

Univerzitet u Beogradu  
Fakultet organizacionih nauka  
Laboratorija za softversko inženjerstvo

Seminarski rad iz predmeta

## Softverski proces

Tema: Razvoj igre „Iks oks“ primenom Larmanove metode

Profesor:

dr Siniša Vlajić

Student:

Luka Orlović 2018/3725

Beograd, 2021.

## Sadržaj

1. Korisnički zahtevi .....	3
1.1. Verbalni opis .....	3
1.2. Slučajevi korišćenja.....	4
1.2.1. SK1: Slučaj korišćenja – Prijavljivanje korisnika .....	5
1.2.2. SK2: Slučaj korišćenja – Registrovanje korisnika .....	6
1.2.3. SK3: Slučaj korišćenja – Započinjanje nove igre .....	7
1.2.4. SK4: Slučaj korišćenja – Čuvanje rezultata igre .....	8
1.2.5. SK5: Slučaj korišćenja – Prikazivanje rang liste .....	9
2. Analiza .....	10
2.1. Ponašanje softverskog sistema – Sistemski dijagram sekvenci.....	10
2.1.1. Dijagram slučaja korišćenja 1 – Prijavljivanje korisnika .....	10
2.1.2. Dijagram slučaja korišćenja 2 - Registrovanje korisnika .....	11
2.1.3. Dijagram slučaja korišćenja 3 – Započinjanje nove igre .....	12
2.1.4. Dijagram slučaja korišćenja 4 – Čuvanje rezultata igre .....	13
2.1.5. Dijagram slučaja korišćenja 5 – Prikazivanje rang liste .....	14
2.2. Ponašanje softverskog sistema – Definisanje ugovora o sistemskim operacijama .....	16
2.3. Struktura softverskog sistema – Konceptualni model .....	18
2.4. Struktura softverskog sistema – Relacioni model .....	18
3. Projektovanje .....	20
3.1. Arhitektura softverskog sistema .....	20
3.2. Projektovanje korničkog interfejsa.....	21
3.2.1. SK1: Slučaj korišćenja – Prijavljivanje korisnika .....	22
3.2.2. SK2: Slučaj korišćenja – Registrovanje korisnika .....	24
3.2.3. SK3: Slučaj korišćenja – Započinjanje nove igre .....	26
3.2.4. SK4: Slučaj korišćenja – Čuvanje rezultata igre .....	28
3.2.5. SK5: Slučaj korišćenja – Prikazivanje rang liste .....	30
3.3. Komunikacija server – klijent.....	32
3.4. Kontroler aplikacione logike .....	35
3.5. Projektovanje strukture softverskog sistema – Domenske klase.....	37
3.6. Projektovanje poslovne logike.....	44
3.7. Projektovanje brokera baze podataka.....	48

3.8. Projektovanje skladišta podataka.....	50
4. Implementacija.....	51
4.1. Mehanizam refleksije .....	52
4.2. <i>Generics</i> mehanizam .....	53
5. Testiranje .....	54
6. Zaključak.....	54
7. Principi, metode i strategije projektovanja softverskog sistema .....	55
7.1. Principi projektovanja softverskog sistema .....	55
7.1.1. Apstrakcija .....	55
7.2. Strategije projektovanja.....	69
7.3. Metode projektovanja.....	72
7.4. Principi objektno orijentisanog projektovanja .....	73
8. Primena paterna u projektovanju .....	80
8.1. Uvod u paterne.....	80
8.2. Opšti oblik GOF paterna projektovanja .....	80
9. Literatura.....	91

# **1. Korisnički zahtevi**

## **1.1. Verbalni opis**

Potrebno je razviti softverski sistem za igranje igre „Iks oks“. Sistem bi trebalo da omogući igraču (korisniku sistema) da, pre svega, može da se uloguje svojim korisničkim imenom i šifrom. Ukoliko korisnik nema kreiran profil, sistem bi trebalo da mu omogući opciju da se registruje korisničkim imenom, šifrom i imejl adresom.

Nakon registracije/ulogovanja na sistem, korisnik može da izabere da započne novu igru. Nova igra se započinje izabirom te opcije u glavnom meniju.

Sistem mora biti u stanju da vodi evidenciju o rezultatima igara datog korisnika, ali i o prikazivanju rang liste svih korisnika.

## 1.2. Slučajevi korišćenja

U ovom slučaju su korisnički zahtevi prikazani i na slici:

1. Prijavljivanje korisnika
2. Registrovanje korisnika
3. Započinjanje nove igre
4. Čuvanje rezultata igre
5. Prikazivanje rang liste



Slika 1 – Dijagram slučaja korišćenja

### 1.2.1. SK1: Slučaj korišćenja – Prijavljivanje korisnika

#### Naziv SK

Prijavljivanje korisnika

#### Aktori SK

Korisnik

#### Učesnici SK

Korisnik i sistem (program)

**Preduslov:** Sistem je uključen i prikazana je forma za prijavljivanje korisnika.

#### Osnovni scenario SK

1. **Korisnik** unosi svoje podatke za prijavljivanje. (APUSO)
2. **Korisnik** kotroliše da li je korektno uneo svoje podatke. (ANSO)
3. **Korisnik** poziva **sistem** da izvrši prijavljivanje datog korisnika. (APSO)
4. **Sistem** vrši prijavljivanje datog korisnika. (SO)
5. **Sistem** prikazuje **korisniku** poruku: "Uspešno logovanje" i omogućava pristup sistemu. (IA)

#### Alternativna scenarija

5.1. Ukoliko **sistem** ne može da pronađe korisnika sa datim podacima, sistem prikazuje **korisniku** poruku "Neuspešno logovanje". (IA)

