rmi.RemoteException;

  // Povezuje bean sa njegovim okruženjem.
  public abstract void  setSessionContext(javax.ejb.SessionContext) throws java.rmi.RemoteException;

// Prebacuje bean u privremeno skladište (file sistem ili bazu podataka).
  public abstract void  ejbPassivate() throws java.rmi.RemoteException;

// Prebacuje bean iz skladišta u memoriju.
 public abstract void  ejbActivate() throws java.rmi.RemoteException;

// Uklanja bean iz container-a.
 public abstract void  ejbRemove() throws java.rmi.RemoteException;
}
```

Svaki entity bean, kod EJB arhitekture, mora da implementira interfejs *java.ejb.EntityBean.*

```
public interfacejavax.ejb.EntityBean implements javax.ejb.EnterpriseBean
{ // Inicijalizuje Entity bean.
 public abstract Object ejbCreate(...);

// Ova metoda se poziva nakon ejbCreate() metode. Pomoću nje referenca od kreiranog bean-a se
// prosleđuje do drugih bean-ova, kako bi isti znali gde se nalazi kreirani bean..
 public abstract void ejbPostCreate();

// Container poziva ovu metodu da poveže bean sa njegovim kontekstom (okruženjem).
 public abstract void setEntityContext();

// Pre nego što se uništi bean, poziva se ova metoda koja prekida veze bean-a sa njegovim
// okruženjem.
 public abstract void unsetEntityContext();

// Prebacivanje bean-a u privremeno skladište (file sistem ili bazu podataka).
 public abstract void ejbPassivate();

// Prebacuje bean iz skladišta u memoriju. Ne prebacuje podatke bean-a jer je za to odgovorna metoda
// ejbLoad()
 public abstract void ejbActivate();

// Uklanja bean iz container-a.
 public abstract void ejbRemove();
// Pamćenje bean-a u perzistentno skladište (file sistem ili baza podataka).
 public abstract void ejbStore();

//  Puni podatke od bean-a iz privremenog skladišta u memoriju.
 public abstract void ejbLoad();