### 1.2.2. SK2: Slučaj korišćenja – Registrovanje korisnika

#### Naziv SK

Registrovanje korisnika

#### Aktori SK

Korisnik

#### Učesnici SK

Korisnik i sistem (program)

**Preduslov:** Sistem je uključen i prikazana je forma za registrovanje korisnika.

#### Osnovni scenario SK

1. **Korisnik** unosi svoje podatke za registrovanje. (APUSO)
2. **Korisnik** kotroliše da li je korektno uneo svoje podatke. (ANSO)
3. **Korisnik** poziva **sistem** da izvrši registrovanje datog korisnika. (APSO)
4. **Sistem** vrši registrovanje datog korisnika. (SO)
5. **Sistem** prikazuje **korisniku** poruku: "Uspešno registrovanje" i omogućava pristup sistemu. (IA)

#### Alternativna scenarija

5.1. Ukoliko **sistem** ne može da registruje korisnika sa datim podacima, sistem prikazuje **korisniku** poruku "Neuspešno registrovanje". (IA)

### 1.2.3. SK3: Slučaj korišćenja – Započinjanje nove igre

#### Naziv SK

Započinjanje nove igre

#### Aktori SK

Korisnik

#### Učesnici SK

Korisnik i sistem (program)

**Preduslov:** Sistem je uključen, korisnik je ulogovan svojim korisničkim imenom i šifrom i prikazana je glavna forma.

#### Osnovni scenario SK

1. **Korisnik** poziva **sistem** da započne novu igru. (APSO)
2. **Sistem** započinje novu igru. (SO)
3. **Sistem** prikazuje **korisniku** poruku: “ Igra startovana!”. (IA)

#### Alternativna scenarija

3.1. Ukoliko **sistem** ne može da startuje igru, sistem prikazuje **korisniku** poruku “Igra ne može biti startovana”. (IA)



#### 1.2.4. SK4: Slučaj korišćenja – Čuvanje rezultata igre

##### Naziv SK

Čuvanje rezultata igre

##### Aktori SK

Korisnik

##### Učesnici SK

Korisnik i sistem (program)

**Preduslov:** **Sistem** je uključen, korisnik je ulogovan svojim korisničkim imenom i šifrom i igra je uspešno započeta.

##### Osnovni scenario SK

1. **Korisnik** poziva **sistem** da sačuva rezultat igre. (APSO)
2. **Sistem** čuva novi rezultat. (SO)
3. **Sistem** prikazuje **korisniku** poruku: “ Igra je sačuvana!”. (IA)

##### Alternativna scenarija

3.1. Ukoliko **sistem** ne može da sačuva rezultat, sistem prikazuje **korisniku** poruku “Igra ne može biti sačuvana”. (IA)

### 1.2.5. SK5: Slučaj korišćenja – Prikazivanje rang liste

#### Naziv SK

Prikazivanje rang liste

#### Aktori SK

**Korisnik**

#### Učesnici SK

**Korisnik i sistem** (program)

**Preduslov:** **Sistem** je uključen, korisnik je ulogovan svojim korisničkim imenom i šifrom i igra je uspešno započeta.

#### Osnovni scenario SK

1. **Korisnik** poziva **sistem** da prikaže rang listu. (APSO)
2. **Sistem** vrši nalaženje i kreiranje rang liste. (SO)
3. **Sistem** prikazuje **korisniku** rang listu. (IA)

#### Alternativna scenarija

3.1. Ukoliko **sistem** ne može da prikaže rang listu, sistem prikazuje **korisniku** poruku "Rang lista ne može biti prikazana". (IA)

## 2. Analiza

### 2.1. Ponašanje softverskog sistema – Sistemski dijagram sekvenci

#### 2.1.1. Dijagram slučaja korišćenja 1 – Prijavljivanje korisnika

##### Osnovni scenario SK

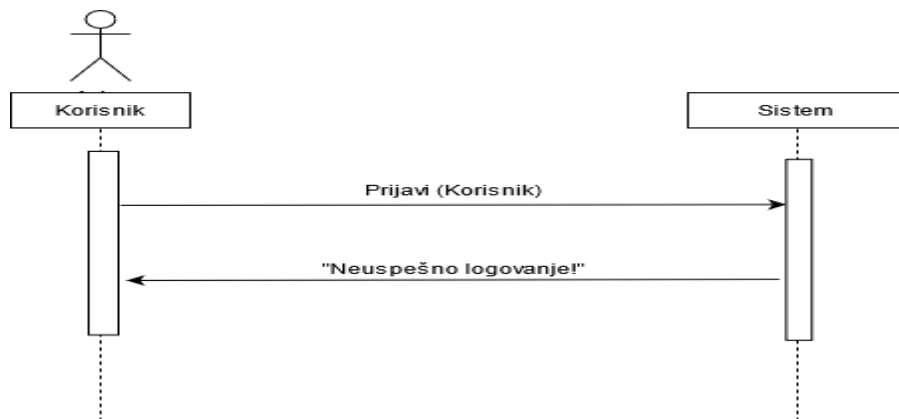
1. **Korisnik** poziva **sistem** da izvrši prijavljivanje datog korisnika. (APSO)
2. **Sistem** prikazuje **korisniku** poruku: "Uspešno logovanje" i omogućava pristup sistemu. (IA)



Slika 2 – Osnovni scenario SK1

##### Alternativna scenarija

- 2.1. Ukoliko **sistem** ne može da pronađe korisnika sa datim podacima, sistem prikazuje **korisniku** poruku "Neuspešno logovanje". (IA)



Slika 3 – Prvi alternativni scenario SK1

Sa navedenih sekvencnih dijagrama uočavaju se 3 sistemske operacije koje treba projektovati:

- 1 .signal **Prijavi**(*Korisnik*)

### 2.1.2. Dijagram slučaja korišćenja 2 - Registrovanje korisnika

#### Osnovni scenario SK

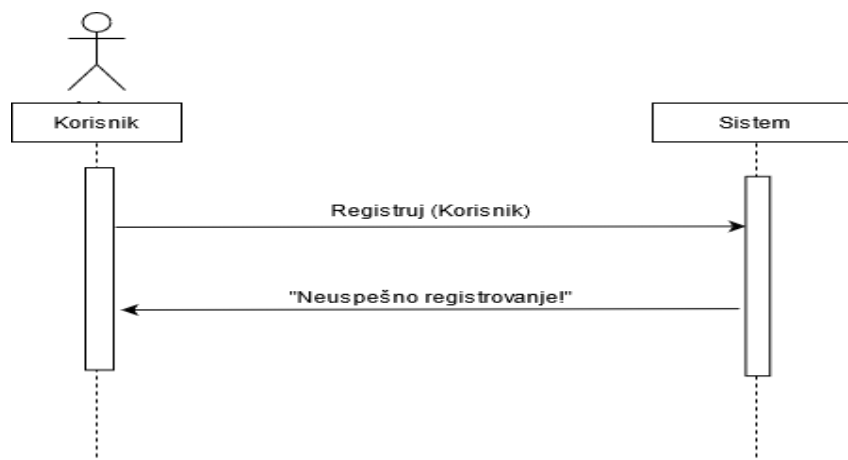
1. **Korisnik** poziva **sistem** da izvrši registrovanje datog korisnika. (APSO)
2. **Sistem** prikazuje **korisniku** poruku: "Uspešno registrovanje" i omogućava pristup sistemu. (IA)



Slika 4 – Osnovni scenario SK2

#### Alternativna scenarija

- 2.1. Ukoliko **sistem** ne može da registruje korisnika sa datim podacima, sistem prikazuje **korisniku** poruku "Neuspešno registrovanje". (IA)



Slika 5 – Prvi alternativni scenario SK2

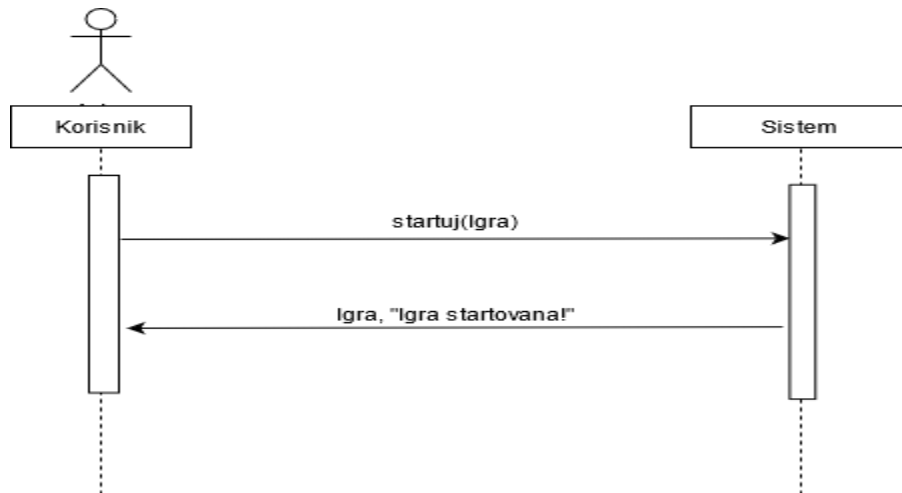
Sa navedenih sekvencnih dijagrama uočavaju se 3 sistemske operacije koje treba projektovati:

- 1 .signal **Registruj**(*Korisnik*)

### 2.1.3. Dijagram slučaja korišćenja 3 – Započinjanje nove igre

#### Osnovni scenario SK

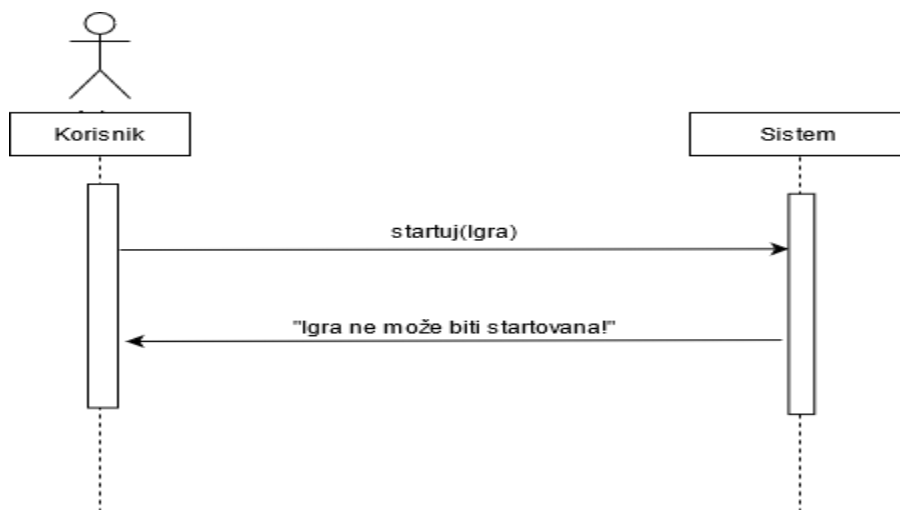
1. **Korisnik** poziva **sistem** da započne novu igru. (APSO)
2. **Sistem** prikazuje **korisniku** poruku: " Igra startovana!". (IA)



Slika 6 – Osnovni scenario SK3

#### Alternativna scenarija

- 2.1. Ukoliko **sistem** ne može da startuje igru, sistem prikazuje **korisniku** poruku "Igra ne može biti startovana". (IA)