// Traži jedan EJB objekat ili njihovu kolekciju iz perzistentnog skladišta.
public abstract < . . . > ejbFind < . . . > (. . . );
}
```

4. Jedinstveni proces razvoja softvera

Jedinstveni proces [JPRS] je softverski razvojni proces koji se sastoji iz skupa aktivnosti potrebnih da se transformišu korisnički zahtevi u softverski sistem. On predstavlja okvir generičkog procesa koji može biti specijalizovan za veliku klasu softverskih sistema. Jedinstveni proces je zasnovan na komponentama, što znači da je softverski sistem izgrađen od softverskih komponenti koje su između sebe povezane dobro definisanim interfejsom.

4.1 Osnovne osobine i struktura JPRS

Osobine JPRS

JPRS ima sledeće osobine:

a) Sa njim se upravlja na osnovu slučaja-korišćenja (use-case driven).
b) Usmeravan je arhitekturom (architecture-centric).
c) Sastoji se od iteracija, koje kao rezultat imaju povećanje (inkrement) svojstva sistema .

U daljem tekstu dajemo detalje navedenih osobina

a) use-case driven

Slučaj korišćenja (SK) je deo funkcionalnosti sistema koji korisniku daje neki rezultat (vrednost). To znači da SK opisuju funkcionalne zahteve. Svi SK zajedno čine model SK koji opisuje kompletnu funkcionalnost sistema. Model SK odgovara kod specifikacije sistema na sledeće pitanje:
Šta sistem treba da radi za svakog korisnika.

za razliku od tradicionalne funkcionalne specifikacije koja je odgovarala na pitanje:
Šta sistem treba da radi?

To znači da su potrebe korisnika u osnovi specifikacije sistema.

SK istovremeno iniciraju i povezuju kompletan proces razvoja softverskog sistema. To znači da se pomoću SK upravlja analizom, projektovanjem, implementacijom i testiranjem softverskog proizvoda.
Slučajevi korišćenja (SK) se razvijaju zajedno sa sistemskom arhitekturom. SK upravljaju sistemskom arhitekturom dok sistemska arhitektura utiče na selekciju SK.

b) architecture-centric [43]

Arhitektura softverskog sistema(ASS) obuhvata najvažnije statičke i dinamičke aspekte sistema. ASS pored toga obuhvata:

- platformu na kojoj se izvršava softver (kompjuterska arhitektura, operativni sistem, DBMS, protokoli za mrežnu komunikaciju),
- raspoložive komponente koje se mogu ponovo koristiti(npr: okvir za grafički korisnički interfejs),
- sistem nasleđa (legacy system),
- nefunkcionalne zahteve (performanse, prenosivost,...).

Veza između SK i arhitekture se može objasniti na sledeći način:
Svaki proizvod ima funkciju i oblik(form) koji moraju biti između sebe povezani. Funkcija korespondira sa SK, dok oblik korespondira sa arhitekturom. SK moraju, kada se realizuju, da se ugrade u arhitekturu. Arhitektura mora da obezbedi prostor da se u nju ugrade svi sadašnji a po mogućnosti i budući SK. Arhitektura i SK moraju da se razvijaju paralelno.

Kod razvoja arhitekture prvo se kreira onaj deo arhitekture (platforma) koji je nezavisan od SK. Zatim se arhitektura razvija zajedno sa SK koji predstavljaju ključne funkcije razmatranog sistema.

[43] U kontekstu životnog ciklusa softvera, znači da se sistemska arhitektura u toku razvoja softvera koristi kao glavni artifakt za konceptualizaciju, konstrukciju, upravljanje i razrađivanje sistema.

c) iterative and incremental

Razvoj složenih softverskog proizvoda može da potraje više meseci ili više godina. Razvoj se sastoji iz više delova ili mini projekata. Svaki mini projekat je iteracija koja rezultuje inkrementom. Iteracija mora da se kontroliše kako bi se izvršila u planirano vreme. Mini projekat prolazi kroz analizu, projektovanje, implementaciju i testiranje. Mini projekat odnosno iteracija može biti vezana za jedan slučaj korišćenja ili za jednu složenu sistemsku operaciju nekog slučaja korišćenja.

Struktura JPRS

JPRS se sastoji iz skupa ciklusa, koji određuju "vek trajanja" softverskog sistema. Kao rezultat izvršenog ciklusa se dobija release (softverski proizvod) sistema. Svaki ciklus se sastoji iz četiri faze: početak (inception), razvoj – usavršavanje (elaboration), građenja (construction) i prelaz (transition)[44]. Ukoliko se želi da softverski proces bude efikasan, potrebno je definisati sekvencu jasno definisanih graničnih tačaka (milestones), u kojima neka faza razvoja softtvera prelazi u sledeću fazu. U svakoj fazi se prolazi kroz seriju kontrolisanih iteracija (mini projekata). Rezultat iteracije(**it**) je interni release(**ir**) (build), koji predstavlja inkrement za sistem. Iteracija prolazi kroz sve radne tokove razvoja softverskog proizvoda: prikupljanja zahteva od korisnika, analizu, projektovanje, implementaciju i testiranje.

Ciklus													
Faze / Radni tokovi	Početak			Razvoj			Građenje			Prelaz			
Prikupljanje zahteva	♦	♦	♦	♦♦	♦♦	♦							
Analiza				♦♦	♦♦	♦							
Projektovanja					♦	♦♦	♦♦						
Implementacija					♦	♦♦	♦♦	♦♦	♦♦	♦			
Testiranje	♦	♦	♦	♦	♦	♦	♦	♦	♦♦	♦♦	♦		
	it 1	it 2	it n	it n+1	it n+2	it m	it m+1	it m+2	it k	it k+1	it k+2	it p	
	ir 1 ●	ir 2	ir n ●	ir n+1	ir n+2	ir m ●	ir m+1	ir m+2	ir k ●	ir k+1	ir k+2	it p ●	Release

Milestones - ●

4.2 Faze JPRS

JPRS sastoji se iz četiri faze: početak (inception), razvoj – usavršavanje (elaboration), građenja (construction) i prelaz (transition).

- Početna faza definiše opseg projekta i viziju krajnjeg proizvoda. Ovde se navode osnovni SK.
- U toku razvojne faze pravi se plan projekta, SK se razrađuju i daje se nacrt arhitekture sistema.
- U fazi građenja dobija se kompletan softver koji se pridružuje do arhitekture sistema. Na taj način se dolazi do Beta Verzije Riliza (BVR).
- U prelaznoj fazi BVR se prosleđuje do korisnika radi testiranja. Nakon ispravke uočenih problema pravi se generalni riliz. U prelaznoj fazi se obučavaju korisnici.

Faze Jedinstvenog procesa nisu značajne u ovom trenutku za formalizaciju Jedinstvenog procesa tako da iste nismo u daljem tekstu razmatrali. Formalizacija Jedinstvenog procesa, koja će dalje biti detaljno objašnjena, je zasnovana na radnim tokovima Jedinstvenog procesa.

[44] U okviru matrice koja opisuje JPRS, polja koja predstavljaju presek kolona i redova, mogu biti prazna imati jedan ili dva znaka lista. Ukoliko je polje prazno to znači da se tu ne dešava nikakva aktivnost JPRS. Ako polje ima jedan list, to znači da se tu aktivnosti JPRS izvršavaju srednjim intezitetom. Ako polje ima dva lista, to znači da se tu aktivnosti JPRS izvršavaju intezivno.

4.3 Modeliranje poslovnih sistema

Pre nego što objasnimo razvoj jednog softverskog sistema pomoću jedinstvenog procesa potrebno je objasniti poslovni sistem, odnosno model poslovnog sistema iz koga se uočavaju sistemski slučajevi korišćenja i sistemski aktori koji predstavljaju ulaz u softverski sistem.

Realni poslovni sistem [45]

Pod realnim poslovnim sistemom, iz perspektive jedinstvenog procesa, podrazumevaju se tipični poslovni sistemi (banke, prodavnice, ..., itd) kao i netipični poslovni sistemi (sistem alarmnih uređaja, sistem rada kamere,...,itd.). Ono što je zajedničko za ovakve poslovne sisteme je moguće postojanja softverskih sistema unutar njih. Softverski sistemi treba da pruže podršku radu poslovnih sistema[46].
Svaki poslovni proces[47] se sastoji iz aktivnosti koje izvršavaju radnici (workers) na osnovu njima dodeljenih odgovornosti. Aktivnosti se izvode u okviru određenih organizacionih jedinica. Svaku aktivnost prate događaji (početak aktivnosti, prekid aktivnosti, završetak aktivnosti,…). U toku aktivnosti, radnici koriste osnovna sredstva sistema (kompjutere, štampače, mašine, alate,…) pomoću kojih manipulišu sa poslovnim entitetima[48] sistema (računi, prijemnice otpremnice, proizvodi,…). Korisnici poslovnog sistema nazivaju se poslovni partneri. Radnike, osnovna sredstva, poslovne entitete, organizacione jedninice i ostale činioce poslovnog sistema zvaćemo, jednim imenom poslovni resursi[49].

Poslovni model

Pomoću poslovnog modela SK se opisuje realni poslovni sistem. Poslovni procesi se opisuju pomoću poslovnih SK , dok se poslovni partneri opisuju pomoću poslovnih aktora . Poslovni model SK se prikazuje pomoću dijagrama slučaja korišćenja.
Realizacija poslovnih SK se opisuje pomoću poslovnog objektnog modela ili pomoću verbalnog opisa poslovnih SK. Poslovni objektni model se prikazuje pomoću interakcionih[50] ili dijagrama aktivnosti.

Za svaki poslovni SK se daje njegov verbalni opis koji treba da sadrži:

- Naziv SK
- Učesnike SK
- Aktore SK
- Početno stanje SK, koje predstavlja preduslov njegovog izvršenja
- Osnovne aktivnosti SK sa njihovim redosledom
- Postuslov izvršenja SK
- Nedopuštene aktivnosti SK
- Alternativni aktivnosti koji će se izvršiti ukoliko su narušeni uslovi izvršenja osnovnih aktivnosti
- Opis interakcija sistema sa aktorima (šta oni međusobno razmenjuju)
- Opis korišćenih objekata i vrednosti sistema
- Mora se eksplicitno opisati šta sistem radi (koje akcije izvršava) i šta aktor radi. Drugim rečima, moraju se podeliti odgovornosti sistema i aktora

Poslovno modeliranje je prvenstveno usmereno na opisivanje ponašanja realnog poslovnog sistema.
Pored poslovnog modeliranja postoji i domensko modeliranje.

Domenski model

Domenski model obuhvata najvažnije koncepte (domenske objekte) konteksta sistema. Domenski objekti predstavljaju "stvari (things)" koji postoje ili događaje koji se dešavaju u okruženju u kome sistem radi.

---

[45] U toku naših istraživanja napisali smo nekoliko radova koji se odnose na poslovne sisteme [Sv12-13, Sv17].

[46] Aktivnosti i delovi aktivnosti poslovnog procesa mogu biti podržani softverskim sistemom.

[47] Kod definisanja terminologije kojom će biti opisan poslovni sistem, dosta su nam pomogli radovi prof. Lazarevića [Laz3-4], koji se bave modeliranjem poslovnih procesa u kontekstu razvoja Informacionih sistema.

[48] Pored poslovnih entiteta, spominje se i termin radna jedinica (work unit). Kod JPRS radna jedinica predstavlja skup poslovnih entiteta koji oblikuju prepoznatljivu celinu za krajnjeg korisnika. Radnu jedinicu možemo shvatiti kao složenu ekransku forma ili složeni dokument koji se sastoji od skupa poslovnih entiteta koji je sačinjavaju.

[49] U daljem tekstu pod terminom **poslovni resurs** podrazumevaćemo:

a) Onoga ko vrši aktivnost (radnik).
b) Sredstva koje se koristi u vršenju aktivnosti (osnovna sredstva).
c) Mesto gde se izvršava aktivnost (organizaciona jedinica).
d) Entitet koji opisuju aktivnost i rezultate izvršenja aktivnosti (poslovni entitet).

Takođe ćemo pod terminom poslovni resurs podrazumevati i druge činioce koji se koriste u izvršenju poslovnih aktivnosti.

[50] Interakcioni dijagrami su sekvencni dijagrami i dijagrami saradnje (kolaboracioni dijagrami).

Domenski objekti se mogu dobiti iz:
- specifikacije zahteva ili
- intervjua sa domenskim ekspertom.

Domenske klase[51] se mogu javiti u tri tipična oblika:

a) Poslovni objekti[52], koji predstavljaju stvari sa kojima se manipuliše u poslovanju (računi, ugovori, ..., itd).
b) Realni objekti i koncepti[53], čiji trag (putanju) sistem treba da sačuva (avioni, projektili, trajektorije).
c) Događaji koji će se desiti ili su se desili (dolazak aviona, odlazak aviona, pauza za ručak, ...,itd).

Domenski model se obično opisuje pomoću dijagrama klasa (konceptualni model kod Larmana).
Kod domenskog modeliranja nije jasno određen put između domenskog modela i sistemskih slučajeva korišćenja[54]. Suprotno tome, kod poslovnog modeliranja je jasno određen put od korisnika poslovnog sistema do sistemskih slučajeva korišćenja.
Domensko modeliranje je nasuprot poslovnom modeliranju, pre svega usmereno na opisivanje strukture realnog poslovnog sistema.
Na osnovu poslovnog i domenskog modela pravi se rečnik termina.

4.4 Razvoj softverskog sistema pomoću jedinstvenog procesa

Pristup koji se koristi u modeliranju poslovnih sistema primenjen je i u modeliranju Jedinstvenog Procesa Razvoja Softvera[55]. Jedinstveni Proces se posmatra kao poslovni sistem koji se sastoji iz sledećih poslovnih procesa (radnih tokova - workflows)[56] :

a) Prikupljanje zahteva
b) Analiza
c) Projektovanje
d) Implementacija
e) Testiranje

U daljem tekstu svaki od radnih tokova biće objašnjen pomoću njegovih aktivnosti. Proizvode (artifakte)[57] koji nastaju kao rezultat radnog toka, i radnike (workera)[58] koji učestvuju u radnom toku nećemo razmatrati kako bi pojednostavili shvatanje JPRS.

4.4.1: PRIKUPLJANJE ZAHTEVA OD KORISNIKA

Proces prikupljanja zahteva (sistemskih slučajeva korišćenja) od korisnika sastoji se iz sledećih aktivnosti:
- Nalaženje aktora i slučajeva korišćenja .
- Određivanje prioritetnih slučajeva korišćenja.
- Detaljna specifikacija slučajeva korišćenja.
- Pravljenje prototipa korisničkog interfejsa.
- Struktuiranje modela slučaja korišćenja.

---

[51] Domenska klasa (objekat) može da opiše poslovnog partnera ili poslovni resurs poslovnog modela.

[52] JPRS kada objašnjava:
- poslovni model koristi termine poslovni entitet i radna jedinica kada želi da ukaže na račune, ugovore, … itd.
- domenski model koristi termin poslovni objekat kada želi da ukaže na račune, ugovore, … itd.

U tom smislu, u daljem tektu termine poslovni entitet i poslovni objekat posmatraćemo kao sinonime.

[53] Smatramo da realni objekti i koncepti u kontekstu našeg shvatanja termina poslovni resursi predstavljaju radnike, osnovna sredstva, organizacione jedinice i poslovne partnere.

[54] Sistemski SK se odnosi na softverski sistem, dok se poslovni SK odnosi na poslovni sistem. Kada se kaže SK podrazumeva se da je u pitanju sistemski SK.

[55] U daljem tekstu JPRS.

[56] Poslovni procesi se posmatraju kao radni tokovi - workflows, pri čemu se radni tokovi opisuju pomoću dijagrama aktivnosti. Radni tok se sastoji iz skupa aktivnosti koje izvršavaju radnici (workers). Rezultat tih aktivnosti predstavljaju određeni proizvodi (artifacts).

[57] U daljem tekstu umesto termina proizvod koristićemo termin artifakt. Artifakti, generalno gledajući, predstavljaju informacije koje su kreirane, promenjene ili korišćene od strane workera u radu (razvoju) sistema. Artifakt može biti model, model element ili dokument.

[58] U daljem tekstu umesto termina radnik koristićemo termin worker.

Nalaženje aktora i slučajeva korišćenja
Poslovni model se koristi kao osnova iz koje se pravi model slučaja korišćenja softverskog sistema. Pošto se svaki poslovni slučaj korišćenja sastoji iz aktivnosti, koje izvršavaju radnici[59], moguće je identifikovati aktivnosti, ili neke njene delove, koji se mogu automatizovati, odnosno za koje se može obezbediti podrška softverskog sistema. Navedene aktivnosti, ili njihovi delovi, postaju sistemski slučajevi korišćenja, dok radnici i poslovni aktori postaju sistemski aktori[60]. Treba naglasiti da se ne mogu sve aktivnosti poslovnih procesa automatizovati.

Određivanje prioritetnih slučajeva korišćenja
Na osnovu ove aktivnosti određuju se slučajevi korišćenja koji trebaju da se razviju ranije (prioritetni slučajevi korišćenja) odnosno kasnije (koji u prvim iteracijama razvoja nisu toliko važni).

Detaljni opis slučajeva korišćenja
Svrha ove aktivnosti je da detaljno opiše tok događaja za svaki slučaj korišćenja, što uključuje početak i kraj slučaja korišćenja kao i njegovu interakciju sa aktorima. Specifikacija SK-a se izvodi na osnovu verbalnog opisa poslovnih SK.

Pravljenje prototipa korisničkog interfejsa
Svrha ove aktivnosti je izgradnja prototipa korisničkog interfejsa. Prototip korisničkog interfejsa pravi se za svaki slučaj korišćenja. Pravljenje prototipa korisničkih interfejsa obavlja se u dva koraka:
a) Prvo se razmatra šta korisnički interfejsi trebaju da imaju (koje domenske objekte) kako bi zadovoljili potrebe aktora za realizaciju slučaja korišćenja. Navedeni korak predstavlja logičko projektovanje korisničkog interfejsa.
b) Nakon toga se prave konkretni (fizički) korisnički interfejsi, odnosno njihovi prototipovi, pomoću kojih aktori komuniciraju sa sistemom, odnosno izvršavaju željene slučajeve korišćenja[61].

Struktuiranje modela slučaja korišćenja
Model slučaja korišćenja se struktuira kako bi se:
- dobili generalni (deljeni) slučajevi korišćenja čije ponašanje mogu koristiti (naslediti) specifični slučajevi korišćenja.
- izveli dopunski ili opcioni slučajevi korišćenja koji mogu da prošire ponašanje specifičnih slučajeva korišćenja.

Kao što je bilo rečeno specifikator slučajeva korišćenja daje detaljne opisa svakog slučaja korišćenja. Na osnovu toga analitičar sistema, posmatrajući model slučaja korišćenja kao celinu (odnosno sve slučajeve korišćenja), vrši restruktuiranje modela slučaja korišćenja. Tako dobijeni model je lakši za razumevanje i rad.

4.4.2: ANALIZA

Zahtevi koji se analiziraju predstavljeni su preko modela slučaja korišćenja, detaljnim verbalnim opisom svakog slučaja korišćenja i prototipovima korisničkih interfejsa za svaki slučaj korišćenja. Razmotrićemo slučajeve korišćenja iz perspektive njihovog daljeg razvoja:
a) Slučajevi korišćenja trebaju biti što je moguće više međusobno nezavisni tako da promena ponašanje jednog slučaja korišćenja ne utiče na promenu ponašanja drugog slučaja korišćenja.
Pre izvršenja slučaja korišćenja proveravaju se potrebni preduslovi za njegovo izvršenje.
b) Slučajevi korišćenja (zahtevi) su često opisani jezikom korisnika koji je neformalan, neprecizan i dvosmislen što predstavlja problem u daljem razvoju slučaja korišćenja. Zbog toga je neophodno da se pre analize slučajeva korišćenja, kada se prave formalne realizacije slučajeve korišćenja) još jednom, što je moguće precizicije i jasnije, daju specifikacije slučajeva korišćenja
c) Slučajevi korišćenja treba da opišu u potpunosti njihovu funkcionalnost, odnosno trebaju da budu realni slučajevi korišćenja.

[59] Uloga radnika predstavlja obavezu radnika da izvrši određenu aktivnost u poslovnom procesu.
[60] Termin sistemski aktori se koristi da označi aktore softverskog sistema.
[61] Smatramo da je prerano, da se u ovoj fazi daje predlog fizičkog izgleda korisničkog interfejsa. Prototip korisničkog interfejsa treba pre svega da navede domenske objekte, koji treba da se nalaze na njemu, bez ulaženja u to kako će fizički izgledati korisnički interfejs.

Analiza omogućava dalje razrađivanje slučaja korišćenja (zahteva), odnosno formalan opis realizacija slučaja korišćenja. Struktura modela analize (zasnovana na klasama i paketima analize) je ortogonalna na strukturu modela slučaja korišćenja (zasnovana na slučajevima korišćenja).

Radni tok analize sastoji se iz sledećih aktivnosti:
a) Analize slučajeva korišćenja
b) Analiza klasa
c) Analiza paketa
d) Analiza arhitekture

Analiza slučajeva korišćenja

Slučajevi korišćenja se analiziraju kako bi se:
- Identifikovale klase analize čiji objekti su potrebni da bi realizovali slučajevi korišćenja.
- Rasporedilo (distribuiralo) ponašanje slučaja korišćenja na objekte analize koji obezbeđuju navedeno ponašanje.
- Prikupili specijalni zahtevi definisani nad slučajevima korišćenja.

Slučajevi korišćenja su verbalno opisani u modelu slučaja korišćenja. Na osnovu njih, u fazi analize prave se dijagrami saradnje[62] koji opisuju realizacije slučajeva korišćenja preko objekata analize. Objekti analize mogu biti granični, kontrolni i entitetski.
Za razliku od opisa slučaja korišćenja kod modela slučaja korišćenja, kod koga su odgovornosti podeljene između aktora i sistema (programa), kod opisa slučaja korišćenja u modelu analize, odgovornosti sistema raspodeljene su i dodeljene aktorima i objektima analize.
Iz navedenog se može preciznije zaključiti da su odgovornosti aktora iz modela SK jednoznačno preslikane u odgovornosti aktora u modelu analize. Međutim odgovornost sistema (programa) iz modela SK su raspodeljene na:

- granične,
- kontrolne i
- entitetske objekte analize .

Odgovornosti koje se dodeljuju objektima analize ustvari predstavljaju sistemske operacije koje ti objekti moraju da obezbede. Sistemske operacije (SO) koje se dodeljuju do objekata analize mogu biti:

- granične SO koje se dodeljuju do graničnih objekata,
- kontrolne SO koje se dodeljuju do kontrolnih objekata i
- entitetske SO koje se dodeljuju do entitetskih objekata.

Razmotrićemo nekoliko činjenica vezanih za dijagrame saradnje:

- Granični objekat prihvata poruku od aktora da izvrši operaciju ili da poruku dalje prosledi do drugih objekata (kontrolnih). Kontrolni objekti prihvataju poruku od graničnih objekata da izvrše operaciju ili da poruku dalje prosledi do entitetskih objekata.
- Kod dijagrama saradnje prave se linkovi između objekata, pri čemu se linkovima dodeljuju poruke. Pored svake poruke data je usmerena strelica koja ukazuje na objekat koji treba da odgovori na navedenu poruku. Poruka predstavlja pobudu da objekat izvrši neku operaciju.
- Linkovi u dijagramima saradnje treba da postanu asocijacije između klasa analize.
- Kod dijagrama saradnje osnovna pažnja je usmerena na: a) veze između objekata i b) zahteve koji se postavljaju objektima[63]. To znači da dijagram saradnje naglasak stavlja na dodeljivanje odgovornosti objektima, za razliku od sekvencnog dijagrama koji naglasak stavlja na redosled izvršenja interakcija između aktora i sistema.
- Dijagrami saradnje treba da se detaljno tekstualno opišu.

Analiza klasa

Na osnovu objekata analize i sistemskih operacija prave se klase analize:

- granična klasa
- kontrolna klasa i
- entitetska klasa

Klasama se dodeljuju odgovornosti na osnovu uloga (odgovornosti) koje imaju njihovi objekti kod realizacija slučaja korišćenja.