Slika 7 – Prvi alternativni scenario SK3

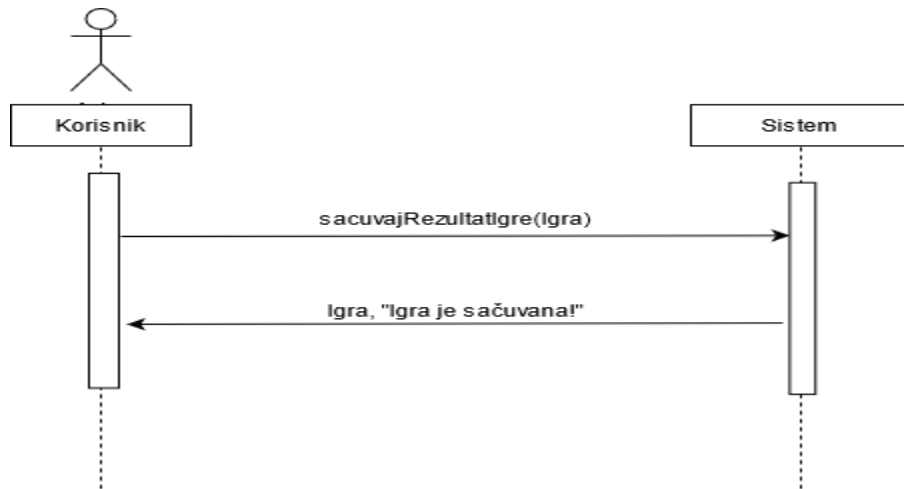
Sa navedenih sekvencnih dijagrama uočavaju se 3 sistemske operacije koje treba projektovati:

- 1 .signal **startuj(Igra)**

#### 2.1.4. Dijagram slučaja korišćenja 4 – Čuvanje rezultata igre

##### Osnovni scenario SK

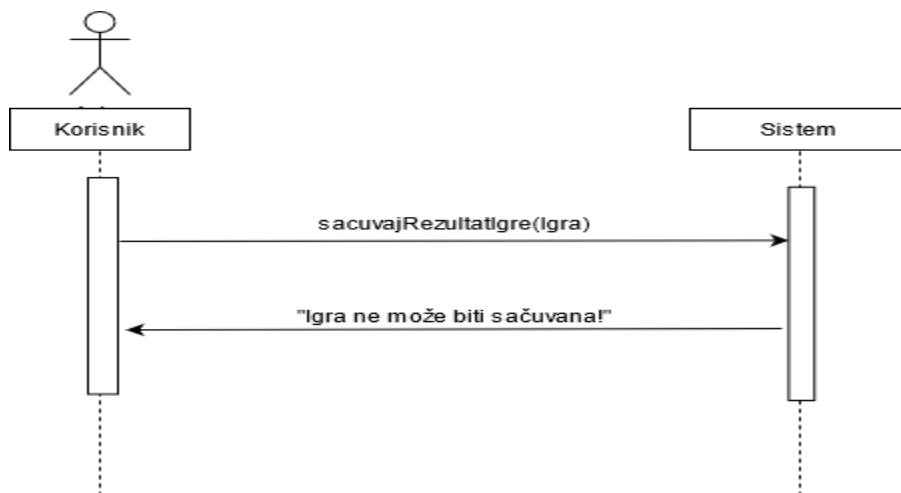
1. **Korisnik** poziva **sistem** da sačuva rezultat igre. (APSO)
2. **Sistem** prikazuje **korisniku** poruku: " Igra je sačuvana!". (IA)



Slika 8 – Osnovni scenario SK4

##### Alternativna scenarija

- 2.1. Ukoliko **sistem** ne može da sačuva rezultat, sistem prikazuje **korisniku** poruku "Igra ne može biti sačuvana". (IA)



Slika 9 – Prvi alternativni scenatio SK4

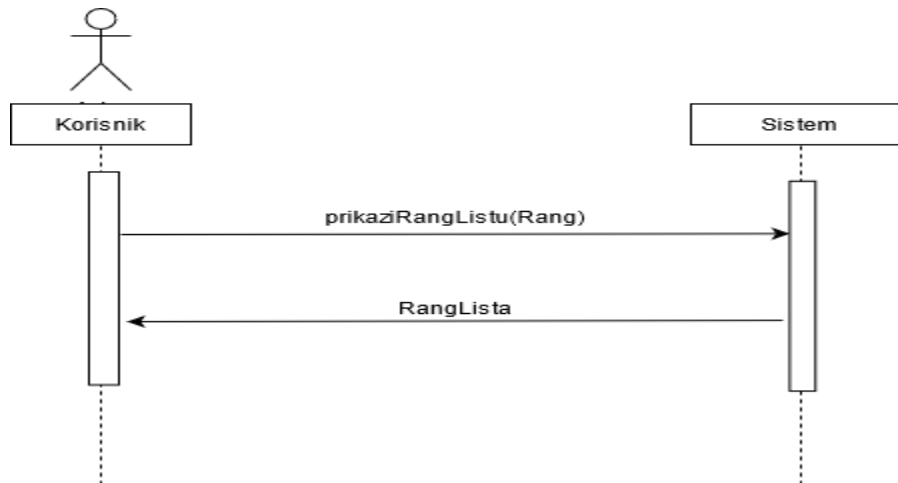
Sa navedenih sekvencnih dijagrama uočavaju se 3 sistemske operacije koje treba projektovati:

- 1 .signal **sacuvajIgru(Igra)**

### 2.1.5. Dijagram slučaja korišćenja 5 – Prikazivanje rang liste

#### Osnovni scenario SK

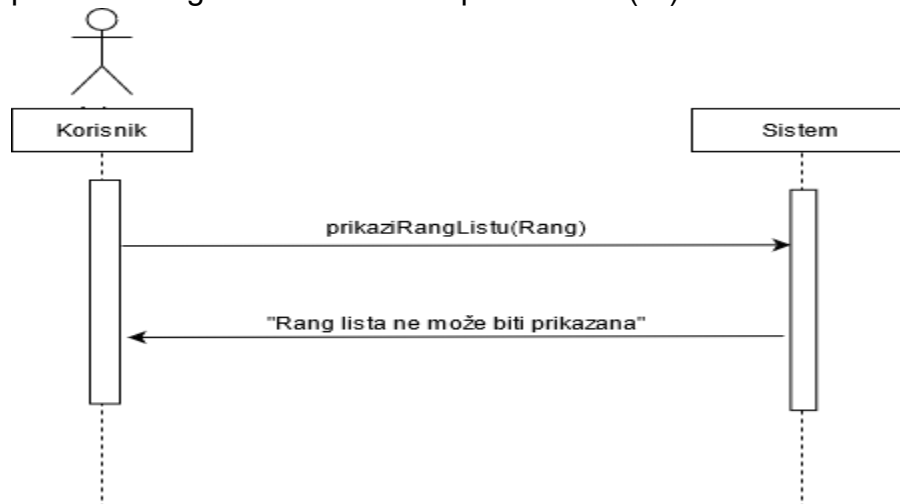
1. **Korisnik** poziva **sistem** da prikaže rang listu. (APSO)
2. **Sistem** prikazuje **korisniku** rang listu. (IA)



Slika 10 – Osnovni scenario SK5

#### Alternativna scenarija

- 2.1. Ukoliko **sistem** ne može da prikaže rang listu, sistem prikazuje **korisniku** poruku "Rang lista ne može biti prikazana". (IA)



Slika 11 – Prvi alternativni scenario SK5

Sa navedenih sekvenčnih dijagrama uočavaju se 3 sistemske operacije koje treba projektovati:

- 1 .signal **prikaziRangListu(Rang)**

Nakon analize scenarija slučajeva korišćenja uočeno je 5 sistmskih operacija koje bi trebalo projektovati:

1. signal **Prijavi**(*Korisnik*)
2. signal **Registruj**(*Korisnik*)
3. signal **startuj**(*Igra*)
4. signal **sacuvajIgru**(*Igra*)
5. signal **prikaziRangListu**(*Rang*)



## **2.2. Ponašanje softverskog sistema – Definisanje ugovora o sistemskim operacijama**

### **1. Ugovor UG1: Prijavi**

**Operacija:** Prijavi(Korisnik): signal

**Veza za SK:** SK1

**Preduslovi:** /

**Postuslovi:** /

### **2. Ugovor UG2: Registruj**

**Operacija:** Registruj(Korisnik): signal

**Veza za SK:** SK2

**Preduslovi:** Vrednosna i strukturna ograničenja nad objektom Korisnik moraju biti zadovoljena

**Postuslovi:** /

### **3. Ugovor UG3: Startuj**

**Operacija:** Startuj(Igra): signal

**Veza za SK:** SK3

**Preduslovi:** Vrednosna i strukturna ograničenja nad objektom Igra moraju biti zadovoljena

**Postuslovi:** Igra je započeta

### **4. Ugovor UG4: SacuvajIgru**

**Operacija:** SacuvajIgru(Igra): signal

**Veza za SK:** SK4

**Preduslovi:** Vrednosna i strukturna ograničenja nad objektom Igra moraju biti zadovoljena

**Postuslovi:** Igra je sačuvana

## **5. Ugovor UG5: PrikaziRangListu**

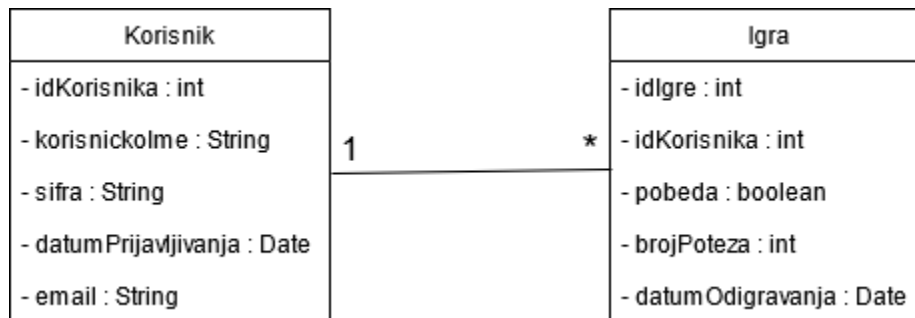
**Operacija:** PrikaziRangListu(Rang): signal

**Veza za SK:** SK5

**Preduslovi:** /

**Postuslovi:** /

### 2.3. Struktura softverskog sistema – Konceptualni model



### 2.4. Struktura softverskog sistema – Relacioni model

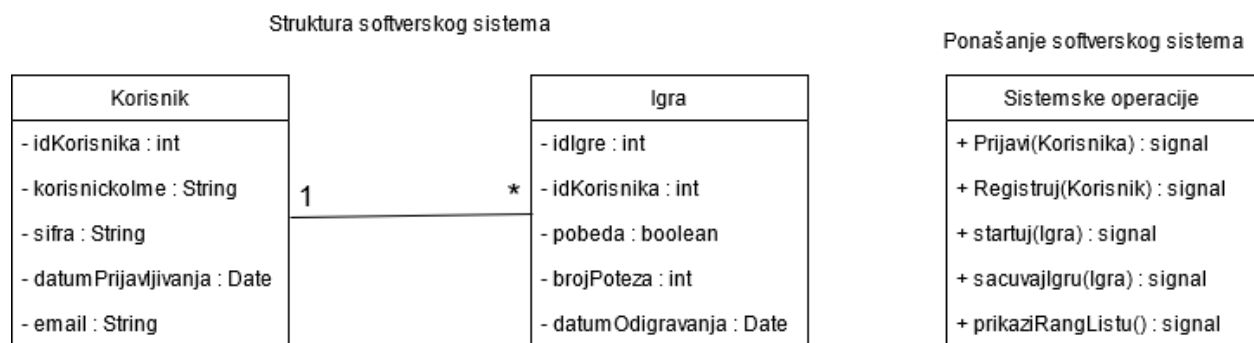
Korisnik (idKorisnika, korisnickolme, sifra, datumPrijavljivanja, email)

Igra (idlgre, idKorisnika, pobeda, brojPoteza, datumOdigravanja)

Tabela Korisnik		Prosto vrednosno ograničenje		Složeno vrednosno ograničenje		Strukturalno ograničenje
Atributi	Ime	Tip atributa	Vrednost atributa	Međuzavisnost atributa jedne tabele	Međuza v. atributa više tabela	INSERT/ UPDATE/ DELETE/
	idKorisnika	Integer	Not null and > 0			
	korisnickolme	String	Not null			
	sifra	String	Not null			
	datumPrijavljivanja	Date	Not null			
	email	String	Not null			

Tabela Igra		Prosto vrednosno ograničenje		Složeno vrednosno ograničenje		Strukturno ograničenje
Atributi	Ime	Tip atributa	Vrednost atributa	Međuzavisnost atributa jedne tabele	Međuza v. atributa više tabela	INSERT/ UPDATE/ DELETE/
	idIgre	Integer	Not null and > 0			
	idKorisnika	Integer	Not null and > 0			
	pobeda	boolean	Not null			
	brojPoteza	Integer	Not null and > 0			
	datumOdigravanja	Date	Not null			

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



Slika 12 – Struktura i ponašanje sistema

### 3. Projektovanje

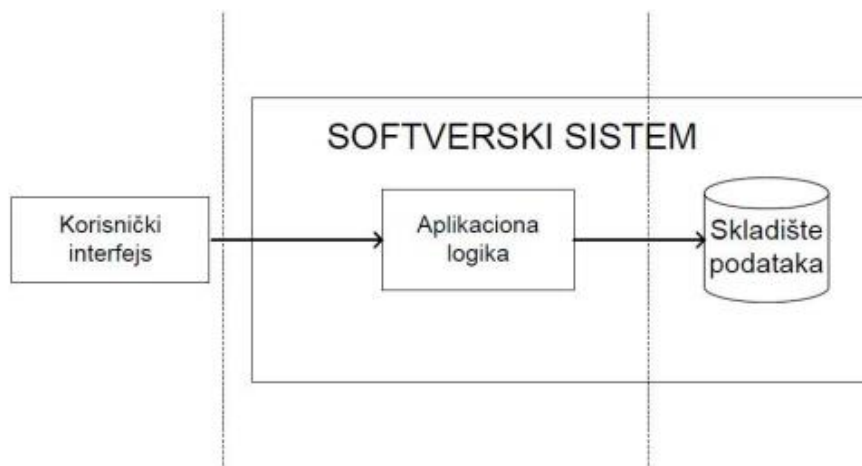
Faza projektovanja opisuje fizičku strukturu i ponašanje softverskog sistema – arhitekturu softverskog sistema.

#### 3.1. Arhitektura softverskog sistema

Arhitektura sistema je tronivojska i sastoji se iz sledećih nivoa:

- korisnički interfejs
- aplikaciona logika
- skladište podataka

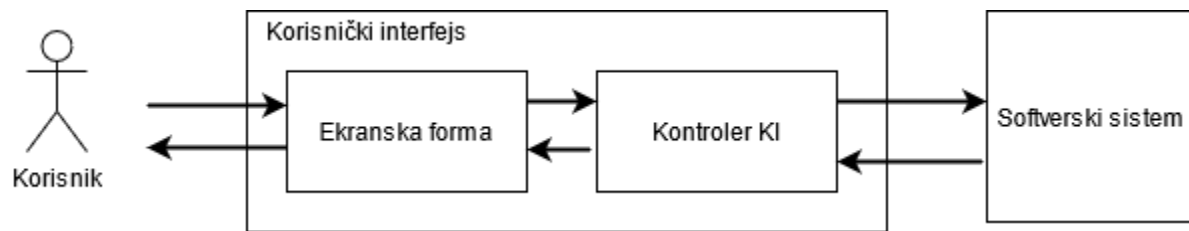
Korisnički interfejs se nalazi na strani klijenta, a aplikaciona logika i skladište podataka na strani servera. Ukoliko postoji potreba, i serverska strana može posedovati korisnički interfejs.



Slika 13 – Tronivojska arhitektura

### 3.2. Projektovanje korisničkog interfejsa

Projektovanje korisničkog interfejsa predstavlja realizaciju ulaza i/ili izlaza softverskog sistema. Ekranska forma ima ulogu da prihvati podatke koje korisnik unosi, prihvata događaje koje pravi korisnik, poziva kontrolera korisničkog interfejsa kako bi mu prosledila podatke i prikazuje podatke dobijene od kontrolera korisničkog interfejsa.