[62] Kod Larmanove metode u fazi analize se prave sekvencni dijagrami za SK. To je veoma bitna razlika između te dve metode.
[63] Redosled izvršenja sekvence akcija kod dijagrama saradnje je manje važan u odnosu na veze između objekata i zahteve koji se postavljaju objektima.

Granične klase (klase korisničkog interfejsa) se koriste da modeluju interakciju između sistema i njegovih aktora. To znači da je granična klasa odgovorna za definisanje interfejsa prema aktorima. Granična klasa često treba na apstraktan način da prikaže prozore, forme, komunikacione interfejse, terminale, ..., itd, koji su definisani prototipovima korisničkog interfejsa. Svaka granična klasa treba da bude povezana najmanje sa jednim aktorom i obrnuto. Granična klasa može da učestvuje u dva ili više slučaja korišćenja.

Dinamika sistema se modeluje pomoću **Kontrolnih klasa** (klase ponašanja). One su odgovorne za:
a) kordinaciju izvršenja pojedinih operacija i
b) kompletnost izvršenja transakcija.
Kontrolne klase nisu odgovorne za komunikaciju sa aktorima i za održavanje entitetskih klasa.

Entitetske klase (klase strukture) se koriste da modeluju objekte koji imaju vremensko trajanje (perzistentne objekte). One se direktno izvode iz domenskih klasa. Pored strukture koju opisuju, entitetske klase sadrže i ponašanje[64]. *Glavna razlika između domenskih i entitetskih klasa je u tome što domenske klase predstavljaju poslovne objekte u opštem smislu, dok entitetske klase predstavljaju objekte koji će biti obrađeni preko softverskog sitema.*

Na osnovu uočenih klasa analize pravi se dijagram klasa analize.

Atributi klasa
Za entitetske klase analize treba identifikovati njihove atribute. Postoje sledeće preporuke, kojih se treba držati, kod identifikovanja atributa:
- Ime atributa treba da bude imenica.
- Tip atributa treba da bude konceptualan i za taj tip ne treba vezivati nikakva implementaciona ograničenja.
- Kada se bira tip atributa, treba koristite već postojeće konceptualne tipove.
- Ukoliko klasa analize postane suviše komplikovana za razumevanje zbog njenih atributa, neki od atributa treba da obrazuju klasu.

Asocijacije između klasa
Objekti analize ulaze u međusobnu interakciju preko linkova iz dijagrama saradnje. Linkovi između objekata postaju asocijacije između klasa analize. Link između objekata može da ukaže na vezu agregacije[65] između objekata. Za asocijaciju se definiše:
- preslikavanja,
- imena uloga,
- klase koje učestvuju u asocijaciji i
- redosled uloga.

Agregacije treba koristiti kada objekti predstavljaju:
- Koncepte koji fizički sadrže druge koncepte (kola sadrži vozača i putnika).
- Koncepte koji su komponovani od drugih koncepata (kola sadrže motor i sedišta).
- Koncepte koji oblikuju kolekciju objekata po nekom semantičkom kriterijumu (porodica koja sadrži oca, majku i decu).

Ukoliko više klasa ima deljeno i zajedničko ponašanje vrši se generalizacija klasa, odnosno prave se generalne klase.

Prikupljanje specijalnih zahteva vezanih za klase analize
U fazi analize se za klasu analize uočavaju nefunkcionalni zahtevi koji treba da se obrade u fazama projektovanja i implementacije.

Analiza paketa
Paketi analize omogućavaju da se model analize podeli u manje upravljive delove. Oni se mogu identifikovati na dva načina:
a) Jedan slučaj korišćenja analize može biti dodeljen do paketa analize ukoliko se želi podržati jedan poslovni proces.
b) Ukoliko se žele nezavisno razvijati granične, kontrolne i entitetske klase jednog slučaja korišćenja onda se prave posebni paketi za:

[64] Ono što smatramo glavnim nedostatkom JPRS jeste zanemarivanje razvoja domenskog modela (strukture sistema) u odnosu na model slučaja korišćenja (ponašanje sistema).
[65] Agregacija je specijalan slučaj asocijacije koji opisuje vezu deo-celina između agregata (celine) i njenog dela.

- granične klase,
- kontrolne klase i
- entitetske klase.

c) Više slučajeva korišćenja mogu se dodeliti do jednog paketa analize ukoliko se želi zadovoljiti sledeći kriterijum:

- slučajevi korišćenja treba da podrže specifičnog aktora sistema.

Ovako dobijeni paketi mogu da lokalizuju promene kod:

- ponašanja poslovnog procesa,
- korisničkog interfejsa jednog slučaja korišćenja,
- poslovne logike jednog slučaja korišćenja i
- aktora.

Iz paketa analize mogu se izvesti:

- Generalni paketi analize. U slučajevima kada dva paketa analize dele istu klasu pravi se novi paket u koji se stavlja deljena klasa. Tako dobijeni paket analize postaje generalni paket od koga su zavisni drugi paketi (koji koriste klase koje se nalaze u generalnom paketu). Deljene klase su obično entitetske klase koje su nastale na osnovu domenskih klasa.
- Paketi servisa. Kada su funkcionalni zahtevi u potpunosti razumljivi, odnosno kada su odgovornosti precizno dodeljenje do konkretnih klasa analize u realizaciji slučaja korišćenja, tada se za klasu ili za skup klasa uočavaju servisi koje oni obezbeđuju za slučajeve korišćenja. Takve klase ili skupovi klasa koje sadrže jedan ili više servisa koji se koriste u realizacijama slučaja korišćenja dodeljuju se do paketa servisa.
- Zavisnosti između paketa analize. Ukoliko između klasa, koje se nalaze u različitim paketima analize postoji zavisnost, tada se zavisnost uspostavlja i između paketa analize koji sadrže te klase. Smer zavisnosti treba da bude isti kao smer veze koja postoji između klasa. Veze između paketa analize treba da budu što je moguće slabije, dok kohezija unutar svakog paketa treba da bude što veća. To znači da treba težiti da se smanje veze (zavisnosti) između paketa analize.
  Paketi iz modela analize koji su aplikaciono-specifični treba da budu u jednom nivou, dok generalni paketi treba da budu u drugom nivou i između njih treba uspostaviti vezu zavisnosti ukoliko ona postoji.

Analiza arhitekture

Analiza arhitekture je aktivnost kojom se opisuju svi artifakti modela analize koji su značajni za arhitekturu sistema. U tom smislu navedena aktivnost treba da identifikuje pakete analize, klase analize kao i specijalne zahteve koji su značajni za arhitekturu sistema.
Specijalni zahtevi koji se javljaju u fazi analize se odnose na:

- Perzistentnost klasa,
- distribuiranje i konkurentnost klasa i njihovih operacija,
- zaštitu klasa i kompletnog sistema,
- toleranciju greške (fault tolerancy) ,
- i upravljanje transakcijama.

4.4.3 PROJEKTOVANJE

U fazi projektovanja sistema daje se oblik sistemu, koji treba da podrži sve postavljene zahteve korisnika (funkcionalne i nefunkcionalne) kao i druga ograničenja.
Kroz fazu projektovanja treba da se:

- Shvate: a) nefunkcionalni zahtevi i ograničenja povezana sa programskim jezicima b) komponente koje se mogu ponovo koristiti c) operativni sistemi d) tehnologije distribuiranja i konkurentnosti e) tehnologije baza podataka f) tehnologije korisničkih interfejsa g) tehnologije upravljanja transakcijama,..., itd.
- Kreira model projektovanja koji sadrži: podsisteme, interfejse i klase. On treba da predstavlja ulaz u fazu implementacije. Model projektovanja predstavlja apstrakciju modela implementacije. Model implementacije treba da sačuva strukturu modela projektovanja i da detaljno razradi tu strukturu.
- Odrede interfejsi između podsistema. Interfejsi, u konktekstu upravljanja razvojem sistema, predstavljaju instrumente sinhronizacije između različitih razvojnih timova. Interfejsi su takođe potrebni kod razvoja arhitekture sistema.

Radni tok projektovanja sastoji se iz sledećih aktivnosti:
a) Projektovanje arhitekture
b) Projektovanje slučaja korišćenja
c) Projektovanje klase
d) Projektovanje podsistema

Projektovanja arhitekture

Aktivnost projektovanja arhitekture kao artifakte daje:
- Kompjuterske čvorove i konfiguracije njihove mreže.
- Podsisteme i njihove interfejse.
- Klase projektovanja koje su značajne za arhitekturu, kao što su aktivne klase.
- Generičke mehanizme projektovanja koji su vezani za perzistentnost, distribuciju, performanse,...,itd.

Kroz ovu aktivnost arhitekta razmatra različite mogućnosti ponovnog korišćenja postojećih softverskih sistema i podsistema.

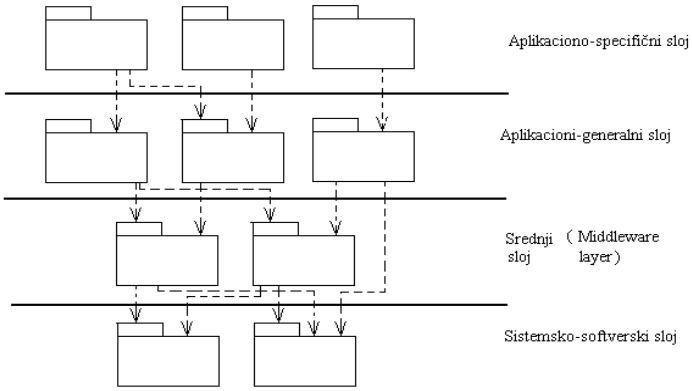
Identifikovanje čvorova i konfiguracije mreže

Fizička konfiguracija mreže ima značajan uticaj na softversku arhitekturu, odnosno na distribuciju funkcija softverskog sistema na različite čvorove mreže. Obično se kod projektovanja konfiguracije mreže koristi *tronivojski uzor*[66] *(three-tier pattern)* gde su klijenti (korisnički interfejs) u jednom, logika poslovne aplikacije u drugom, dok su funkcije baze podataka u trećem nivou. Jednostavan *klijent/server uzor (client/server pattern)* je specijalan slučaj tronivojskog uzora gde je logika poslovne aplikacije prudružena do jednog od dva postojeća nivoa (do klijenta ili do baze podataka).
Kod projektovanja konfiguracija mreže mora se voditi računa o sledećem:
- Koji čvorovi su obuhvaćeni mrežom i koji su kapaciteti čvorova (izraženi preko snage procesora i veličine memorije)?
- Koji tipovi veza su između čvorova i koji komunikacioni protokoli se koriste između njih?
- Koje su osobine veza i komunikacionih protokola, kao što su raspoloživost veze, kvalitet veze,...,itd.?
- Da li postoji potreba za: a) redudantnim mogućnostima procesiranja, b) migracijom procesa, c) čuvanjem rezervnih kopija podataka, ..., itd.?

Ukoliko se znaju ograničenja i mogućnosti čvorova i njihovih veza, arhitekta može da inkorporira odgovarajuće tehnologije u softverski sistem, kao što je npr. object request broker i servisi za replikaciju podataka koji se mogu koristiti za realizaciju distribucije objekata i podataka u sistemu.
Svaka konfiguracija mreže, uključujući i specijalne konfiguracije za testiranje i simulaciju, treba da bude opisana u odvojenom dijagramu raspoređivanja. Tek nakon toga je moguće započeti distribuiranje funkcija softverskog sistema na čvorove mreže.

Identifikovanje podsistema i njihovih interfejsa

Podsistemi su elementi modela koji obezbeđuju da se model projektovanja organizuje preko skupa upravljivih delova (podsistema).
Identifikovanje podsistema započinje na aplikaciono-specifičnom i aplikaciono- generalnom sloju (Slika SLAR) gde se podsistemi mogu jednoznačno identifikovati na osnovu paketa analize.



Slika SLAR : Podsistemi u različitim slojevima arhitekture

[66] Problem koji rešava navedeni uzor, može da se definiše preko sledećeg pitanja:
Na koji način treba distribuirati odgovornosti aplikacije u najopštijem smislu?

Jednoznačno preslikavanje između paketa analize i podsistema može da bude narušeno u sledećim slučajevima:

- Jedan deo paketa analize je preslikan u njegov sopstveni podsistem. To znači da će se paket analize preslikati u više podsistema.
- Jedan deo paketa analize će biti realizovan pomoću postojećeg softverskog proizvoda. U tom slučaju taj deo paketa analize postaje podsistem srednjeg sloja ili podsistem sloja sistemskog softvera.
- Paketi analize nisu pripremljeni za raspoređivanje do odgovarajućih čvorova u mreži. Od jednog takvog paketa analize može nastati više podsistema koji će biti dodeljeni do različitih čvorova u mreži.

Nakon toga se identifikuju podsistemi iz srednjeg sloja (Middleware layer) i sloja sistemskog softvera. Srednji sloj sadrži podsisteme koji se mogu ponovo koristiti u razvoju novih softverskih proizvoda. U podsisteme srednjeg sloja spadaju: tehnologije distribucije objekata, tehnologije upravljanja tansakcijama, oruđa za projektovanje grafičkog korisničkog interfejsa, …, itd.
U sloj sistemskog softvera spadaju operativni sistemi, sistemi za upravljanje bazama podataka, komunikacioni softver, interfejsi prema hardverskim uređajima,...,itd.
Kada se biraju podsistemi srednjeg nivoa i podsistemi nivoa sistemskog softvera oni treba da imaju eksplicitni interfejs[67], kako bi korisnici ovih podsistema mogli da za postojeći interfejs izaberu, ukoliko ima potrebe, drugu realizaciju tog interfejsa. To znači da tekući projekat razvoja softvera, treba po mogućnosti, da ima što je moguće manju zavisnost od navedenih softverskih proizvoda (podsistema) i njihovih proizvođača.
Zavisnosti između podsistema treba da budu definisane, ako postoje veze između elemenata koji čine te podsisteme. Smer zavisnosti između podsistema treba da bude isti kao smer veze između elemenata podsistema. Ukoliko podsistemi imaju interfejse, zavisnost između podsistema se predstavlja kao zavisnost između njihovih interfejsa.

Interfejs nekog podsistema definiše skup operacija (klasa i/ili podsistema) tog podsistema. Interfejsi se prave za podsisteme u sledećim slučajevima:
a) Kada podsistemi imaju direktnu zavisnost, tada treba za svaki podsistem napraviti interfejs i zavisnost između podsistema predstaviti kao zavisnost između njihovih interfejsa.
b) Ako postoji paket analize koji ima klasu koja se poziva izvan paketa, tada za tu klasu, odnosno za taj paket analize, treba obezbediti interfejs. Navedeni paket analize se u fazi projektovanja može preslikati u jedan ili više podsistema. U tom slučaju će i navedeni interfejs paketa analize da se preslika u jedan ili više interfejsa podsistema projektovanja.

Identifikovanje klasa projektovanja značajnih za arhitekturu
Klase projektovanja se mogu dobiti (jednoznačno preslikati) na osnovu klasa koje su dobijene u fazi analize. Takođe klase projektovanja mogu se dobiti na osnovu sistemskih operacija iz faze analize. U tom slučaju klasa projektovanja je odgovorna da realizuju neku od sistemskih operacija.
Ukoliko u sistemu postoji potreba za konkurentnim izvršavanjem više objekata prave se aktivne klase. Aktivna klasa, koja predstavlja značajan proces za sistem, je kandidat za izvršnu komponentu koja će biti identifikovana u toku faze implementacije.

Identifikovanje generičkih mehanizama kod projektovanja
U ovom koraku se razmatraju mogućnosti obrade specijalnih zahteva, koji su identifikovani u fazi analize, na osnovu postojećih tehnologija projektovanja i implementacije. Generički mehanizmi projektovanja nastaju kao rezultat ovog koraka i oni predstavljaju rešenja postavljenih specijalnih zahteva.
Generički mehanizmi projektovanja se odnose na:

- perzistentnost objekata sistema
- transparentnu distribuciju objekata
- zaštitu sistema
- otkrivanje grešaka i oporavak sistema
- upravljanje transakcijama sistema

[67] Treba da imaju razdvojen interfejs podsistema od realizacije tog podistema.

Navedeni mehanizmi se smeštaju u srednji sloj arhitekture sistema. Generički mehanizmi projektovanja, koji se odnose na realizacije slučaja korišćenja - projektovanja i klase projektovanja smeštaju se u aplikaciono - generalni sloj arhitekture sistema.

Projektovanje slučaja korišćenja

Slučajevi korišćenja se projektuju kako bi se:

- Identifikovale klase projektovanja, podsistemi i interfejsi čiji objekti su potrebni da bi realizovali slučajeve korišćenja.
- Opisala interakcija između objekata kod realizacije SK.
- Opisala interakcija između podsistema
- Prikupili implementacioni zahtevi koji se odnose na realizaciju SK.

Projektovanje klasa

Klase koje se projektuju, treba da realizuju slučajeve korićenja i nefunkcionalne zahteve. U toku projektovanja klasa, izvode se sledeći koraci u kojima se:

- Pravi nacrt klasa projektovanja.
- Vrši identifikuje njihovih operacija.
- Vrši identifikovanje njihovih atributa
- Definišu veze između njih (asocijacija i agregacija).
- Vrši identifikovanje generalizovanih klasa projektovanja.
- Projektuju metode, koje predstavljaju realizacije operacija.
- Opisuju stanja klasa projektovanja.
- Obrađuju specijalni zahtevi.

Projektovanja podsistema

Podistemi koji se projektuju:

- Treba da budu što je moguće više nezavisni od drugih podsistema i/ili njihovih interfejsa.
- Treba da obezbede dobre intrefejse.
- Treba da sadrže korektne realizacije potrebnih operacija.

4.4.4 IMPLEMENTACIJA

Rezultati projektovanja ulaze u implementaciju i na osnovu njih se prave komponente sistema (izvorni kod, byte kod, izvršni kod, skriptovi i slično)[68].

Osnovna svrha implementacije se odnosi na pravljenje izvršne arhitekture celoga sistema. Pored toga, u toku implementacija treba da se:

- Planira integracija sistema.
- Distribuiraju izvršne komponente u čvorove.[69]
- Implementiraju klase projektovanja i podsistemi, koji su definisani u toku projektovanja sistema. Od klasa projektovanja nastaju komponente koje sadrže izvorni kod.
- Testiraju komponente, koje se zatim integrišu sa celinom sistema.

Radni tok implementacije sastoji se iz sledećih aktivnosti:

a) Implementacija arhitekture.
b) Integracije sistema.
c) Implementacije podsistema.
d) Implementacije klasa.
e) Testiranje komponenti kao individualnih jedinica.

Implementacija arhitekture

U toku aktivnosti implementacije arhitekture navode se najvažnije komponente arhitekture[70], koje se zatim preslikavaju u čvorove mreže. Najvažnije komponente arhitekture su izvršne komponente[71].

---

[68] U toku naših istraživanja napisali smo nekoliko radova *[Sv7,Sv20,Cir5,Cir9]* koji se odnose na softverske komponente. Takođe je napisana i mr. teza koja se bavi softverskim komponentama *[Sv22]*.

[69] Čvorovi su uočeni u modelu raspoređivanja.

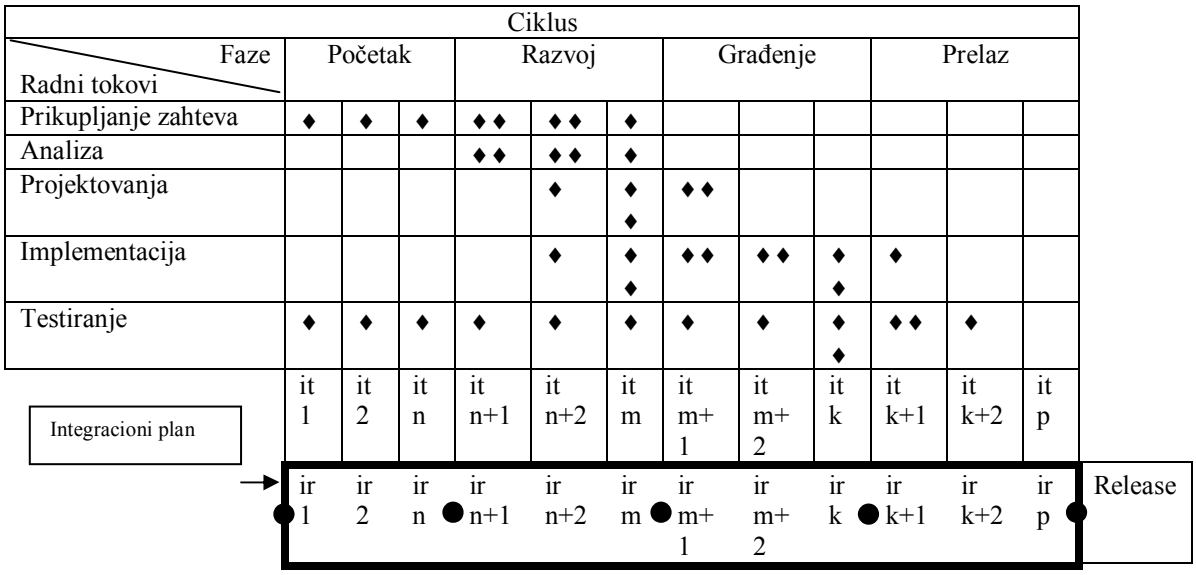
[70] Komponente se dobijaju na osnovu klasa projektovanja. **Komponente su sredstva za pakovanje klasa (koje sadrže izvorni kod) u datoteke.**

[71] Izvršne komponente se mogu izvršiti na čvorovima mreže. Izvršna komponenta sadrži jednu ili više aktivnih klasa, pri čemu svaki objekat aktivnih klasa ima svoju nit trajanja. Kada program u Javi počne da se izvršava, on automatski kreira i izvršava jednu nit koja se zove glavna programska nit. Iz glavne programske niti mogu se kreirati objekti aktivnih klasa, koji ponavljamo, svaki ponaosob, imaju svoju nit trajanja.

Za zvaki podsistem i interfejs projektovanja, pravi se odgovarajući implementacioni podsistem i interfejs (preslikavanje je izomorfno). Svaki podsistem implementacije sastoji se iz skupa komponenti. Ono što predstavlja glavni izazov u toku implementacije jeste izrada komponenti koje treba da realizuju podsisteme.

Integracija sistema

Kao što je na početku objašnjenja JPRS rečeno, svaka faza JPRS prolazi kroz seriju iteracija (it 1,it 2, …, it p). Rezultat iteracije je interni riliz (ir 1, ir 2, …, ir p) ili build, koji predstavlja inkrement za sistem.



Milestones - ●

Aktivnost integracije sistema se sastoji iz dva koraka:

a) Kreira se integracioni plan koji opisuje "build-ove".
b) U sistem se integriše svaki "build" koji je zadovoljio testiranje.

Integracioni plan opisuje sekvencu zahtevanih "build-ova" u jednoj iteraciji. Takav plan za svaki "build" opisuje:

- Funkcionalnost koja se očekuje da bude implementirana u "build-u". To je listing slučajeva korišćenja i/ili njihovih scenaria (tokova aktivnosti). Taj listing može takođe da ukaže i na suplementarne zahteve.
- Koji deo implementacionog modela (podsistema i komponenti) je zadužen da obezbedi navedenu funkcionalnost.

Nakon toga se prave realizacije podsistema i komponenti koje treba da obezbede funkcionalnost build-a. Realizovane komponente[72] se ugrađuju u build.

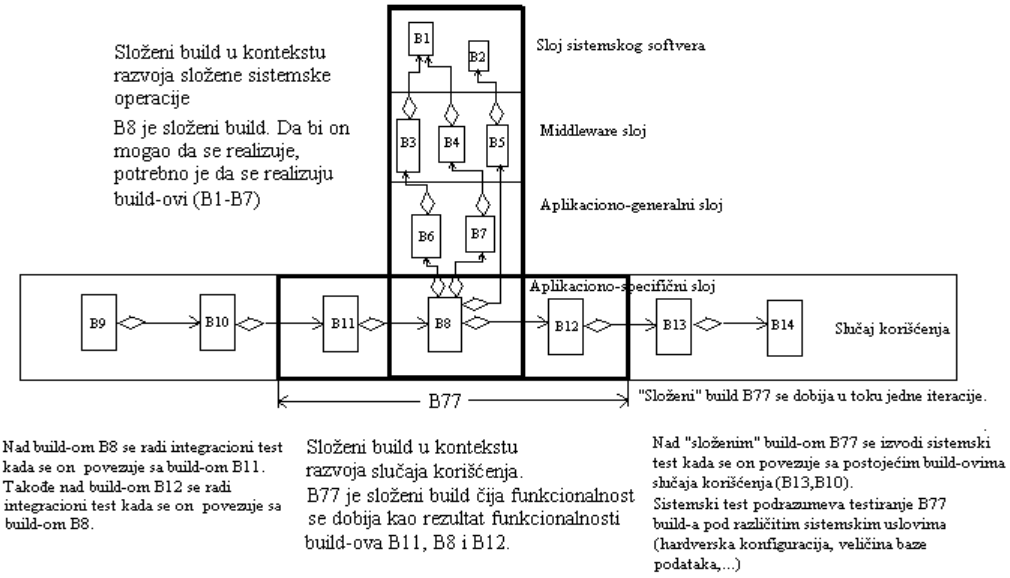
Složenost build-a može da se ogleda u dva konteksta:

1. U kontekstu razvoja slučaja korišćenja i i/ili nekog njegovog scenaria (toka aktivnosti). Tada svaki sledeći build dodaje funkcionalnost predhodnom build-u.
2. U kontekstu razvoja neke od složenih sistemskih operacija. "Build" jednog sloja arhitekture treba da bude zasnovan na "build-u" predhodnog sloja arhitekture. Tada se sistem širi po slojevima hijerarhije. Razvoj build-ova treba da ide od sloja sistemskog softvera ka aplikacionim slojevima.

Dobijeni build se testira (integraciono i sistemsko testiranje), pre nego što se ugradi u sistem. Integracioni test se radi pri povezivanju (integraciji) build-ova. Sistemski test se radi na kraju iteracije, kada se ispituje ponašanje build-a pod različitim sistemskim uslovima (brzina procesora, veličina operativne memorije, veličina baze podataka, broj korisnika,…, itd).Integracioni test build-a se zasniva na slučajevima korišćenja.

[72] Pošto se podsistemi sastoje iz komponenti, kao rezultat realizacije podsistema i komponenti se dobija skup realizovanih komponenti. Za realizovane komponente se kaže da predstavljaju jedinice testiranja (unit tested).

Build ne treba da uključi suviše mnogo novih ili da previše razrađuje postojeće komponente. Inače, može biti veoma teško da se on testira i integriše u sistem. Ako je neophodno neke komponente mogu biti implementirane kao "stub[73]" kako bi se minimizirao broj novih komponenti koje se uključuju u "build".

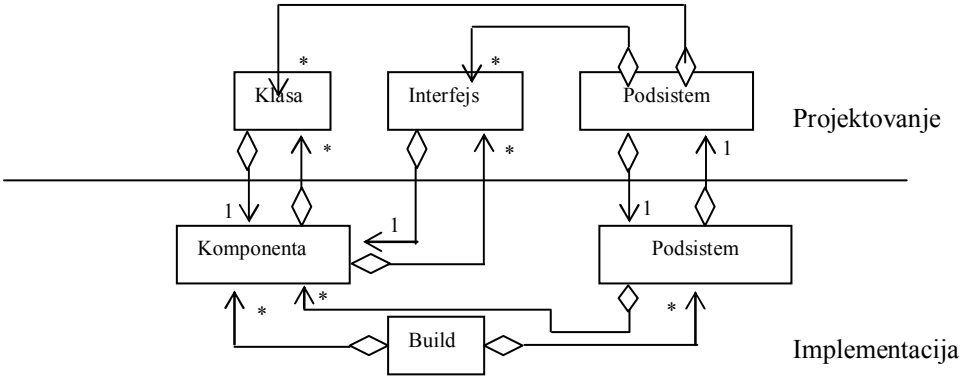


JPRS35: "Složeni " build u kontekstu razvoja slučaja korišćenja i u kontekstu razvoja složene sistemske operacije

Implementacija podsistema

U planu integracije "build-ova" se navodi redosled po kome će se implementirati podsistemi i komponente, koje su zadužene da realizuju funkcionalnost "build-ova".

Svaki podsistem implementacije je izomorfno vezan za podsistem projektovanja (Slika Build). Svaka klasa projektovanja nekog podsistema vezana je za jednu komponentu, međutim komponenta može biti vezana za više klasa projektovanja. Sličan je odnos između interfejsa projektovanja i komponenti koje treba da ih realizuju. Jedan interfejs je vezan za jednu komponentu, dok jedna komponenta može realizovati više interfejsa. Podsistem projektovanja se sastoji od jedne ili više klasa i/ili interfejsa. Podsistem implementacije sastoji se iz jedne ili više komponenti.



Slika Build: Odnos build-a i elemenata projektovanja i implementacije

Implementacija klasa

Za svaku klasu projektovanja pravi se odgovarajuća komponenta, koja treba da je implementira. Aktivnost implementacije klasa sastoji se iz sledećih koraka:

- Navode se komponente (sadržaj komponenti je izvorni kod) koje treba da se implementiraju.
- Izvorni kod se dobija iz klasa projektovanja kao i veze klasa projektovanja sa drugim klasama projektovanja.
- Implementiraju se operacije klasa projektovanja pomoću metoda.
- Obezbeđuje se da komponente imaju isti interfejs kao klase projektovanja.

Kao što je rečeno jedna komponenta može biti vezana za više klasa projektovanja. Iz praktičnih razloga preporučuje se da svaka klasa projektovanja bude vezana za jednu komponentu.

[73] Stub je komponenta sa minimalnim sadržajem naredbi (skeletom), koje su potrebne da bi se razvijale i testirale druge komponente koje zavise od nje (stuba-a).

Testiranje komponenti kao individualnih jedinica

Pre integracionog i sistemskog testiranja komponente, odnosno "build-a" za koji je vezana komponenta, vrši se testiranje rada komponente kao izdvojene celine. Postoje sledeći tipovi testiranja:

a) Testiranje specifikacije – ili testiranje komponente kao crne kutije, gde se testira spoljašnja ponašanja komponente.
b) Testiranje strukture – ili testiranje komponente kao bele kutije, gde se testira interna implementacija komponente.

Pored navedenih testova postoje i testovi vezani za performanse komponente. Nakon testiranja komponenti kao izdvojenih jedinica, vrši se integraciono i sistemsko testiranje, kojim se ispituje rad više međusobno povezanih komponenti koje treba integrisati u sistem.

4.4.5 TESTIRANJE

U toku testiranja treba da se verifikuju rezultati implementacije, pre nego što rezultat implementacije postane generalni riliz[74].

Na početku testiranja se pravi plan testiranja za svaki "build" i za svaku iteraciju. Zatim se projektuju i implementiraju:

d) Test slučajevi koji opisuju šta test treba da proveri,
e) test procedure koje opisuju kako će se izvršiti test i
f) test komponente koje treba da automatizuju test procedure ukoliko je to moguće.

Na kraju se rade konkretni testovi, čiji rezultati se sistematski obrađuju. Postoje dve osnovne vrste testa: integracioni i sistemski test. Integracioni test se radi za svaki build koji se povezuje (integriše) sa drugim build-ovima, dok se sistemski test radi za build, koji je nastao kao rezultat jedne iteracije[75].

Ukoliko se u toku testiranja desi neki problem (defekt), tada se "build" koji je testiran, vraća nazad do projektovanja ili implementacije kako bi se uočeni problem rešio.

Radni tok testiranja sastoji se iz sledećih aktivnosti:

a) Planiranje testa.
b) Projektovanje testa.
c) Implementacija testa.
d) Testiranje integracije.
e) Sistemsko testiranje.
f) Testiranje evaluacije.

Planiranje testa

U toku aktivnosti planiranja testa izvode se sledeće akcije:

a) Opisuje se strategija testiranja,
b) navode se potrebe za resursima (ljudskim i sistemskim) i
c) pravi se raspored izvršenja aktivnosti.

Projektovanje testa

U toku aktivnosti projektovanja testa se:

a) Identifikuju i opisuju test slučajevi za svaki build.
b) Identifikuju i opisuju test procedure koje ukazuju na način izvršenja test slučajeva.

Test slučajevi mogu se podeliti na integracione i sistemske test slučajeve.

Integracioni test slučajevi se koriste da verifikuju interakciju komponenti koje treba da obezbede funkcionalnost "build-a".

Neki test slučajevi vezani za ranije "build-ove" mogu se koristiti kod testiranja novih "build-ova". Takvi test slučajevi nazivaju se regresioni testovi.

Test slučajevi moraju da budu fleksibilni ukoliko se desi promena softvera.

Implementacija testa

Kod aktivnosti implementacije testa prave se test komponente koje treba da automatizuju test procedure. Test komponente se kreiraju na sledeći način:

a) pomoću alata za automatizaciju testiranja ili
b) neposrednim programiranjem test komponenti.

---

[74] Generalni riliz (release) predstavlja izvršni softverski proizvod koji se isporučuje krajnjim korisnicima. Pre testiranja se pravi beta riliz, koji predstavlja neispitani (netestiranu) izvršni softverski proizvod. Često se beta riliz namerno, kao demo program, daje na upotrebu krajnjim korisnicima, kako bi oni mogli da ispitaju (testiraju) program. Krajnji korisnici tada obaveštavaju projektni tim, ukoliko primete test slučajeve u kojima program ne daje očekivane rezultate.

[75] Kada smo objašnjavali implementaciju, detaljno smo objasnili kada se radi integracioni a kada sistemski test.

Testiranje integracije

Test integracije se zahteva za svaki "build" koji je kreiran u toku iteracije. Integraciono testiranje se sastoji iz sledećih koraka:

a) Izvršava se test integracije za "build", odnosno izvršava se test procedura za svaki test slučaj . Ukoliko su test procedure automatizovane izvršavaju se test komponente.

b) Upoređuju se rezultati testiranja sa očekivanim rezultatima i traže se devijacije.

c) Ukoliko se pronađu defekti kod testiranja o njima se obaveštava inženjer komponenti.

d) Defekti se takođe prosleđuju do test projektanta, koji tada koristi defekte da evaluira rezultate testiranja.

Sistemsko testiranje

Sistemski testovi se koriste, na kraju svake iteracije. Ono može da započne kada se integracioni testovi koji se obavljaju u toj iteraciji pokažu kao validni. Tada se build, koji je dobijen kao rezultat iteracije, testira u kontekstu različitih sistemskih uslova. Ovi uslovi se odnose na:

a) hardverske konfiguracije (procesore, operativnu memoriju, hard diskove,…),
b) operativne sisteme,
c) veličinu baze podataka,
d) broj aktora koji koriste sistem,…, itd.

Koraci kod izvršenja sistemskog testiranja su analogni po formi sa integracionim testiranjem.

Testiranje evaluacije

Za svaki test se prati njegova evaluacija (razvoj). Test projektant prati rezultate testiranja i upoređuje ih sa ciljnim rezultatima. On treba da pripremi mere vezane za kompletnost testiranja i pouzdanost testiranog sistema, koje će da odrede nivo kvaliteta testiranog softvera. Test projektant pravi dijagrame koji prate pojavu defekata kod testiranja.

4.5 Studijski Primer Poslovnog Sistema Prodaje robe

4.5.1 SPPSP – STUDIJSKI PRIMER – POSLOVNI SISTEM

SPPS 1: ***Korisnički zahtev***

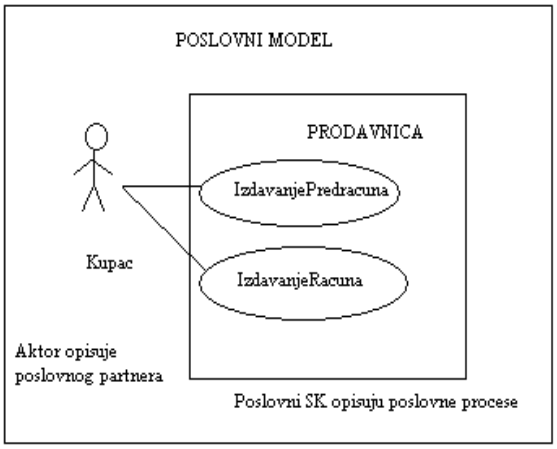
Omogućiti obradu i izdavanje predračuna i računa za kupca.

SPPS 2: ***Realni poslovni sistem***

Na osnovu navedenog zahteva uočavaju se dva poslovna procesa: Izdavanje predračuna i Izdavanje računa kao i jedan poslovni partner: Kupac

SPPS 3: ***Poslovni model***

Realni poslovni sistem se modelira pomoću dijagrama SK (Slika PM).



Slika PM: Poslovni model predstavljen preko dijagrama SK

Dijagram SK sadrži dva poslovna SK : *Izdavanje predračuna* i *Izdavanje računa* i jednog aktora: *Kupac*.

Realizacije navedenenih poslovnih SK modeliraćemo pomoću verbalnog opisa SK.

SPPS 4: ***Poslovni slučaj korišćenja – Izdavanje predračuna***

Naziv SK

Izdavanje predračuna

Učesnici SK

Kupac i prodavnica.

Aktori SK

Kupac

Preduslov*:*

Prodavnica je otvorena i spremna da usluži kupca.

Osnovne aktivnosti

1. Kupac ulazi u prodavnicu i traži da mu se izda predračun ili zove telefonom prodavnicu da mu se predračun pošalje preko faksa, za robu koju će da kupi.
2. Prodavnica pravi predračun.
3. Prodavnica šalje predračun do kupca. Jedan primerak predračuna prodavnica zadržava za sebe.
4. Kupac prima predračun od prodavnice.

Postuslov

Slučaj korišćenja je završen kada je kupac primio predračun od prodavnice.

Alternativne aktivnosti

1. Ukoliko kupac odustane od zahteva za predračunom prekida se poslovni SK.

Opis interakcija sistema sa aktorima
1. Kupac traži od prodavnice da mu izda ili pošalje predračun za robu koju će da kupi.
2. Prodavac daje predračun kupcu.

Opis korišćenih objekata sistema
1. Poslovni objekti
- *a) prodavnica (org. jedinica) u kojoj se nalazi prodaje roba.*
- *b) predračun (poslovni entitet) koji prate proces prodaje robe.*

2. Vrednosti sistema: Napravljeni predračun

Odgovornost učesnika
1. Prodavnica pravi i daje predračun kupcu.

SPPS 5: *Poslovni slučaj korišćenja – Izdavanje računa*

Naziv SK
Izdavanje računa

Učesnici SK
Kupac i prodavnica.

Aktori SK
Kupac

Preduslov:
Prodavnica je otvorena i spremna da usluži kupca.

Osnovne aktivnosti
1. Kupac ulazi u prodavnicu i traži da mu se izda račun za kupljenu robu, jer je platio predračun za tu robu.
2. Prodavnica, pregledom izvoda, kontroliše da li je kupac izvršio plaćanje po određenom predračunu.
3. Prodavnica pravi račun (na osnovu predračuna).
4. Prodavnica daje račun kupcu. Jedan primerak računa prodavnica zadržava za sebe.
5. Kupac prima račun od prodavnice.

Postuslov
Slučaj korišćenja je završen kada je kupac primio račun od prodavnice ili kada je kupac odustao od kupovine.

Alternativne aktivnosti
1. Ukoliko kupac nije izvršio uplatu po predračunu, aktivnost 2, prekida se poslovni SK.

Opis interakcija sistema sa aktorima
1. Kupac traži od prodavnice da mu izda račun za kupljenu robu
2. Prodavnica daje račun kupcu.

Opis korišćenih objekata sistema
1. Poslovni objekti
- *a) prodavnica (org. jedinica) u kojoj se nalazi i prodaje roba.*
- *b) račun, predračun i izvod(poslovni entiteti) koji prate proces prodaje robe.*

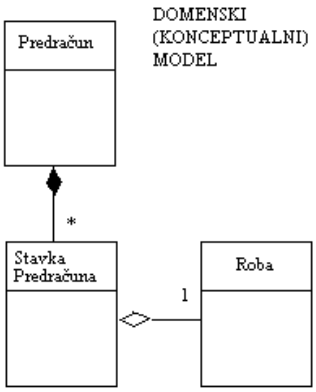
2. Vrednosti sistema: Napravljeni račun

Odgovornost učesnika
1. Prodavnica pravi i daje daje račun kupcu.

Kasnije ćemo videti, kod opisa sistemskih SK da je korisnik sistema prodavac (radnik koji radi u prodavnici), dok je sistem program koji prodavac koristi.

SPPS 6: Domenski model

Na osnovu korisničkog zahteva dobijaju se osnovni koncepti sistema[76], koji su predstavljeni preko sledećeg dijagrama klasa (Slika DM):



Slika DM: Domenski model predstavljen preko dijagrama klasa

SPPS 7: *Rečnik termina*

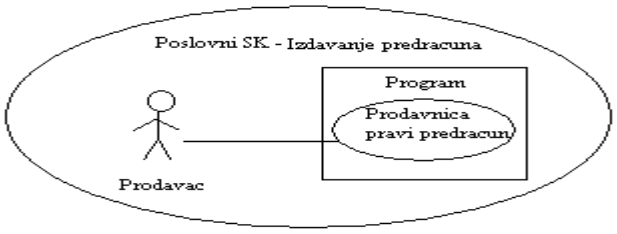
Na osnovu poslovnog i domenskog modela pravi se rečnik termina:
Kupac, Predračun, StavkaPredračuna, Račun, Prodavnica, IzdavanjePredracuna, IzdavanjeRacuna.

4.5.2 SPPZ – STUDIJSKI PRIMER - PRIKUPLJANJE ZAHTEVA

SPPZ1: *Model slučaja korišćenja*

Iz poslovnih SK se izdvavaju one aktivnosti koje se mogu automatizovati preko programa:

1. *Prodavnica pravi predračun.*



2. Prodavnica pravi račun.

SPPZ2*: Sistemski slučaj korišćenja – Prodavnica pravi predračun*

Naziv sistemskog SK
Prodavnica pravi predračun.

Učesnici SK
Prodavac i program (u daljem tekstu sistem).

Aktori SK
Prodavac

[76] Pošto su predračun i račun predstavljeni istim dokumentom, nećemo posebno razmatrati račun kao koncept.

Preduslov:

1. Sistem je uključen i prodavac je ulogovan pod svojom šifrom. Sistem prikazuje formu za obradu predračuna.

Osnovne aktivnosti SK

1. *Prodavac prilazi do sistema.*
2. *Prodavac kreira novi predračun i unosi podatake u njega.*
3. *Prodavac zahteva od sistema da zapamti predračun. Sistem pamti novi predračun.*
4. *Sistem javlja da je zapamtio predračun.*
5. *Prodavac zahteva da sistem odštampa predračun u 2 primerka.*
6. *Sistem štampa predračun na štampaču.*
7. *Prodavac uzima oštampane predračune iz štampača i predaje ih kupcu.*

Osnovne aktivnosti SK se mogu prikazati i na sledeći način:

Aktivnosti aktora	***Aktivnosti (odgovor) sistema***
1. Prodavac prilazi do programa.	
2.Prodavac kreira novi predračun i unosi podatake u njega.	
3. Prodavac zahteva od sistema da zapamti predračun.	*3.1. Sistem pamti novi predračun.*
	4. Sistem javlja da je zapamtio predračun.
5. Prodavac zahteva da sistem odštampa predračun u 2 primerka.	*6. Sistem štampa predračun na štampaču.*
7. Prodavac uzima oštampane predračune iz štampača i predaje ih kupcu.	

Postuslov

Slučaj korišćenja je završen kada prodavac uzme oštampane predračune.

Alternativne aktivnosti

- *Ukoliko uneti podaci na predracunu nisu dobri, oni se menjaju ili brisu.*

Opis korišćenih objekata sistema

1. Poslovni objekti

a) prodavac (radnik)
b) predračun (poslovni entitet)
c) program – sistem (osnovno sredstva)

2. Vrednosti sistema: Oštampani predračun

Odgovornost učesnika

Odgovornosti prodavca

- *prilazi do sistema.*
- *kreira novi predračun i unosi podatake u njega.*
- *zahteva od sistema da zapamti predračun.*
- *zahteva da sistem odštampa predračun u 2 primerka.*
- *uzima oštampane predračune iz štampača.*

Odgovornost sistema

- *prikazuje formu za obradu predračuna.*
- *pamti novi predračun.*
- *javlja da je zapamtio predračun.*
- *štampa predračun na štampaču.*
- *menja podatke o predračunu.*
- *briše podatke o predračunu.*

SPPZ2: *Prototipovi korisničkog interfejsa za sistemski slučaj korišćenja - Prodavnica pravi predračun*

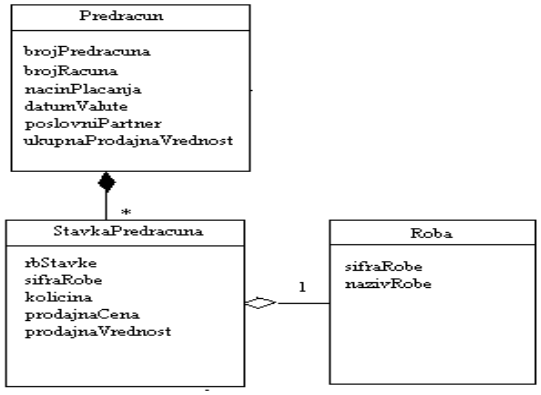
Šifra:PR01
Opis: Forma za obradu predračuna.
Prototip treba da sadrži sledeća polja: *brojPredracuna, nacinPlacanja, datumValute, poslovniPartner, ukupnaProdajnaVrednost, rbStavke, sifraRobe, kolicina, prodajnaCena, prodajnaVrednost;*

Šifra:PRI02
Opis: Sistem štampa predračun.
Prototip treba da sadrži sledeća polja: *brojPredracuna, nacinPlacanja, datumValute, poslovniPartner, ukupnaProdajnaVrednost, rbStavke, sifraRobe, kolicina, prodajnaCena, prodajnaVrednost;*

SPPZ3*: Sistemski slučaj korišćenja – Prodavnica pravi račun*
Na sličan način se radi kao predhodni SK. U daljem tekstu nećemo pratiti razvoj SK Prodavnica pravi račun.

SPPZ4: *Rečnik termina*
Na osnovu poslovno – sistemskih SK: *Prodavnica pravi predračun i Prodavnica pravi račun* dobijaju se novi termini,koji se dodaju postojećim:
Postojeći termini:
Kupac, Predračun, StavkaPredračuna, Račun, Prodavnica, IzdavanjePredracuna, IzdavanjeRacuna.
Novi termini:
Prodavac, Sistem, brojRačuna, brojPredracuna, nacinPlacanja, datumValute, poslovniPartner, ukupnaProdajnaVrednost, rbStavke,sifraRobe, kolicina, prodajnaCena, prodajnaVrednost
U nastavku studijskog primera nećemo pratiti razvoj rečnika termina.

SPPZ5: *Domenski model sa atributima*
Postojećim konceptima dodaju se atributi, što je predstavljeno sledećim dijagramom klasa (Slika DMA):



Slika DMA: Domenski model sa atributima predstavljen preko dijagrama klasa

4.5.3 SPAN – STUDIJSKI PRIMER – ANALIZA

Ponašanje modela analize je opisano preko realizacije SK *Prodavnica pravi predračun* i *Prodavnica pravi račun*. Na osnovu njih se izvode dijagrami klasa ponašanja. Nakon toga su dati specijalni zahtevi. Struktura modela analize je opisana preko domenskog modela čiji entiteti imaju tipizirane atribute. Na kraju su dati paketi analize.

SPAN1:*Realizacija SK – Prodavnica pravi predračun*

SPAN1.1:Tekstualni opis osnovnog i alternativnih scenaria SK – Prodavnica pravi predračun

Dati su tekstualni opisi osnovnog scenaria SK i alternativnih scenaria kada se menja predračun i kada se briše predračun.

SPAN1.1.1: TEKSTUALNI OPIS OSNOVNOG SCENARIA SISTEMSKOG SK

- Sistem je uključen i prodavac je ulogovan pod svojom šifrom. Sistem prikazuje formu za obradu predracuna. Prodavac prilazi do sistema. Prodavac kreira novi predračun i u njega unosi podatke. Nakon toga prodavac zahteva od sistema da pamti predračun. Sistem pamti predračun. Sistem javlja da je zapamtio predračun. Prodavac zahteva da sistem oštampa predračun u 2 primerka. Sistem štampa predračun na štampaču. Prodavac uzima oštampane predračune iz štampača i daje ih kupcu.

Ukoliko uneti podaci na predracunu, nakon pamćenja nisu dobri, oni se menjaju ili brišu po sličnom scenariju kako se pamti predračun.

SPAN1.1.2: TEKSTUALNI OPIS ALTERNATIVNOG SCENARIA SK - PROMENA PREDRAČUNA

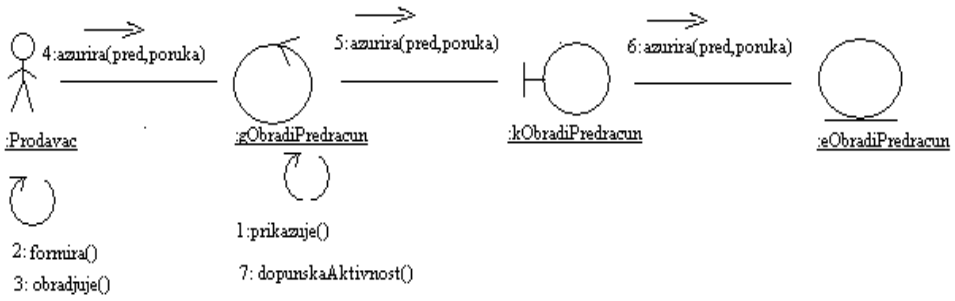
- *Sistem je uključen i prodavac je ulogovan pod svojom šifrom. Sistem prikazuje formu za obradu predracuna. Prodavac prilazi do sistema. Prodavac bira postojeći predračun i u njega unosi podatke. Nakon toga prodavac zahteva od sistema da menja predračun. Sistem menja predračun. Sistem javlja da je promenio predračun. Prodavac zahteva da sistem oštampa predračun u 2 primerka. Sistem štampa predračun na štampaču. Prodavac uzima oštampane predračune iz štampača i daje ih kupcu.*

SPAN1.1.3:TEKSTUALNI OPIS ALTERNATIVNOG SCENARIA SK - BRIŠI PREDRAČUN

- *Sistem je uključen i prodavac je ulogovan pod svojom šifrom. Sistem prikazuje formu za obradu predracuna. Prodavac prilazi do sistema. Prodavac bira postojeći predračun. Nakon toga prodavac zahteva od sistema da briše predračun. Sistem briše predračun. Sistem javlja da je obrisao predračun.*

SPAN1.2:Dijagram saradnje SK – Prodavnica pravi predračun

Kao rezultat analize dobija se generalni scenario sistemskog SK, za koji se pravi dijagram saradnje(Slika DSSK).



Slika DSSK: Dijagram saradnje sistemskog SK – Prodavnica pravi predračun

SPAN1.3:Ugovori sistemskih operacija SK – Prodavnica pravi predračun

Za SK *Prodavnica pravi predračun* se uočavaju sistemske operacije i za njih se prave ugovori[77].

SO1

Ime sistemske operacije: ***kreirajEkranskuFormuiObjekatKojiCeIspunitiFormu():boolean***

Odgovornost: Kreira ekransku formu na kojoj ce dokumenti biti prikazani. Kreira objekat koji ce cuvati podatke o tekucem dokumentu i tekucoj stavci koja se nalazi na ekranskoj formi.

[77] Koncept ugovora je preuzet od Larmana. On ne postoji kod JPRS. Smatramo da isti treba da postoji kako bi se precizno shvatila specifikacija svake od sistemskih operacija.

Postuslovi: Operacija je uspesno izvrsena ukoliko su uspesno kreirani ekranska forma i objekat koji cuva podatke o tekucem dokumentu i tekucoj stavci.

SO2
Ime sistemske operacije:
pretražiDokumente(pred:Predracun ,por:Poruka ,broj:integer, ukupnoStavki:integer):boolean

Odgovornost: Traži predračun na osnovu zadatog parametra pred.

SO3
Ime sistemske operacije:
pretraziStavke(pred:Predracun,por:Poruka, brojDok:integer,brojStavke:integer):boolean

Odgovornost: Pretrazuje postojece stavke zadatog predracune na osnovu zadate stavke predracuna.

Preduslov: Postojece stavke se pretrazuju ukoliko postoji zadati predracun.

SO4
Ime sistemske operacije: ***kreirajDokument(pred:Predracun[78],po:Poruka):boolean***
Odgovornost: Ukoliko predracun koji se kreira ne postoji on se pamti kao novi predracun. Ukoliko postoji njemu se menjaju vrednosti shodno prosledjenom predracunu.
Ukoliko tekuća stavka koja treba da se azurira ne postoji ona se pamti kao nova stavka zadatog predracuna. Ukoliko postoji njoj se menjaju vrednosti shodno prosledjenoj stavci.

SO5
Ime sistemske operacije: ***brisiDokument(pred:Predracun, po:Poruka):boolean***
Odgovornost: Brise se zadati predracun iz transijentne baze (objektna baza) i perzistentne baze (relaciona baza).
Preduslov: Pre brisanja zadatog predracuna mora da se proveri da li on postoji.

SO6
Ime sistemske operacije: ***brisiStavkuDokumenta(pred:Predracun, po:Poruka):boolean***
Odgovornost: Brise se zadati stavka zadatog predracun iz transijentne i perzistentne baze .

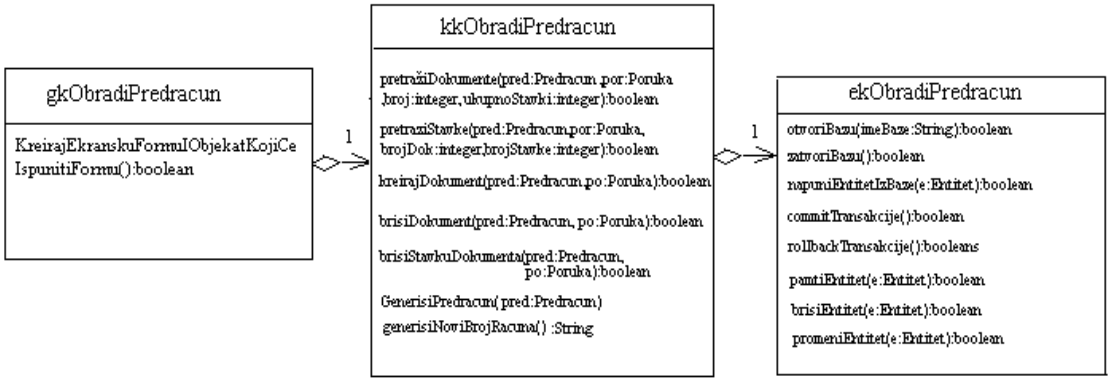
Preduslov: Pre brisanja zadate stavke zadatog predracuna mora da se proveri da li postoji zadati predracun i zadata stavka.

SO7
Ime sistemske operacije: ***GenerisiPredracun(pred:Predracun)***
Odgovornost: Generise HTML datoteku na osnovu zadatog predracuna pred.
Izlaz: Ukoliko se desi izuzetak prekida se generisanje predracuna.

[78] Predračun koji se kreira sadrži tekuću aktivnu stavku. To znači da predračun i njegova jedna stavka predstavljaju ulaz u operaciju kreirajDokument().

SPAN1.4: Dijagram klasa ponašanja

Na osnovu dijagrama saradnje prave se dijagrami klasa ponašanja(Slika DKP).



Slika DKP: Dijagram klasa ponašanja

SPAN1.4.1 :KLASE PONAŠANJA

Na osnovu dijagrama klasa ponašanja mogu se identifikovati sledeće entitetske klase ponašanja:

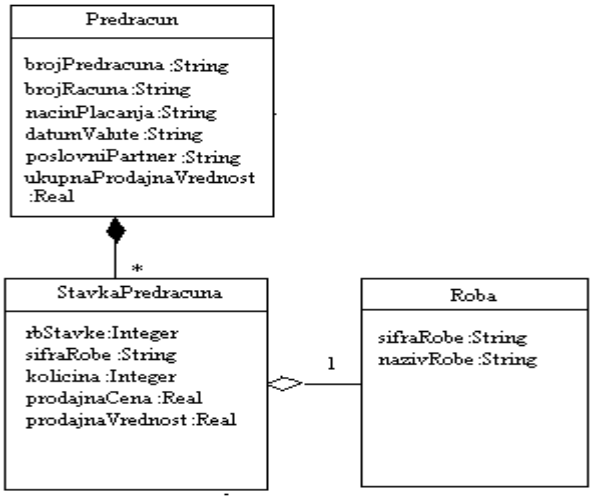
1. *GkObradiPredracun koja je odgovorna za korisnički interfejs.*
2. *KkObradiPredracun koja je odgovorna za poslovnu logiku*
3. *EkObradiPredracun koja je odgovorna za rad sa bazom podataka.*

SPAN1.5:Specijalni zahtevi

1. Obezbediti da klase Predracun, StavkePredracuna i Roba budu perzistentne. Preciznije rečeno omogućiti da se objekti navedenih klasa čuvaju u relacionoj bazi.
2. Omogućiti transakcioni mehanizam pri izvršenju sistemskih operacija.
3. Omogućiti da više korisnika istovreme može pristupiti programu.
4. Obezbediti mehanizam zaključavanja objekata.
5. Napraviti program koji će po svojoj konceptualnoj arhitekturi biti sličan J2EE arhitekturi.

SPAN2:*Domenski model sa tipiziranim atributima*

Za svaki atribut domenskih entiteta određuje se konceptualni tip (Slika DMTA).



Slika DMTA: Domenski model sa tipiziranim atributima

SPAN2.1:Relacioni model

Na osnovu domenskog modela pravi se relacioni model:

Predracun(brojPredracuna,brojRacuna,nacinPlacanja,datumValute,poslovniPartner,ukupnaProdajna Vrednost)

StavkePredracuna(brojPredracuna,rbStavke,sifraRobe,kolicina,prodajnaCena,prodajnaVrednost)

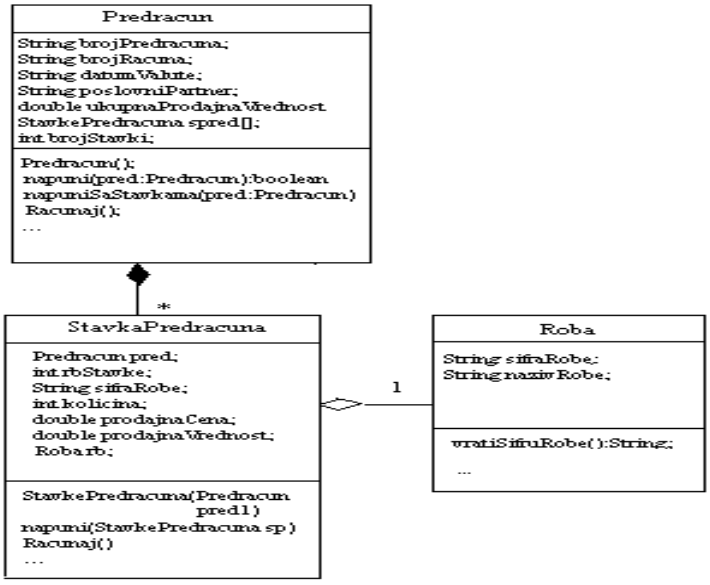
Roba(sifraRibe,nazivRobe)

Podvučeni su primarni ključevi relacija.

U ovom studijskom primeru nećemo da ulazimo u detaljnu specifikaciju relacionog modela[79].

SPAN2.2:Dijagram klasa strukture

Na osnovu domenskog modela pravi se dijagram klasa strukture(Slika DKS):



Slika DKS: Dijagram klasa strukture - analiza

SPAN2.2.1:KLASE STRUKTURE

Dijagram klasa se sastoji iz sledećih klasa strukture:

```
class Predracun
 {  String brojPredracuna;
    String brojRacuna;
    String datumValute;
    String poslovniPartner;
    double ukupnaProdajnaVrednost;
    StavkePredracuna spred[];
    int brojStavki;

   Ime: Predracun() - Konstruktor metoda klase Predracun
   Odgovornost: Pri kreiranju novog predracuna odvaja se prostor za  stavke predracuna, dok se atribut
   brojStavki postavlja na 0.

  Ime: napuni(Predracun pred):boolean
  Odgovornost: Predracun se puni sa vrednoscu pred.
  Izlaz: Ukoliko se punjenje predracuna izvrsi uspesno metoda vraca true,
  inace metoda vraca false.

Ime: napuniSaStavkama(Predracun pred)
Odgovornost: Puni stavke predracuna sa stavkama od pred.
```

[79] U našim istraživanjima mi smo se bavili relaciono objektnim modelom i u tom smislu smo napisali nekoliko radova[DeSt, Sv1-4, Sv7,Sv9-10,Cir8,Cir11] .

```
Izlaz: Ukoliko se punjenje stavki predracuna izvrsi uspesno metoda vraca true, inace
metoda vraca false.

Ime: Racunaj()
Odgovornost: Racuna ukupnu vrednost svih stavki predracuna prema formuli:
ukupnaProdajnaVrednost = stavka[1].prodajnaVrednost + stavka[2].prodajnaVrednost +
 ... + stavka[n].prodajnaVrednost, gde n - predstavlja broj stavki.

// Pocetak skupa operacija koje pune (set) i vracaju (get) vrednosti atributa**********
String vratiBrojPredracuna();
String vratiBrojRacuna();
String vratiDatumValute();
String vratiPoslovniPartner();
double vratiUkupnaProdajnaVrednost();

void napuniBrojPredracuna(String brojPredracuna1);
void napuniBrojRacuna(String brojRacuna1);
void napuniDatumValute(String datumValute1);
void napuniPoslovniPartner(String poslovniPartner1);
void napuniUkupnaProdajnaVrednost(double ukupnaProdajnaVrednost1);

int vratiBrojStavki();
void napuniBrojStavki(int brojStavki1);
// Kraj skupa operacija koje pune (set) i vracaju (get) vrednosti atributa**********
}

class StavkePredracuna
{ Predracun pred;
  int rbStavke;
  String sifraRobe;
  int kolicina;
  double prodajnaCena;
  double prodajnaVrednost;
  Roba rb;

  Ime: StavkePredracuna(Predracun pred1) – konstruktor klase StavkePredracuna
  Odgovornost: Pri kreiranju novog predracuna od dobija referencu na  predracun koji je vec kreiran a
  koji  se prenosi kao argument.

 Ime: napuni(StavkePredracuna sp)
 Odgovornost: Stavka predracuna se puni sa vrednoscu sp.
 Izlaz: Ukoliko se punjenje predracuna izvrsi uspesno metoda vraca true, inace metoda vraca false.

Ime: Racunaj()
Odgovornost: Racuna vrednost stavke predracuna na osnovu formule:
prodajnaVrednost = prodajnaCena * kolicina;
// Pocetak skupa operacija koje pune (set) i vracaju (get) vrednosti atributa**********
… Slično kao kod klase Predracun
// Kraj skupa operacija koje pune (set) i vracaju (get) vrednosti atributa**********
}

class Roba
{ String sifraRobe;
  String nazivRobe;