Slika 14 – Struktura korisničkog interfejsa

### 3.2.1. SK1: Slučaj korišćenja – Prijavljivanje korisnika

#### Naziv SK

Prijavljivanje korisnika

#### Aktori SK

Korisnik

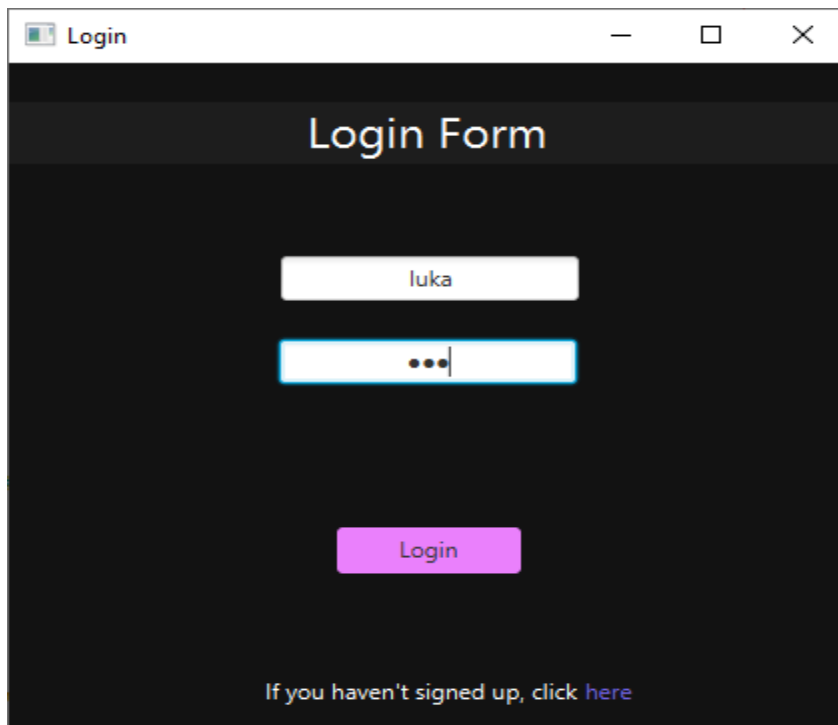
#### Učesnici SK

Korisnik i sistem (program)

**Preduslov:** Sistem je uključen i prikazana je forma za prijavljivanje korisnika.

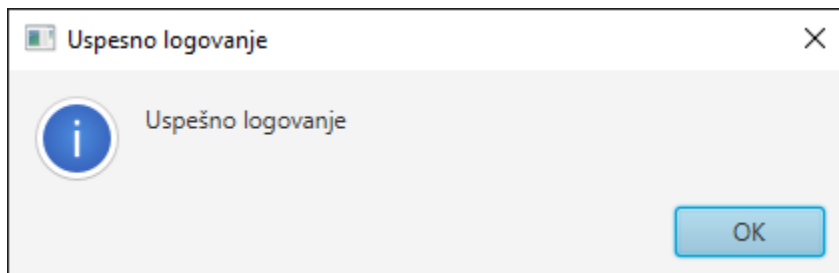
#### Osnovni scenario SK

1. **Korisnik** unosi svoje podatke za prijavljivanje. (APUSO)



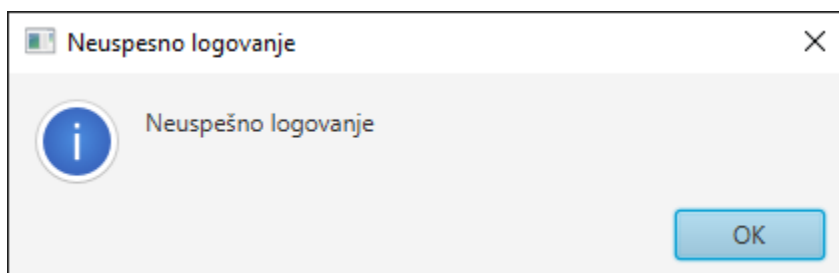
2. **Korisnik** kontroliše da li je korektno uneo svoje podatke. (ANSO)
3. **Korisnik** poziva **sistem** da izvrši prijavljivanje datog korisnika. (APSO)
4. **Sistem** vrši prijavljivanje datog korisnika. (SO)

5. **Sistem** prikazuje **korisniku** poruku: “Uspešno logovanje” i omogućava pristup sistemu. (IA)



#### Alternativna scenarija

- 5.1. Ukoliko **sistem** ne može da pronađe korisnika sa datim podacima, sistem prikazuje **korisniku** poruku “Neuspešno logovanje”. (IA)





### 3.2.2. SK2: Slučaj korišćenja – Registrovanje korisnika

#### Naziv SK

Registrovanje korisnika

#### Aktori SK

Korisnik

#### Učesnici SK

Korisnik i sistem (program)

**Preduslov:** Sistem je uključen i prikazana je forma za registrovanje korisnika.

#### Osnovni scenario SK

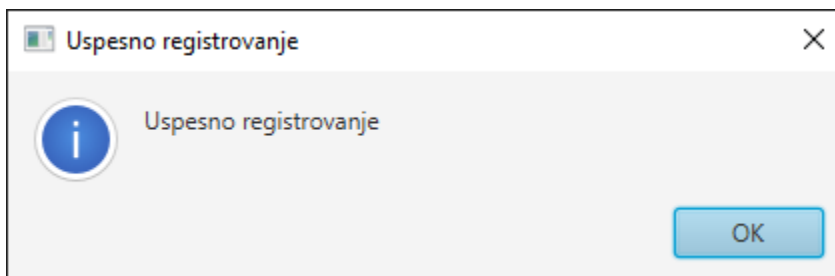
1. **Korisnik** unosi svoje podatke za registrovanje. (APUSO)

here'." data-bbox="173 393 686 781"/>

The image shows a web browser window with the title 'Login'. The main content is a 'Signup Form' on a dark background. It features three white input fields stacked vertically. The first field contains the text 'luka'. The second field contains three dots '...'. The third field contains the email address 'luka@mail.com'. Below these fields is a blue rectangular button with the text 'Signup'. At the bottom of the form, there is a line of text: 'If you have signed up, click [here](#)'.