// Pocetak skupa operacija koje pune (set) i vracaju (get) vrednosti atributa**********
… Slično kao kod klase Predracun
// Kraj skupa operacija koje pune (set) i vracaju (get) vrednosti atributa**********
}
```

Svaka od navedenih klasa ima atribute i operacije. Elementarne operacije koje pune i vraćaju vrednosti atributa definisali smo preko njihovih naziva (npr. vratiBrojPredracuna), dok smo malo složenije operacije definisali preko ugovora(npr. Ime: Racunaj() Odgovornost: Racuna vrednost stavke predracuna na osnovu formule: prodajnaVrednost = prodajnaCena * kolicina;).

SPAN3:Paketi analize

Postoje sledeći paketi analize:

1. Klasa gkObradiPredracun postaje paket odgovoran za komunikaciju sa klijentom.
2. Klasa kkObradiPredracun postaje paket odgovoran za poslovnu logiku sistema.
3. Klasa ekObradiPredracun postaje paket odgovoran za komunikaciju sa bazom podataka.
4. Klase gkObradiPredracun, kkObradiPredracun i ekObradiPredracun zajedno čine paket KlaseAnalizePonašanja, koji je odgovoran za ponašanje sistema.
5. Dijagram klasa strukture postaje paket koji je odgovoran za definisanje objektne strukture sistema.
6. Relacioni model postaje paket koji je odgovoran za definisanje relacione strukture sistema.
7. Realizacija SK postaje paket koji je odgovoran za definisanje sistemskih operacije.

4.5.4 PR7: SPPR – **S**tudijski **P**rimer – **Pr**ojektovanje[80]

SPPR1: *Arhitektura softverskog sistema*

U okviru arhitekture okruženja sistema ukazaćemo na sistemski softver, middleware generičke mehanizme, načine distribucije objekata, nasleđene sisteme i standarde razvoja sistema koji će biti izabrani u toku projektovanja sistema.

SPPR1.1: Projektovanje sloja sistemskog softvera

Sistemski softver se razmatra u kontekstu operativnog sistema, programskog jezika i SUBP koji će biti izabran.

Izbor operativnog sistema i programskog jezika

Softverski sistem – Prodaja robe treba da ima mogućnost izvršavanja na bilo kom operativnom sistemu (Unix, Solaris, NT4.0,Win2000). Da bi se navedeni zahtev rešio potrebno je softverski proizvod (program) napisati u programskom jeziku, koji je nezavistan od operativnog sistema na kome će se izvršiti. Suština je da se jednom napisan program može transparentno prenositi (bez promene) na razne operativne sisteme.

Navedeni problem može formalno da se opiše na sledeći način, shodno pravilu PJVR[SV24]:

Ukoliko se između programskog jezika (PJ) i operativnog sistema (OS), uspostavi relacija:
Rst(PJ,OS)

javlja se problem jake povezanosti između njih, u kontekstu razvoja programa na nekom programskom jeziku[81]. To znači da je programski jezik vezan samo za jedan operativni sistema. Ukoliko se želi da programski jezik bude vezan za više operativnih sistema:

$\rightarrow$

$R(PJ,OS), OS \in (Unix,Solaris,NT4.0,Win2000,...)$

tada je potrebno izvršiti proces simetrizacije nad navedenom relacijom, na osnovu preporuke PKIPS. Kao rezultat simetrizacije dobiće se sledeće relacije:

$Rst(PJ,APJ) \wedge Rsg(APJ,OS)$

Java programski jezik, zadovoljava navedene relacije kada se stavi umesto AJP. To znači da se Java programski jezik može izvršiti na više operativnih sistema. Specifikacija Java jezika predstavlja zajednički interfejs za bilo koji operativni sistem. Realizaciju tog interfejsa rade operativni sistemi pomoću njihove JVM (Java Virtual Machine), koja interpretira Java programe[82] [Java1-6].

Napomena: Drugi programski jezici mogu takođe da se izvršavaju na više operativnih sistema, međutim prebacivanje programa (npr. napisanog u C++ programskom jeziku) sa jednog na drugi operativni sistem zahteva promenu toga programa, što nije slučaj kod Jave.

Izbor sistema za upravljanje bazom podataka

Sistem za upravljanje bazom podataka (SUBP), za navedeni primer nije toliko bitan, jer ključne funkcionalnosti programa vezane za perzistentnost i transakcije obezbediće programski jezik Java. To znači da program koji se napiše u Javi, a koji je vezan je za rad sa bazom podataka, može nepromenljiv da se koristi za bilo koji SUBP[83].

Napomena**:** APJ u razmatranoj relaciji takođe zadovoljavaju i svi oni jezici (C++, C#, Visual Basic,…) koji sadrže ADO(ActiveX Date Objects) API (Application Programming Interface). ADO API predstavlja zajednički interfejs za bilo koji SUBP. Realizaciju tog interfejsa rade ADO drajveri SUBP.

SPPR1.2: Projektovanje middleware sloja

Postoje sledeći middleware generički mehanizmi projektovanja koji treba da se koriste:

- Perzistentnost objekata i obrada transakcija treba da bude obrađena preko Javinih klasa koje se nalaze u paketu **java.sql.*.** Oporavak sistema biće direktno zasnovan na Javinom transakcionom mehanizmu.

- Komunikacija između klijentskih i serverskog programa treba da bude ostvarena pomoću soketa. Klase koje podržavaju rad sa soketima, kod Jave se nalaze u paketu **java.net.*.**

---

[80] U izradi ovog studijskog primera mi smo koristili refactoring princip (dobijanje modela projektovanja iz implementiacje) o kome je pisao Martin Fowler *[Refact]*. To praktično znači da smo posle analize prvo uradili implementaciju a onda smo inverzno došli do modela projektovanja.

[81] To praktično znači da program napisan na nekom programskom jeziku, koji je jako vezan za neki operativni sistem, ne bi mogao nepromenjiv da se izvrši na nekom drugom operativnom sistemu.

[82] Preciznije rečeno JVM interpretira bajt kod oblik programa, koji se dobija nakon kompajliranja Javinog izvornog programa.

[83] Jedino se menja naredba koja uspostavlja konekciju prema drajveru izabrane baze.

- Podaci koji se budu razmenjivali između klijentskih i serverskog programa treba da budu serijalizovani. Klasa koja podržava serijalizaciju objekata, kao i klase pomoću kojih se izvršavaju ulazno-izlazne operacije klijentskih i serverskog programa nalaze se u paketu **java.io.*.**
- Korisnički interfejs treba da bude obrađen preko Javinih klasa koje se nalaze u paketima **java.awt.*** i **java.applet.Applet.** Događaji[84] treba da budu obrađeni korićenjem Javinog paketa **java.awt.event.*.**
- Greške, odnosno izuzeci koji se budu javljali u programu treba da budu obrađeni preko Javinog mehanizma obrade izuzetaka, čije se klase nalaze u Javinom paketu **java.lang.**

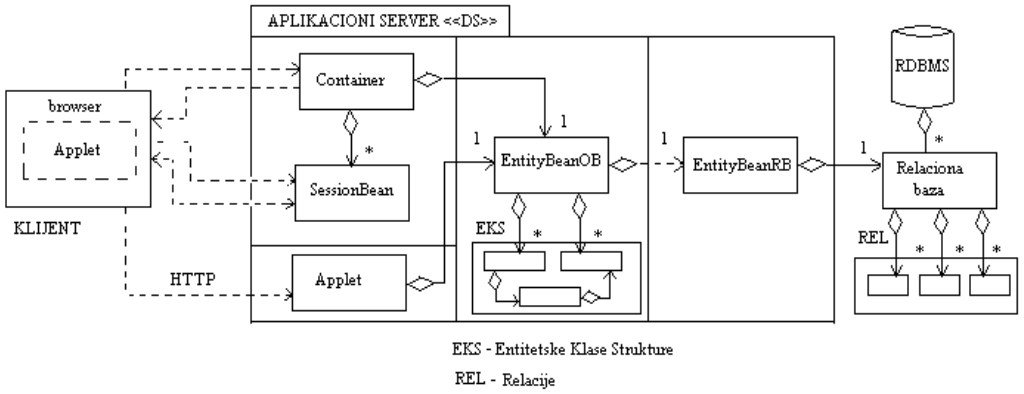
Navedeni Javini paketi detaljno su obrađeni u [Java1-6].

SPPR1.3: Projektovanje aplikacionog sloja – distribucija objekata

Distribucija objekata biće organizovana na principu ECF uzora. U tom smislu jednostavni aplikacioni server koji budemo razvijali, biće u skladu sa osnovnim principima J2EE tehnologije.

SPPR1.3.1 STRUKTURA APLIKACIONOG SERVERA

Aplikacioni server treba da ima sledeću strukturu (Slika SAS):



Slika SAS: Struktura aplikacionog servera

Objekti aplikacionog servera su:

***1. Container** je objekat koji ima ulogu da:*

a) Puni EntityBeanOB objekat iz relacione baze.
b) Kreira niz koji treba da cuva session bean-ove.
c) Kreira soket koji ce povezati aplikacioni server sa klijentima.
d) Kreira objekat kontrole rada aplikacionog servera. (Klijenti pre nego sto se povezu sa aplikacionim serverom proveravaju da li se aplikacioni server uopste izvrsava).
e) Povezuje klijente sa aplikacionim serverom a zatim za svakog klijenta generiše session bean objekat, koji u daljem radu programa preuzima odgovornost za komunikaciju sa klijentom. Container objekat takođe postavlja kontekst session bean objektima.

Kontekst Container objekta čine:

- *SessionBean objekti koje je on generisao.*
- *EntityBeanOB objekat.*

***2. SessionBean** je objekat koji ima ulogu da:*

a) Prihvata zahtev od klijenta da izvrši sistemsku operaciju.
b) Preusmerava zahtev za izvršenje sistemske operacije do EntityBeanOB objekta.
c) Ukoliko je izvršena sistemska operacija promenila stanje baze, javlja ostalim session bean objektima (klijentima) da osveže svoju lokalnu strukturu podataka.

Kontekst Session bean objekta čine:

- *Registracioni broj session bean-a.*
- *Kontekst aplikacionog servera.*
- *Informacije o Container objektu koji ga je generisao.*
  - *Soket S1 koji ostvaruje konekciju tekuceg sessin bean-a sa njegovim klijentom. On se koristi da: a) prihvati zahtev od klijenta za izvrsenje operacije i b) pošalje kontekst aplikacionog servera do klijenta.*
  - *Soket S2 takodje ostvaruje konekciju tekuceg sessin bean-a sa njegovim klijentom. On se koristi da pošalje zahtev do klijenta da osveži svoju lokalnu strukturu podataka.*

***3. EntityBeanOB** je objekat koji ima ulogu da:*

a) Realizuje sistemske operacije koje su definisane u kontrolnoj klasi KKObradaPredračuna iz analize.
b) Napuni svoj kontekst sa slogovima iz relacija koje se nalaze u relacionoj bazi.

[84] U našim istraživanjima vezanim za koncept događaja napisali smo jedan rad *[Sv21]* koji se bavi modelom događaja u MS Access-u.

Kontekst EntityBean objekta čine egzistencijalno nezavisne entitetske klase strukture (u našem primeru to su Predračun i Roba)[85]. Navedeni objekti, odnosno struktura podataka koja ih čuva (u našem primeru će to biti niz), reprezentuju slogove iz relacija[86].

4. EKS (Entitetske klase struktura) je podsistem koji sadrži sve klase strukture (u našem primeru to su Predračun, StavkePredračuna i Roba). Svaka od navedenih klasa sadrži svoju strukturu i interne operacije. Navedene klase su između sebe povezane preko objekata. Naglašavamo da je ovde data definicija klasa a ne struktura podataka koje će da čuvaju objekte tih klasa(EntityBeanOB objekat sadrži strukture podataka koje čuvaju objekte).

5. EntityBeanRB je objekat koji ima ulogu da omogući komunikaciju između EntityBeanOB objekta i relacione baze.

6. Applet je objekat koji ima ulogu da:

a) Poveže klijenta sa aplikacionim serverom.
b) Kreira niti koje prihvataju informacije od aplikacionog servera.
c) Kreira ekranske forme i objekta cije ce vrednosti atributa ispuniti graficke elemente (tekst polja, kombo boksovi,...) ekranske forme

d) Obradi predračun preko ekranske forma i da pošalje zahtev aplikacionom serveru da izvrši neku sistemsku operaciju (kreirajDokument, brisiDokument,…).

Kontekst Aplet objekta čine:

- Polja koja ce biti prikazana na formi.
- Dugmad koja ce biti prikazana na formi.
- Lokalna "Objektna Baza u programu" (Lokalna OB) sa kojom radi aplet(klijent).
- Soket koji: a) prihvata kontekst aplikacionog servera i b) šalje zahtev aplikacionom serveru da izvrsi sistemsku operaciju.
- Soket koji prihvata zahtev od aplikacionog servera da osvezi Lokalnu OB.
- Tekuci dokument koji se vidi na ekranskoj formi.
- Tekuca stavka koja se vidi na ekranskoj formi.
- Kontekst aplikacionog servera koji sadrži informacije o: a) novom stanju OB i b) operaciji koja je izvrsena na aplikacionom serveru.

SPPR1.3.2 OSNOVNA SCENARIJA U RADU APLIKACIONOG SERVERA

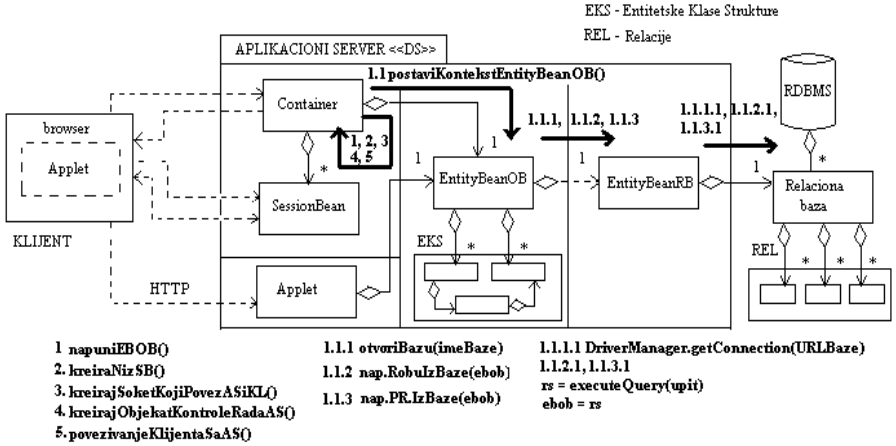
Postoje tri osnovna scenarija u radu aplikacionog servera:

1. Inicijalizacija aplikacionog servera (Slika IAS).
2. Povezivanje klijenta sa aplikacionim serverom (Slika PKAS).
3. Prihvatanje zahteva od klijenta za izvršenje neke od sistemskih operacija (Slika PZKSO).

S1: Inicijalizacija aplikacionog servera

Container objekat inicijalizuje aplikacioni server u 5 koraka:

a) *Kreira se EntityBeanOB objekat i isti se puni iz relacione baze(1). Punjenje se obavlja na sledeći način: EntityBeanOB objekat otvara bazu (1.1.1.) a zatim puni egzistencijalno nezavisne entitetske*



Slika IAS: Inicijalizacija aplikacionog servera

[85] U tom smislu se klasa StavkaPredračuna ne unosi, jer je ona egzistencijalno zavisna od klase Predračun.

[86] U daljem tekstu kada budemo koristili termin "objektna baza u programu" ili "objektna baza", mislićemo na strukturu podataka u programu, koja čuva objekte koji reprezentuju slogove relacija iz relacione baze. EntityBean objekat po definiciji ima ulogu da predstavi poslovne podatke (preko strukture podataka) aplikacije. U tom smislu smo Sufiks OB dodali do termina EntityBean, kako bi ukazali da se "objektna baza" čuva preko strukture podataka.

klase strukture[87] *(u našem primeru Robu i Predračun) iz baze(1.1.2,1.1.3). Navedene operacije EntityBeanOB objekat izvršava pomoću EntityBeanRB objekta, tako što EntityBeanRB objekat otvara bazu pomoću DriverManager-a (1.1.1.1), a navedene operacije punjenja Roba i Predračuna radi pomoću upita (1.1.2.1, 1.1.3.1) koji se izvšavaju nad relacionom bazom.*

b) Kreira se niz koji treba da cuva session bean-ove(2).

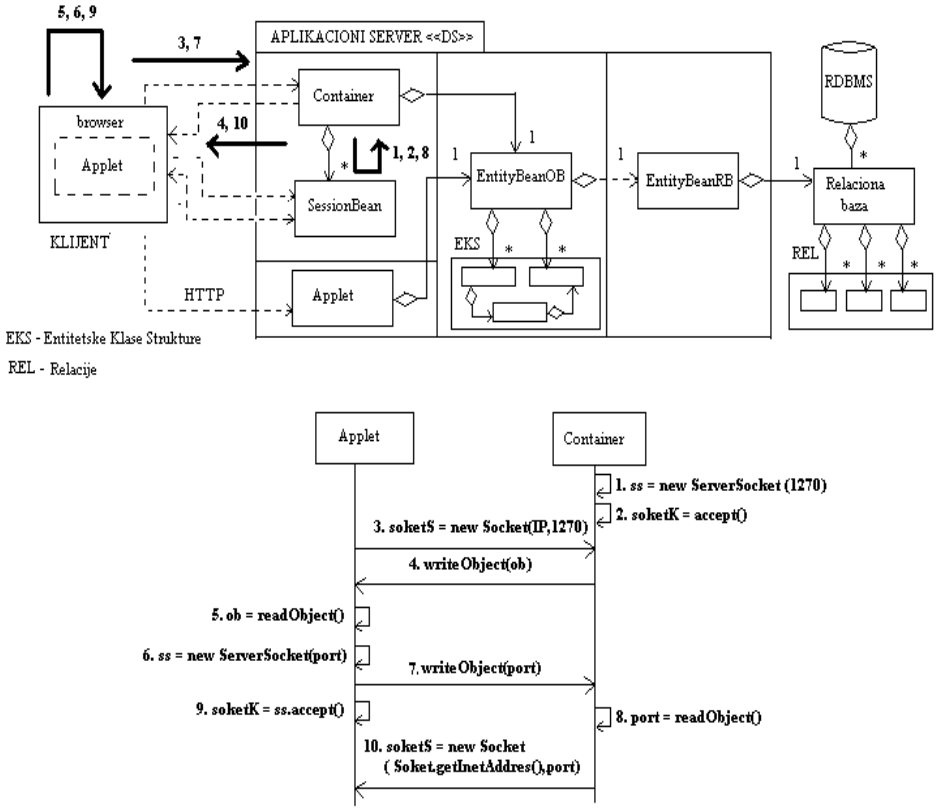
c) Kreira se serverski soket koji ce povezati aplikacioni server sa klijentima. Serverski soket se podiže na portu 1270(3).

d) Kreira objekat kontrole rada aplikacionog servera(4).

e) Povezuje klijente sa aplikacionim serverom (5). U navedenoj metodi aplikacioni server zaustavlja svoje izvršenje na naredbi (ss.accept(), ss – serverski soket) kada počinje da osluškuje mrežu, očekujući da se neki od klijenata poveže sa njim.

S2:Povezivanje klijenta sa aplikacionim serverom

Aplikacioni server je inicijalizovan. On podiže serverki soket ss na portu 1270 (1). Nakon toga on počinje da osluškuje mrežu(2). Aplet se povezuje sa aplikacionim serverom (3). Konekciju sa aplikacionim serverom aplet čuva u objektu SoketS. Konekciju sa apletom aplikacioni server čuva u objektu SoketK. Aplikacioni server šalje "objektnu bazu" (ob) do apleta (4). Aplet prihvata "objektnu bazu" i puni svoju lokalnu "objektnu bazu" (5). Aplet podiže svoj serverski soket (ss) na izabranom portu (7). Aplet šalje aplikacionom serveru broj navedenog porta (7). Aplikacioni server čita broj porta (8). Aplet počinje da osluškuje mrežu(9). Aplikacioni server se povezuje sa klijentovim serverskim soketom (10).Konekciju sa aplikacionim serverom aplet čuva u objektu SoketK. Konekciju sa apletom aplikacioni server čuva u objektu SoketS.



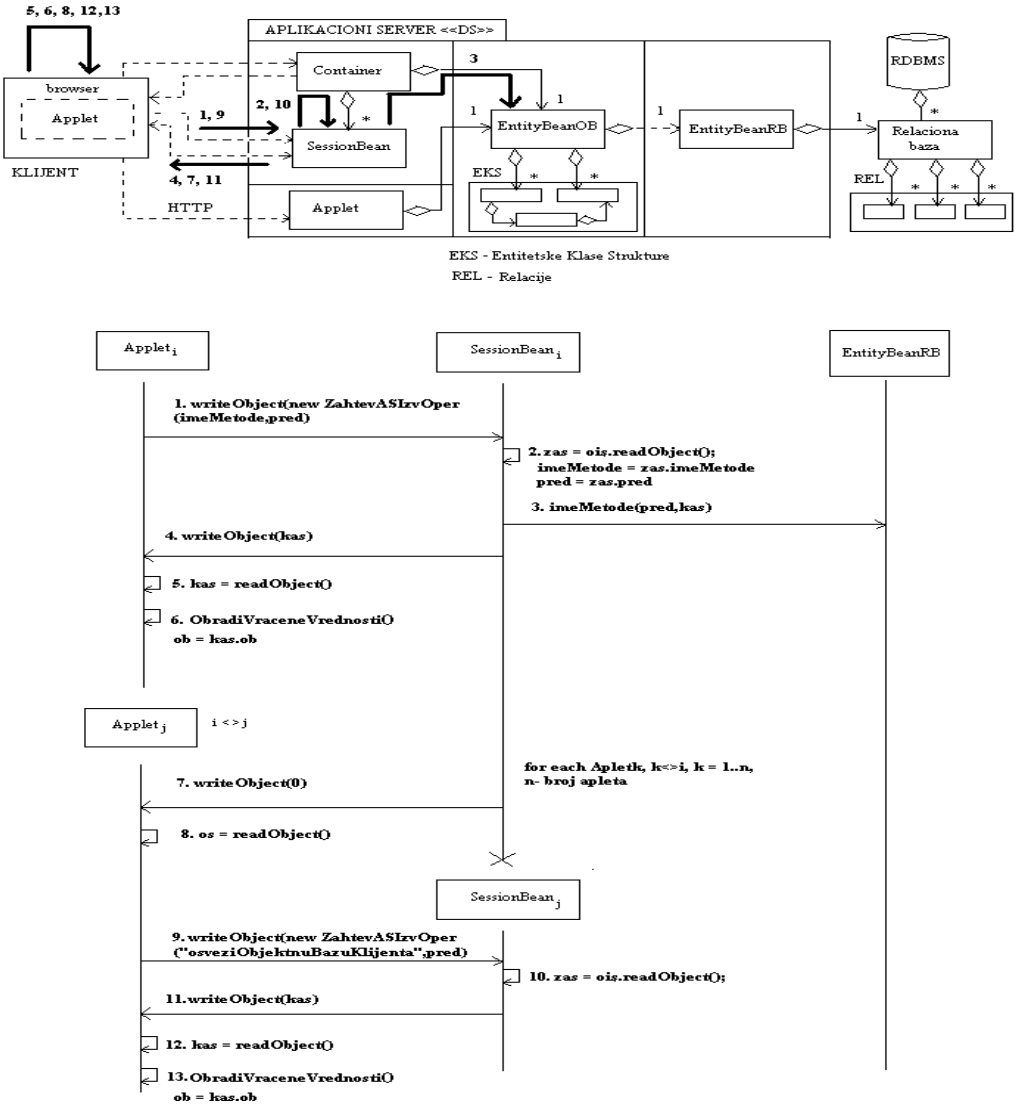
Slika PKAS: Povezivanje klijenta sa aplikacionim serverom

Iz navedenog scenaria može da se vidi da su ostvarene dve veze između apleta i aplikacionog servera. Preko prve veze (soketa), aplet šalje zahtev klijentu da izvrši neku od sistemskih operacija. Preko te iste veze aplet prihvata od aplikacionog servera njegov kontekst, nakon što je izvršena neka od sistemskih operacija. Preko druge veze aplet prima informaciju od aplikacionog servera da treba da izvrši osvežavanje svoje lokalne "objektne baze".

[87] Kada se puni egzistencijalno nezavisna klasa, automatski se pune sve klase koje su zavisne od nje, rekurzivno. To znači u našem primeru da kada se puni klasa Predračun, automatski se puni i klasa StavkaPredračuna.

S3: Prihvatanje zahteva od klijenta za izvršenje neke od sistemskih operacija

Klijent šalje zahtev aplikacionom serveru da izvrši sistemsku operaciju (zahtev sadrži ime sistemske operacije koja treba da se izvrši i podatke o predračunu (pred) koji treba da se obradi) (1). Aplikacioni



Slika PZKSO: Prihvatanje zahteva od klijenta za izvršenje neke od sistemskih operacija

server (session bean) prihvata zahtev (2) i preusmerava poziv sistemske operacije do EntityBeanOB objekta (3) šaljući mu kao parametre pred[88] i objekat kas (kontekst aplikacionog servera[89]). EntityBeanOB objekat izvršava sistemsku operaciju. Sistemska operacija može da promeni stanje baze podataka. Rezultat sistemske operacije se čuva u objektu kas. Aplikacioni server šalje kas objekat ka klijentu (4). Klijent ga prihvata (5) i na osnovu njega vrši ažuriranje svoje lokalne "objektne baze" (6), ako je promenjeno stanje baze podataka. Ukoliko je promenjeno stanje baze podataka, tekući session bean šalje informaciju svim klijentima da osveže svoje lokalne "objektne baze" (7).

[88] U našem primeru objekt pred (Predračun) se prenosi kao parametar. Međutim tu može, u opštem slučaju, biti bilo koji objekat koji treba da se obradi.

[89] Kontekst aplikacionog servera čine:

- Ime metode koja se izvrsava na aplikacionom serveru.
- Stanje EntityBeanOB objekta ("objektna baza u programu").
- Indeks tekuceg dokumenta kod EntityBeanOB objekta
- Indeks tekuce stavke tekuceg dokumenta kod EntityBeanOB objekta
- Poruka o rezultatu izvrsenja metode koja se izvrsava na aplikacionom serveru.
- Uspesnost izvrsenja metode koja se izvrsava na aplikacionom serveru (true - uspesno, false -ne uspesno).
- Broj dokumenata EntityBeanOB objekta
- Broj stavki tekuceg dokumenta EntityBeanOB objekta

Svaki od klijenata prihvata zahtev (8) i šalje zahtev svom session bean objektu, da on (klijent) bude osvežen (9). Session bean svakog klijenta prihvata zahtev (10) i šalje nazad svom klijentu "osveženu bazu" (11). Klijent prihvata "osveženu bazu" (12) i osvežava svoju lokalnu "objektnu bazu".

SPPR1.4 Prikaz arhitekture softverskog sistema (Slika ASS) nakon projektovanja slojeva

Nakon izbora programskog jezika, operativnog sistema, sistema za upravljanje bazom pod, middleware generičkih mehanizama i aplikacionog servera moguće je dati arhitekturu softverskog sistema, koja se sastoji iz tri nivoa:

- korisnički interfejs
- aplikaciona logika
- baza podataka

i četiri sloja:

- sistemski softver
- middleware
- aplikaciono-generalni
- aplikaciono-specifični

U navedenoj arhitekturi može da se primeti da smo od Applet, EntityBeanOB i EntityBeanRB klasa aplikacionog servera, napravili za svaku od njih po jednu dopunsku klasu, koju smo podigli na aplikaciono-generalni sloj arhitekture (AppletOpsti, AEntityBeanOB i AEntityBeanRB). Ovako dobijene klase postaju ili:

a) **interfejsi**, koji definišu sistemske operacije, bez njihove realizacije (Izvedene klase treba da realizuju sistemske operacije), ili
b) **apstraktne klase**, koje definišu sistemske operacije, sa i bez realizacije. (Ukoliko nema realizacije, sistemske operacije postaju apstraktne metode, koje u izvedenim klasama treba da se realizuju. Ukoliko ima realizacije, sistemske operacije postaju konkretne metode, koje imaju neko generalno svojstvo[90].).

U našem primeru od klasa ponašanja gkObradaPredračuna, kkObradaPredračuna i ekObradaPredračuna iz analize, nastaju AppletOpsti, AEntityBeanOB i AEntityBeanRB[91].

U daljem tekstu ćemo videti da neke od ovako dobijenih klasa već postoje i da se kao takve mogu koristiti u razvoju našeg programa.

SPPR1.5:Nasleđeni sistemi

Klase AppletOpsti i AEntityBeanRB su ranije napravljene, u razvoju drugih programa, ali se kao takve mogu koristiti i u razvoju našeg programa (Slika NS).

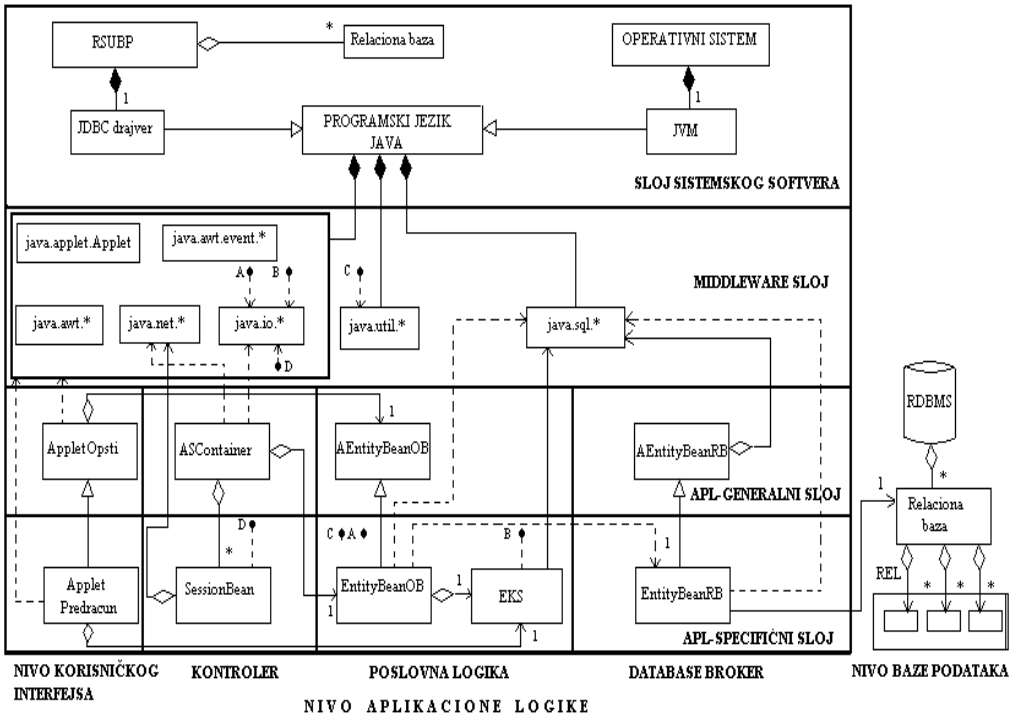
1. AEntityBeanRB je apstraktna klasa koja koja omogućava komunikaciju aplikacionog servera i relacione baze. Navedena klasa je entitetska klasa ponašanja.

```
import java.sql.*;

abstract class AEntityBeanRB
{
  static Connection con; // Konekcija prema relacionoj bazi.
  static Statement st;   // Naredba koja se izvrsava nad relacionom bazom.
```

[90] Konkretna metoda neke klase (nezavisno od toga da li je klasa apstraktna ili obična) je generalna u sledeća dva slučaja:
a) Ukoliko se kao takva (nepromenljiva) nasleđuje preko izvedenih klasa.
b) Ukoliko je njena realizacija invarijantna na različite tipove parametara koji joj se prosleđuju.

[91] Navedeno preslikavanje je jednoznačno. To znači da se sve operacije klasa ponašanja iz analize preslikavaju u odgovarajuće operacije klasa projektovanja.

```
abstract boolean otvoriBazu(String imeBaze);

boolean commitTransakcije()
   {  try{ con.commit();}
       catch(SQLException esql)
           {  System.out.println("Nije uspesno uradjen commit. traansakcije" + esql);  return false; }
      return true;
   }

 boolean rollbackTransakcije()
   {  try{ con.rollback();}
      catch(SQLException esql)
           {  System.out.println("Nije uspesno uradjen rollback. transakcije" + esql);  return false; }
      return true;
  }

    public boolean pamtiEntitet(SpecificneMetode_AEntityBeanRB smaebrb)
    { String upit;
      try{    st = con.createStatement();
              upit ="INSERT INTO " + smaebrb.vratiImeKlase()  +
                      " VALUES (" + smaebrb.vratiVrednostiAtributa() + ")";
              st.executeUpdate(upit);
         }  catch(SQLException esql)  { System.out.println("Nije uspesno zapamcen slog u bazi: " + esql);  return
                                        false;  }
      return true;
    }

  public boolean brisiEntitet(SpecificneMetode_AEntityBeanRB smaebrb)     { … }

  public boolean promeniEntitet(SpecificneMetode_AEntityBeanRB smaebrb)   { … }

. . .
}
```

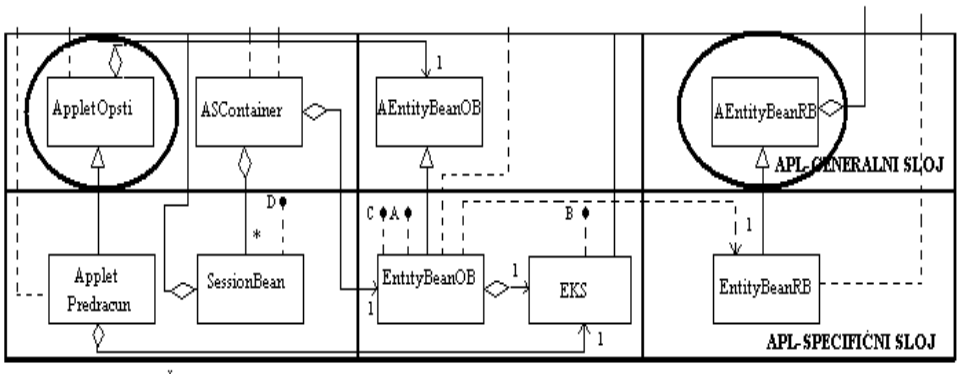


Slika ASS: Arhitektura softverskog sistema

Navedena apstraktna klasa, zajedno sa njenom izvedenom klasom, koja će da implementira apstraktne metode, predstavlja realizaciju entitetske klase ponašanja ekObradiPredračun iz analize.

Pored navedenih metoda navedenoj apstraktnoj klasi dodaćemo još dve apstraktne metode koje treba da omoguće prebacivanje slogova predracuna i roba iz relacione baze u EntityBeanOB objekat aplikacionog servera. Navedene metode su specifične metode, koje su vezane za domen problema i one nemaju generalno svojstvo.

```
// Ime:napuniEntityBeanOBPredracun(AEntityBeanOB aob);
// Odgovornost: Puni atribut Predracun EntityBeanOB objekta iz relacije
// Predracun zadate relacione baze.
// Izlaz:Ukoliko je metoda uspesno izvrsena vraca true, inace vraca false.
abstract boolean napuniPredracunIzBaze(SpecificneMetode_EntityBeanRB smebrb);

// Ime:napuniRobuIzBaze(AEntityBeanOB aob);
// Odgovornost:Puni atribut Roba EntityBeanOB objekta iz relacije
// Roba zadate relacione baze.
// Izlaz:Ukoliko je metoda uspesno izvrsena vraca true, inace vraca false.
abstract boolean napuniRobuIzBaze(SpecificneMetode_EntityBeanRB smebrb);
```



Slika NS: Nasleđeni sistemi (zaokruženi na slici)

2. ApletOpsti je apstraktna klasa koja koja omogućava komunikaciju klijenta i aplikacionog servera Navedena klasa je granična klasa ponašanja.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;

abstract public class ApletOpsti extends Applet implements  KeyListener, ActionListener,  ItemListener
{
Polje po[];        // Polja koja ce biti prikazana na formi.
Dugme dug[];       // Dugmad koja ce biti prikazana na formi.
AEntityBeanOB ob;   // Objektna baza(OB) sa kojom radi aplet(klijent). Na formi ce biti prikazani objekti OB.
Socket soketS;      // Soket koji uspostavlja vezu izmedju apleta i aplikacionog servera. Preko njega aplet prihvata kontekst
                    // aplikacionog servera i salje zahtev aplikacionom serveru da izvrsi odgovarajucu metodu.
Socket soketK;      // Soket koji uspostavlja vezu izmedju apleta i aplikacionog servera. Preko njega aplet prihvata
zahtev od                                   // aplikacionog servera da se osvezi OB na strani apleta.
int TekuciDokument = 0; // Tekuci dokument koji se vidi na ekranskoj formi.
int TekucaStavka = 0;      // Tekuca stavka koja se vidi na ekranskoj formi.
boolean BlokadaNiti = false; // Aplet prekida izvrsenje kada se aplikacionom serveru postavi zahtev da izvrsi neku od sistemskih
                             // operacija. Navedeni logicki atribut prati stanja blokiranosti apleta.
boolean InicijalizacijaApleta = false; // Prati da li je aplet uspesno inicijalizovan.
KontekstAplikacionogServera kas; // Objekat koji aplikacioni server salje apletu. On sadrzi informacije o: a) novom stanju OB i
                                 //  b) metodi koja je izvrsena na aplikacionom serveru.
NitPrihvataObjekteBaze npob; // Nit preko koje se prihvata kontekst apl. servera.
NitPrihvataInformacijeOOsvezavanju npioo; // Nit preko koje se prihvata zahtevod aplikacionog servera da se osvezi OB
                                          // na strani apleta. Takodje preko ove niti se salje zahtev apl. serveru da osvezi OB
                                          // apleta.

public void init()
{ try {      InicijalizacijaApleta = inicijalizacijaApleta();    }
  catch(Exception e) {System.out.println("Izuzetak kod inicijalizacije apleta: " + e);}
}

public boolean inicijalizacijaApleta()
{ boolean signal = false;  boolean pasaa;  boolean nkpioas;  boolean efof;
  try {   // Preduslovi
          boolean ppas= proveraIzvrsavanjaAplikacionogServera();
          if (ppas == false)
              return signal;
          System.out.println("Aplikacioni server se izvrsava");
     }
  catch(Exception e){ System.out.println("Izuzetak kod ispitivanja preduslova: " + e);  return signal;  }
```

```
  try {   // Odgovornost
          pasaa = poveziApletSaAplikacionimServerom();
         nkpioas = kreirajNitiKojePrihvatajuInformacijeOdAplikacionogServera();
         efof = kreirajEkranskuFormuiObjekatKojiCeIspunitiFormu();
      }
  catch(Exception e) { System.out.println("Izuzetak kod izvrsenja operacije: " + e);  return signal;  }

 try {  // Postuslovi
        if ((pasaa == false) || (nkpioas == false) || (efof == false))
           signal = false;
        else
          signal = true;
    }
 catch(Exception e) { System.out.println("Izuzetak kod ispitivanja postuslova: " + e);  return signal;  }
 return signal;
}

. . .
}
```

Ova apstraktna klasa sadrži atribute Polje i Dugme, za koje je data njihova definicija:

import java.applet.*;

```
class Polje
 { String nazivPolja;
    Label l;
    TextField tf;
   Choice ch;
   boolean umetak;
   int BrojUmetaka;

   Polje(String nazivPolja1,Label l1,TextField tf1, boolean umetak1, int BrojUmetaka1,Choice ch1)
     { nazivPolja = nazivPolja1; l=l1; tf=tf1;umetak=umetak1;BrojUmetaka=BrojUmetaka1;ch=ch1;}
 }


class Dugme
 { String nazivDugmeta;
   Button d;
   Dugme(String nazivDugmeta1, Button d1)
     { nazivDugmeta = nazivDugmeta1; d=d1;}

 }
```

Takođe je data klasa koja sadrži generalne metode grafičkih elemenata:

```
class  generalneMetodeGrafickihElemenata
{ static String VratiString(Polje po[],String nazivPolja)
 static void NapuniString(Polje po[],String nazivPolja, String vrednost)
}
```

SPPR1.6:Standardi

U razvoju softverskog proizvoda treba koristiti UML standard za specifikaciju, vizuelno predstavljanje i dokumentovanje elemenata softverskog sistema.

SPPR2:*Struktura softverskog sistema*

Strukturu sistema objasnićemo preko relacionog modela i klasa projektovanja strukture.

SPPR2.1: Relacioni model

Relacioni model se povezuje sa konkretnim SUBP. U ovom studijskom primeru nećemo ulaziti u detaljnu specifikaciju relacija preko izabranog SUBP.

SPPR2.2: Klase projektovanja strukture

Klase strukture iz analize su realizovane preko sledećih klasa projektovanja: Predracun, StavkePredracuna i Roba.

```
import java.sql.*;
import java.io.*;

// Domenske (entitetske) klase objektne baze:
class Predracun implements SpecificneMetode_AEntityBeanRB, ApstraktniEntitet
 {  String brojPredracuna;
    String brojRacuna;
    String datumValute;
    String poslovniPartner;
    double ukupnaProdajnaVrednost;
    StavkePredracuna spred[];
    int brojStavki;

    // Ime: Konstruktor metoda klase Predracun
    // Odgovornost: Pri kreiranju novog predracuna odvaja se prostor za
    // deset stavki predracuna, dok se atribut brojStavki postavlja na 0.
    Predracun() { spred = new StavkePredracuna[10]; brojStavki=0;}
  …
}
```

class StavkePredracuna implements SpecificneMetode_AEntityBeanRB, ApstraktniEntitet

```
{ Predracun pred;
  int rbStavke;
  String sifraRobe;
  int kolicina;
  double prodajnaCena;
  double prodajnaVrednost;
  Roba rb;

  // Ime: Konstruktor metoda klase StavkePredracuna
  // Odgovornost: Pri kreiranju novog predracuna od dobija referencu na
  // predracun koji je vec kreiran a koji se prenosi kao argument.
  StavkePredracuna(Predracun pred1){ pred = pred1;}

  …
}


class Roba implements SpecificneMetode_AEntityBeanRB, ApstraktniEntitet
{ String sifraRobe;
  String nazivRobe;

   // Ime: napuni(ResultSet rs)
   // Odgovornost: Roba se puni sa vrednoscu rs objekta.
   // Izlaz: Ukoliko se punjenje robe izvrsi uspesno metoda vraca true,
   // inace metoda vraca false.
   // Napomena: rs objekat klase ResultSet predstavlja slog relacione baze i
   // on se dobija kao rezultat izvrsenja upita nad relacionom bazom.
  boolean napuni(ResultSet rs)
   {   try { sifraRobe = rs.getString("sifraRobe");   nazivRobe = rs.getString("nazivRobe");     } catch(Exception e)
         { System.out.println("Izuzetak kod napuni robu iz ResultSet objekta " + e);      return false;      }   return true;  }
  … }
```

Svaka od navedenih klasa realizuje interfejs SpecificneMetode_AEntityBeanRB. Na taj način je moguće da se objekti navedenih klasa mogu pamtiti preko metode pamtiEntitet() klase AEntityBeanRB. Klase struktura naleđuju interfejs AptraktniEntitet, kako bi se omogućilo generičko predstavljanje klasa strukture[92]:

```
interface ApstraktniEntitet extends SerijalizovanaKlasa {}
```

Svaka klasa koja treba da se prenosi preko mreže, izmedju aplikacionog servera i klijenta, treba da podržava mehanizam serijalizacije. U tom smislu Interfejs *AptraktniEntitet* nasleđuje interfej *SerijalizovanaKlasa*.

```
import java.io.*;
interface SerijalizovanaKlasa extends Serializable{}
```

SPPR3:*Ponašanje softverskog sistema*

Ponašanje sistema objasnićemo preko realizacija SK, interfejsa i klasa projektovanja ponašanja.

[92] To će kasnije biti korišćeno kod implementacije sistemskih operacija, kada se želi da parametri istih budu generički.

SPPR3.1: Realizacije SK projektovanja

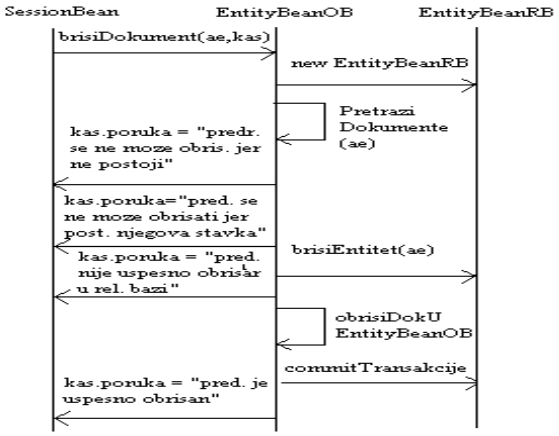
Na primeru jedne sistemske operacije, mi ćemo pokazati kako se tekstualno opisuje metoda koja treba da je realizuje. Nakon toga ćemo prikazati sekvencni dijagram za datu metodu.

SPPR3.1.1: TEKSTUALNI OPIS METODE SISTEMSKE OPERACIJE SK
Daćemo primer tekstualnog opisa metode sistemske operacije:
boolean brisiDokument(ApstraktniEntitet ae,KontekstAplikacionogServera kas)

Metoda sa povezuje sa EntityBeanRB objektom koji je zaduzen za rad sa bazom (Kod Larmana EntityBeanRB objekat bi bio database broker). Pre brisanja predračuna proverava se da li on postoji na osnovu parametra ae. (Objekat ae može da prihvati bilo koji dokument koji je izveden iz klase ApstraktniEntitet).
Ukoliko ne postoji predracun, prekida se izvršenje metode. Metoda vraca false i preko kas parametra pamti poruku: "PREDRACUN SE NE MOZE OBRISATI JER NE POSTOJI".
Ako postoji bar jedna stavka na predracunu prekida se izvršenje metode (DELETE Predracun RESTRICTED StavkePredracuna). Metoda vraca false i preko kas parametra pamti poruku: "PREDRACUN SE NE MOZE OBRISATI JER POSTOJI NJEGOVA STAVKA"
Ukoliko predracun postoji u bazi a za njega nije vezana ni jedna stavka on se brise u relacionoj bazi.
Ukoliko je predracun uspesno obrisan u relacionoj bazi, on se brise i u objektu EntityBeanOB klase, koji u operativnoj memoriji cuva sve slogove relacione baze. Transakcija se uspesno zavrsava i kas parametar pamti poruku: "PREDRACUN JE OBRISAN". Metoda vraca true.
Ukoliko predracun nije uspesno obisan u relacionoj bazi, prekida se izvrsenje metode i ona vraca false. Parametar kas pamti poruku:
"PREDRACUN NIJE USPESNO OBRISAN U RELACIONOJ BAZI"
Ukoliko u toku rada programa se desi neki nepredvidjeni izuzetak metoda ce prekinuti izvrsenje, dok ce parametar kas da zapamti poruku: "IZUZETAK KOD BRISANJA PREDRACUNA"

Na sličan način se daju tekstualni opisi metoda drugih sistemskih operacija SK.

SPPR3.1.2: SEKVENCNI DIJAGRAM METODE SISTEMSKE OPERACIJE SK
Daćemo primer sekvencnog dijagrama (Slika SDM) sistemske operacije brisiDokument():



Slika SDM: Sekvencni dijagram metode *BrisiDokument()*

Na sličan način se prave sekvencni dijagrami drugih sistemskih operacija SK.

SPPR3.1.3: KLASE PROJEKTOVANJA PONAŠANJA I INTERFEJSI
Kontrolna klasa kkObradaPredračuna iz analize je preslikana u interfejs AEntityBeanOB, sa malim modifikacijama koje su se odnosile na generičnost parametara metoda sistemskih operacija. Nakon toga interfejs AEntityBeanOB je realizovan sa klasom EntityBeanOB.

```
interface AEntityBeanOB extends SerijalizovanaKlasa
{
 // Sistemske operacije
 // Ime: pretraziDokumente(ApstraktniEntitet ae,KontekstAplikacionogServera obas);
 // Odgovornost: Pretrazuje postojece predracune na osnovu zadatog predracuna.
 // Izlaz: Ukoliko predracun vec postoji, metoda vraca true dok kontekstni objekat (kas)
 // dobija sledece vrednosti:
 // kas.uspesno = true;
 // kas.poruka = "PREDRACUN VEC POSTOJI";
 // kas.BrojDok = -indeks pronadjenog predracuna-;
 // kas.UkupnoStavki = -broj stavki predracuna koji je pronadjen-;
 // Ukoliko predracun ne postoji ili se desi neki izuzetak, metoda vraca false, dok
```

```
  // kontekstni objekat (kas) dobija sledece vrednosti:
  // kas.uspesno = false;
  // kas.poruka = "PREDRACUN NE POSTOJI";
  // kas.BrojDok = -vrednost indeksa koja je manja od 0-;
  boolean pretraziDokumente(ApstraktniEntitet ae,KontekstAplikacionogServera obas);
…
  boolean pretraziStavke(ApstraktniEntitet ae,KontekstAplikacionogServera obas);
…
  boolean brisiDokument(ApstraktniEntitet ae,KontekstAplikacionogServera obas);
   …
  boolean brisiStavkuDokumenta(ApstraktniEntitet ae,KontekstAplikacionogServera obas);
…
  boolean kreirajDokument(ApstraktniEntitet ae,KontekstAplikacionogServera obas);
…
  int vratiBrojDokumenata();
…
  int vratiBrojStavki(int TekuciDokument);
…
  String generisiNoviBrojRacuna();
…

   …
}
import java.sql.*;
import java.util.*;

class EntityBeanOB implements AEntityBeanOB, SpecificneMetode_EntityBeanRB
 { Predracun pred[];
    Roba rb[];
    int brojRoba;
    int brojPredracuna;
. . .

public boolean pretraziDokumente(ApstraktniEntitet ae,KontekstAplikacionogServera kas)
…
}
```

Takođe se vrši implementacija apstraktne klase AEntityBeanRB, koja je ranije definisana, preko klase EntityBeanRB.

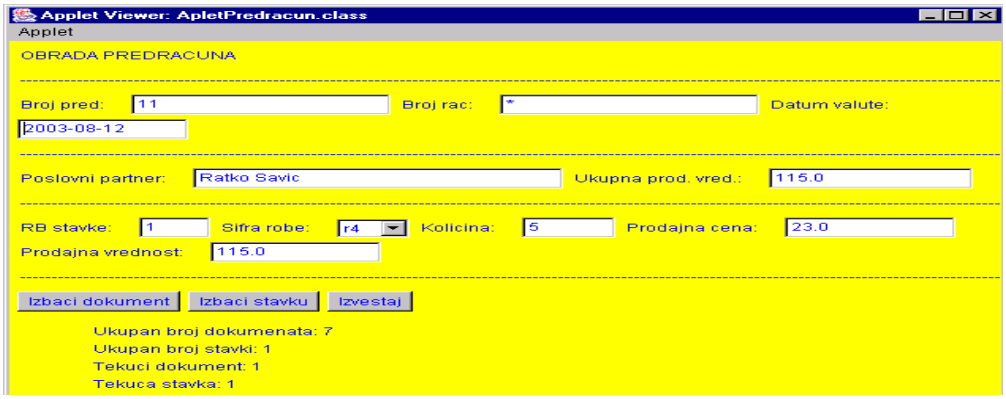
```
import java.sql.*;

class EntityBeanRB extends AEntityBeanRB
  {  public boolean otvoriBazu(String imeBaze)  {…}
     public boolean zatvoriBazu() {…}
     public boolean napuniPredracunIzBaze(SpecificneMetode_EntityBeanRB smebrb) {…}
     public boolean napuniRobuIzBaze(SpecificneMetode_EntityBeanRB smebrb) {…}
}
```

Na sličan način kao što smo uveli interfejs *SpecificneMetode_AEntityBeanRB,* možemo uvesti interfejs *SpecificneMetode_EntityBeanRB* koji mora da bude realizovan ukoliko se želi koristiti klasa EntityBeanRB.

Na kraju vršimo implementacija apstraktne klase AppletOpsti, koja je ranije definisana, preko klase AppletPredracun.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;

public class AppletPredracun extends AppletOpsti
{ Predracun pred;   int poruka;
  public boolean proveraIzvrsavanjaAplikacionogServera()
  public boolean poveziApletSaAplikacionimServerom()
  public boolean kreirajObjekatEkranskeForme()
  public void kreirajTekstPolja()
  public void napuniTekstPolja()
  public void napuniDokument()
  public boolean obradiDogadjaj(Object obj)
… }
```

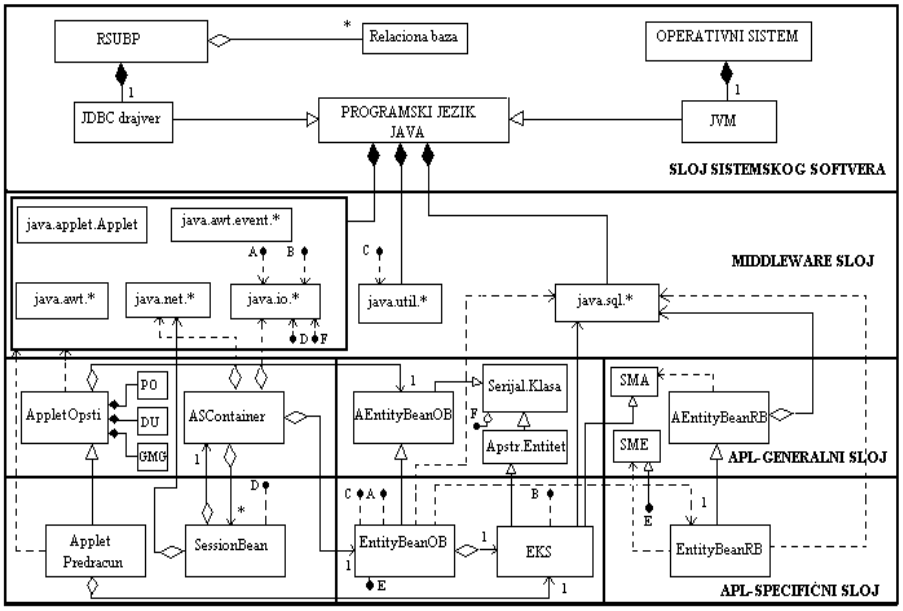
Navedena klasa treba da kao rezultat da ekransku formu preko koje će se vršiti obrada Predračuna (Slika EFP):



Slika EFP: Ekranska forma za obradu Predračuna

SPPR4: *Opis arhitekture iz perspektive modela projektovanja*

Rezultat kompletnog razvoja soft. sistema je predstavljen arhitekturom soft. sistema (Slika ASSK):



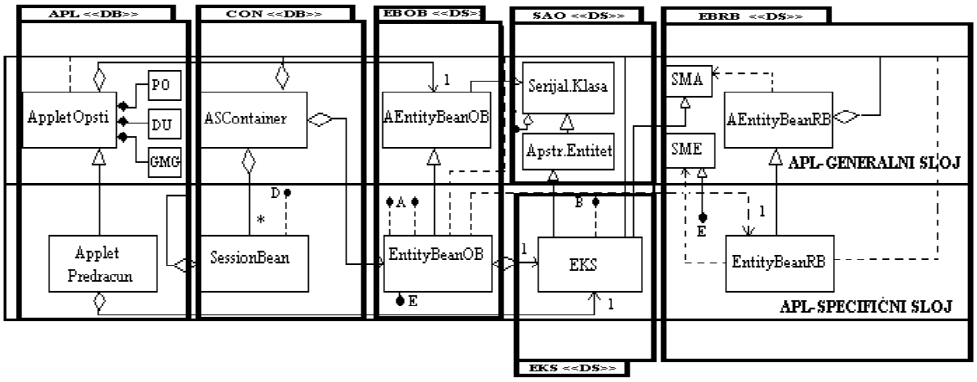
Slika ASSK: Arhitektura softverskog sistema

Arhitektura softverskog sistema je proširila arhitekturu okruženja sistema tako što je:

a) povezala klasu *AppletOpsti* sa klasama *PO* (polje), *DU* (dugme) i *GMG* (generalne grafičke metode) konkretnog programskog jezika.
b) Uvela klasu *SerijalizovanaKlasa* kako bi omogućila serijalizaciju nad objektima.
c) Razradila klasu *EKS (Entitetske klase strukture).*
d) Uvela dva interfejsa *SMA (*SpecificneMetode_AEntityBeanRB) i *SME (*SpecificneMetode_EntityBeanRB) kako bi se *EKS* i *EntityBeanOB* klase povezale sa *AEntityBeanOB* i *EntityBeanRB* klasama.

SPPR4.1: Aplikacioni podsistemi projektovanja

Iz arhitekture sistema (aplikacionih slojeva) smo identifikovali sledeće podsisteme (Slika APPR):



Slika APPR: Aplikacioni podsistemi projektovanja

1. *Podsistem **APL,** koji ima ulogu da:*
   - *Poveže klijenta sa aplikacionim serverom.*
   - *Kreira niti koje prihvataju informacije od aplikacionog servera.*
   - *Kreira ekranske forme i objekta cije ce vrednosti atributa ispuniti graficke elemente (tekst polja, kombo boksovi,...) ekranske forme*
   - *Obradi predračun preko ekranske forma i da pošalje zahtev aplikacionom serveru da izvrši neku sistemsku operaciju (kreirajDokument, brisiDokument,…).*
2. *Podsistem, **CON** koji ima ulogu da:*
   - *Puni EntityBeanOB objekat iz relacione baze.*
   - *Kreira niz koji treba da cuva session bean-ove.*
   - *Kreira soket koji ce povezati aplikacioni server sa klijentima.*
   - *Kreira objekat kontrole rada aplikacionog servera. (Klijenti pre nego sto se povezu sa aplikacionim serverom proveravaju da li se aplikacioni server uopste izvrsava).*
   - *Povezuje klijente sa aplikacionim serverom a zatim za svakog klijenta generiše session bean objekat, koji u daljem radu programa preuzima odgovornost za komunikaciju sa klijentom. Postavlja kontekst session bean objektima.*
   - *Prihvata zahtev od klijenta da izvrši sistemsku operaciju.*
   - *Preusmerava zahtev za izvršenje sistemske operacije do EntityBeanOB objekta.*
   - *Ukoliko je izvršena sistemska operacija promenila stanje baze, javlja klijentima da osveže svoju lokalnu "bazu podataka".*

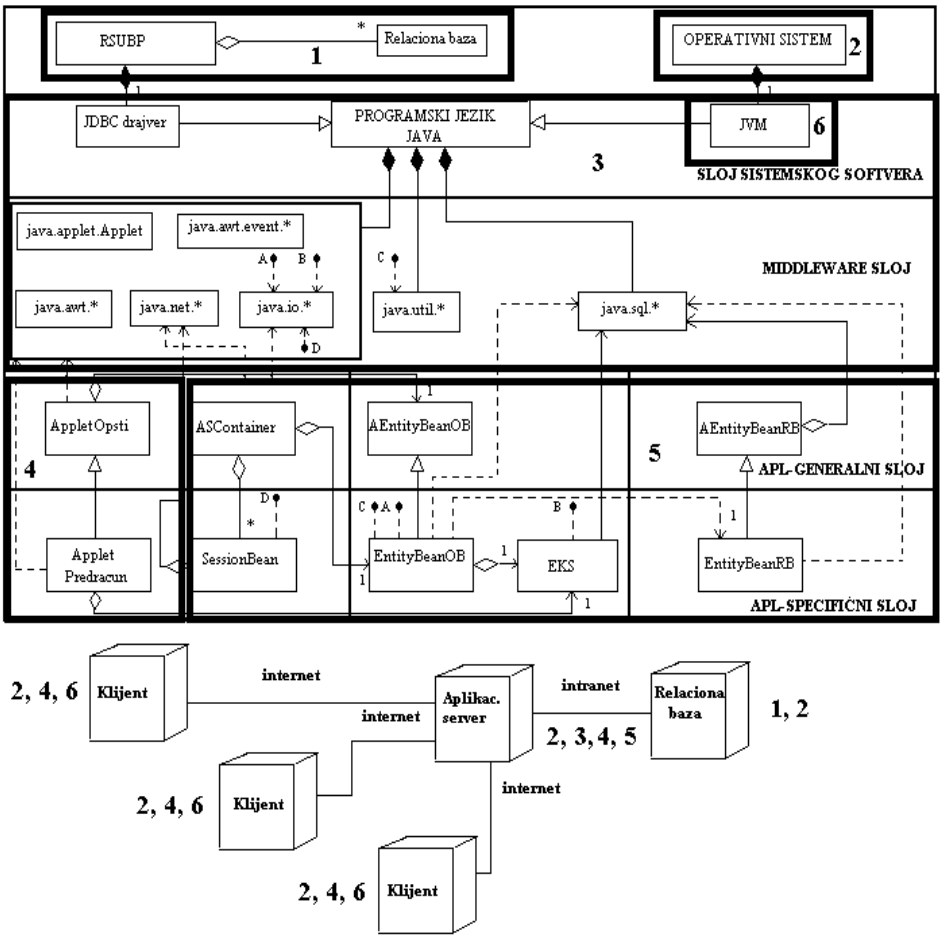
*3. Podsistem **EBOB,** koji ima ulogu da:*
- *Realizuje sistemske operacije koje su definisane u kontrolnoj klasi KKObradaPredračuna iz analize.*
- *Napuni svoj kontekst sa slogovima iz relacija koje se nalaze u relacionoj bazi.*

*4. Podsistem **SAO,** je zadužen da obezbedi serijalizaciju objekata i generalnost entitetskih klasa trukture.*
*5. Podsistem **EKS,** sadrži sve entitetske klase strukture (Predračun, StavkePredračuna i Roba). Svaka od navedenih klasa sadrži svoju strukturu i interne operacije. Navedene klase su između sebe povezane preko objekata. Naglašavamo da je ovde data definicija klasa a ne struktura podataka koje će da čuvaju objekte tih klasa (EntityBeanOB objekat sadrži strukture podataka koje čuvaju objekte).*
*6. Podsistem **EBRB,** ima ulogu da omogući komunikaciju između EntityBeanOB objekta i relacione baze.*

SPPR5: *Model raspoređivanja*

Kada se opisuje model raspoređivanja (Slika MR) tada se posmatra kompletna arhitektura softevrskog sistema, ne samo aplikacioni slojevi. Tu se uočavaju arhitekturni podsistemi, koje smo označili sa rednim brojevima od 1 do 6. Navedeni podsistemi se dodeljuju do čvorova modela raspoređivanja.

Treba primetiti, da arhitekturni podsistemi za razliku od aplikacionih podsistema projektovanja obuhvataju različit skup klasa.



Slika MR: Model rapoređivanja

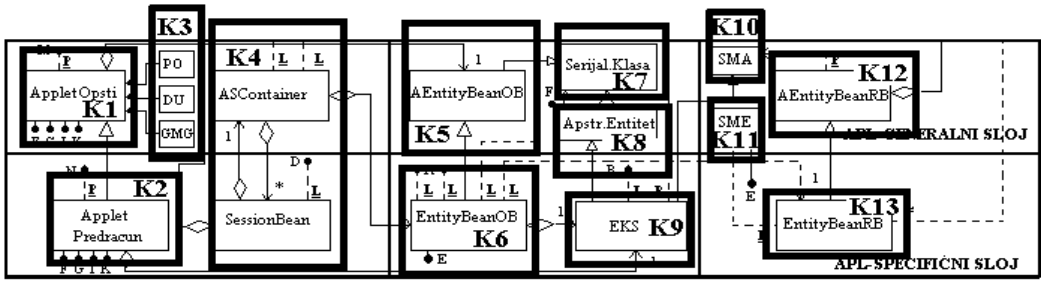
Navešćemo relacije koje postoje između klasa arhitekture softverskog sistema, koje ćemo grupisati po slojevima (sistemski softver, middleware i aplikacioni sloj, koji zajedno čine aplikaciono generalni i aplikaciono specifični sloj) i nivoima (korisnički interfejs, poslovna logika i baza podataka).

4.5.5: **SPIM – S**TUDIJSKI **P**RIMER – **IM**PLEMENTACIJA

U studijskom primeru prikazaćemo komponente koje smo dobili iz arhitekture softverskog sistema. Zatim ćemo prikazati redosled po kome se izvršavaju komponente i njihovu međusobnu zavisnost.

SPIM5.1: *Komponente*

Komponente možemo dobiti iz klasa i interfejsa ili podsistema aplikacionih slojeva. U našem primeru većina klasa projektovanja ponašanja je jednoznačno preslikana u komponente (Slika KA).



Slika KA: Komponente aplikacije

Dajemo spcifikaciju svake od navedenih komponenti:

- K1 – naziv komponente: *AppletOpsti.java* ;
  Klase koje se čuvaju u komponenti: *AppletOpsti*
- K2 – naziv komponente: *AppletPredracun.java* ;
  Klase koje se čuvaju u komponenti: *AppletPredracun*
- K3 – naziv komponente:*GeneralniGrafickiElementi.java* ;
  Klase koje se čuvaju u komponenti: *Polje, Dugme i generalneMetodeGrafickihElemenata*
- K4 – naziv komponente: *ASContainer.java* ;
  Klase koje se čuvaju u komponenti: *ASContainer i SessionBean*
- K5 – naziv komponente: *AEntityBeanOB.java* ;
  Klase koje se čuvaju u komponenti: *AEntityBeanOB*
- K6 – naziv komponente: *EntityBeanOB*;
  Klase koje se čuvaju u komponenti: *EntityBeanOB*
- K7 – naziv komponente: *Serijalizacija.java*;
  Klase koje se čuvaju u komponenti: *SerijalizovanaKlasa*
- K8 – naziv komponente: *ApstraktniEntitet.java*;
  Klase koje se čuvaju u komponenti: *ApstraktniEntitet*
- K9 – naziv komponente: *EntitetskeKlaseS.java* ;
  Klase koje se čuvaju u komponenti: *Predracun, StavkePredracuna i Roba*
- K10 – naziv komponente: *SM_AEBRB.java*;
  Klase koje se čuvaju u komponenti: *SpecificneMetode_AEntityBeanRB*
- K11 – naziv komponente: *SM_EBRB.java*;
  Klase koje se čuvaju u komponenti: *SpecificneMetode_EntityBeanRB*
- K12 – naziv komponente: *AEntityBeanRB.java*;
  Klase koje se čuvaju u komponenti: *AEntityBeanRB*
- K13 – naziv komponente: *EntityBeanRB.java*;
  Klase koje se čuvaju u komponenti: *EntityBeanRB*

SPIM5.2: *Redosled kompajliranja komponenti*

Komponente koje se nalaze u aplikaciono generalnom sloju se prve kompajliraju[93]. Nakon toga se kompajliraju komponente u aplikaciono specifičnog sloja.
Redosled kompajliranja možemo da predstavimo preko datoteke *redosledKompajliranja.bat* :

[93] U skripti [Cir 1] prof. Ciric V. govori o redosledu implementacije i testiranja klasa, u tom smislu on kaže:
Graf klasa ime horizontalne i vertikalne nivoe. Redosled implementacije i testiranja klasa unutar horizontalnog nivoa je proizvoljan jer su klase medjusobno nezavisne. Medjutim implementacija i testiranje klasa po vertikalnim nivoima mora se vršiti po redosledu tih nivoa. Graf klasa može biti od koristi:

1. *U timskom radu na projektu, jer olakšava rukovodiocu tima upravljanje projektom.*
2. *Kod konkurentnog izvršenja pojedinih procesa, za koje se na sličan način može konstruisati graf procesa na osnovu slučaja korišćenja, dijagrama klasa i postojećih interaktivnih dijagrama.*

javac Serijalizacija.java
javac ApstraktniEntitet.java
javac SM_AEBRB.java
javac SM_EBRB.java
javac AEntityBeanOB.java
javac AEntityBeanRB.java
javac GeneralniGrafickiElementi.java
javac ApletOpsti.java
javac EntitetskeKlaseS.java
javac EntityBeanRB.java
javac EntityBeanOB.java
javac ASContainer.java
javac AppletPredracun.java

Na kraju se izvršavaju komeponente *ASContainer.java* na serverskoj strani i *AppletPredracun.java* na klijentskoj strani.

4.5.6: **SPT – S**TUDIJSKI **P**RIMER – **T**ESTIRANJE

Naš studijski primer testirali smo u 3 koraka:

1. Testirali smo svaku od komponenti (KOM1-KOM13) kao nezavisne funkcionalne jedinice.
2. Integrisali smo komponente u aplikacioni server, testirajući pri tom tri osnovna scenarija u radu aplikacionog servera:
   - Inicijalizaciju aplikacionog servera.
   - Povezivanje klijenta sa aplikacionim serverom.
   - Prihvatanje zahteva od klijenta za izvršenje neke od sistemskih operacija.

3. Na kraju smo testirali kako se aplikacioni server ponaša u okruženju više korisnika.

5. Literatura

[AC1] Alexander Cristopher*: Notes on the Synthesis of Form.;* Cambridge, MA: Harvard University Press, 1974.

[AC2] Alexander Cristopher i drugi*: A Pattern Language;* New York: Oxford University Press,1977.

[AC3] Alexander Cristopher*: The Timeless Way of Building;* New York: Oxford University Press ,1979.

[Budgen] Budgen David – *Software design,* Pearson, Addison Wesley, Edinburg Gate, England, 2003.

[Gartner95] Schulte, R., 1995: *Three – Tier Computing Architectures and Beyond,* Published Report Note R-401-134, Gartner Group.

[GOF] Erich Gamma, Richard Helm, Ralph Johnson, JohnVlissides: Design patterns*, Addison : Wesley, 18th Printing, September 1999.*

[DeBr] DeBruler, ...: A Generative Pattern Language for Distributed Processing*, Reading, MA: Addison-Wesley, 1995.*

[EA1] Endres Albert, Rombach Dieter: *A Handbook of Software and Systems Engineering,* Pearson – Addison Wesley, Edinburg Gate, England, 2003.

[Java1] Cay S. Horstmann, Gary Cornell: *core JAVA, Volume I – Fundamentals*, Sun Microsystems Press, 2000, California, USA.

[Java2] Cay S. Horstmann, Gary Cornell: *core JAVA 2, Volume II – Advanced Features*, Sun Microsystems Press , 2000, California, USA.

[Java3] Clifford J. Berg: *advanced JAVA 2, Development for Enterprise Application,* Sun Microsystems Press, 2000, New Jersey, USA.

[Java4] Danny Ayers, Hans Bergsten,…: *Profesional Java Server Programming,* Wrox Press, 1999, Birmingham, UK.

[Java5] Herbert Schildt: *kompletan priručnik, JAVA 2, prevod četvrtog izdanja* (preveli: Dejan Smiljanić, Milorad Popović), Mikro knjiga, 2001, Beograd.

[Java6] Vlajić Siniša, Ćirić Vidojko, Savić Dušan: *Projektovanje programa (Praktikum – Programski jezik Java),* sopstveno izdanje, 2003, Beograd.

[JPRS] Ivar Jacobson, Grady Booch, James Rumbaugh: *The Unified Software Development Process,* Rational Software Corporation, Addison-Wesley, 1999.

[Laz3] Lazarević Branko, Siniša Nešković: *Metodologija modeliranja poslovnih procesa,* Simpozijum YU-INFO, 1998.

[Laz4] Lazarević Branko, Siniša Nešković: *Modelovanje poslovnih procesa kao osnova menadžmenta i razvoja informacionih sistema,* Simpozijum InfoTeh, Vrnjačka Banja, 1998.

[Laz5] Lazarević Branko, Siniša Nešković: *Sistemsko – teorijska kritika objektno – orijentisanih pristupa razvoju softvera,* Simpozijum InfoTeh, Vrnjačka Banja, 1997.

[Larman] Craig Larman: *Applying UML and Patterns,* Second edition, Prentice Hall, ,New Jersy, 1998

[PB] Petrović Bratislav – *Teorija sistema,* FON*,* Beograd*, 1998.*

[Pree] W. Pree: *Design Patterns for Object-Oriented Software Development*, Wokingham, England, Addison-Wesley, 1995.

[Refact] Martin Flower: *Refactoring – Improving the Design of Existing Code,* Addison-Wesley, USA, 1999.

[RKo] R. Koenig: *Patterns and Antipatterns.* Journal of Object-Oriented Programming, 1995.

[Rum] J. Rumbaugh i drugi : *Object-Oriented Modeling and Design,* Prentice Hall, Englewood Cliffs, NJ, 1991.

[Sv12] Siniša Vlajić, Stojan Dabić, Vidojko Ćirić: *Informacioni sistem za upravljanje portfeljom hartija od vrednosti,* Zbornik radova SymOrg '98: VI međunarodni simpozijum "Okruženje, menadžment, konkurentnost", Strane 457-461, 3-5. 06. 1998., Zlatibor, SR Jugoslavija.

[Sv13] Siniša Vlajić, Danilo Pešić, Bojan Ilić, Ćirić Vidojko: *Matematički model za analizu troškova nabavke i ispitivanja materijala u skladu sa smernicama serije standarda ISO 9000,* Zbornik radova SYM-OP-IS '99: Jugoslovenski simpozijum o operacionim istraživanja, Beograd, SR Jugoslavija.

[Sv17] Siniša Vlajić, Vidojko Ćirić: *Razvoj funkcije preslikavanja podataka između komercijalne i finansijske službe,* Maj 2002, Zbornik radova SymOrg 2002, Zlatibor, SR Jugoslavija.

[Sv21] S. Vlajić, V. Ćirić: *Model događaja u Microsoft-ovom Access-u 2.0*, SimInfo 95, Zlatibor.

[Sv24] S. Vlajić, *Formalizacija jedinstvenog procesa razvoja softvera pomoću uzora,* Doktorska disertacija, FON, Beograd, 2003.

[TC] Tom Cargill : *C++ Programming Style,* Addison-Wesley, Reading, MA, 1992.

[TK78] Tsichiritzis, D., and Klug: *The ANSI/X3/SPARC DBMS framework,* Report of the study group on database management systems. Information Systems, 3 1978.

[UML] Rational Software Corporation, *Unified Modeling Language (UML)*, www.rational.com

[Cir1] Vidojko Ćirić, S. Černe, M. Tomić- *Principi programiranja sa Zbirkom I deo*, sopstveno izdanje, 1986, Beograd.

[Cir2] Vidojko Ćirić, S. Černe, M. Tomić- *Principi programiranja sa Zbirkom II deo*, sopstveno izdanje, 1986, Beograd.

[Cir4] Vidojko Ćirić, Siniša Vlajić: *Metodologija programiranja i programski jezik Java,* skripta, poslediplomske specijalističke studije, Beograd, 2002.

[Cook] Cook W., Palsberg J.: *A Denotational Semantics of Inheritance and its Correctness,* IN OOPSLA '89 Conference Proceedings, SIGPLAN Notices 24, 1989, New York.

[Cop1] James O. Coplien: *Software Patterns,* Bell Labs, The Hillside Group, 2000.

[Cop2] James O. Coplien: *A Development Process Generative Pattern Language. Pattern Language of Program Design*, Reading MA: Addison-Wesley, 1995.

[Cop3] J. Coplien: *Advanced C++ Programming Styles and Idioms,* Reading, MA: Addison-Wesley, 1992.

[Ullman] Ullman D. Jeffrey, Widom Jennifer, *A First Course in Database Systems,* Prentice Hall, New Jersey, USA, 1997.