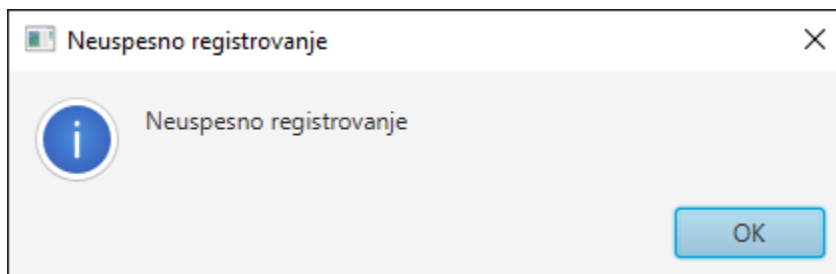
2. **Korisnik** kotroliše da li je korektno uneo svoje podatke. (ANSO)
3. **Korisnik** poziva **sistem** da izvrši registrovanje datog korisnika. (APSO)
4. **Sistem** vrši registrovanje datog korisnika. (SO)

5. **Sistem** prikazuje **korisniku** poruku: “Uspešno registrovanje” i omogućava pristup sistemu. (IA)



### Alternativna scenarija

- 5.1. Ukoliko **sistem** ne može da registruje korisnika sa datim podacima, sistem prikazuje **korisniku** poruku “Neuspešno registrovanje”. (IA)



### 3.2.3. SK3: Slučaj korišćenja – Započinjanje nove igre

#### Naziv SK

Započinjanje nove igre

#### Aktori SK

Korisnik

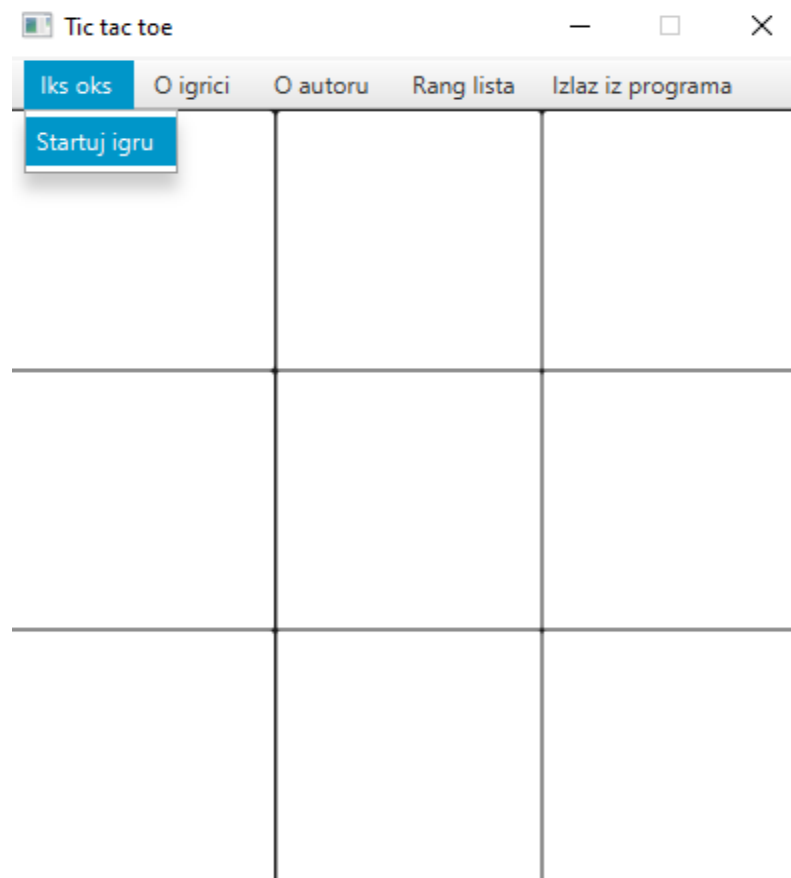
#### Učesnici SK

Korisnik i sistem (program)

**Preduslov:** Sistem je uključen, korisnik je ulogovan svojim korisničkim imenom i šifrom i prikazana je glavna forma.

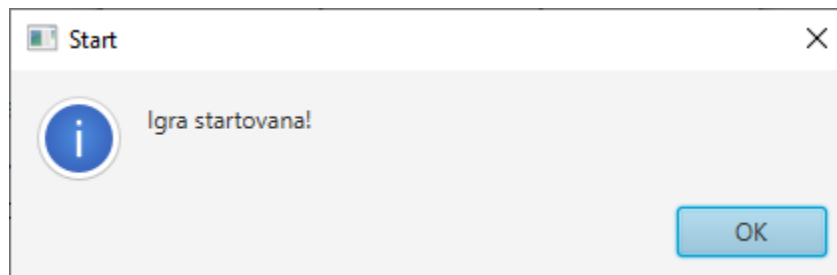
#### Osnovni scenario SK

1. **Korisnik** poziva **sistem** da započne novu igru. (APSO)



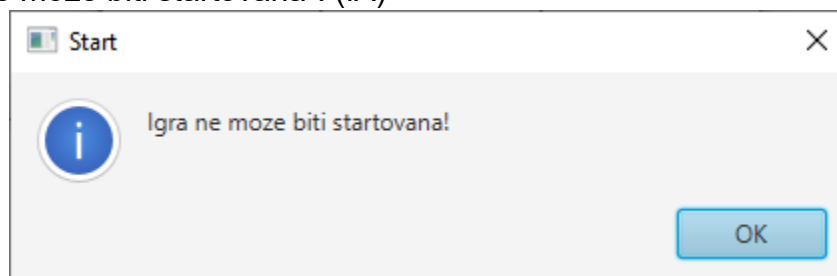
2. **Sistem** započinje novu igru. (SO)

3. **Sistem prikazuje korisniku** poruku: " Igra startovana!". (IA)



#### Alternativna scenarija

- 3.1. Ukoliko **sistem** ne može da startuje igru, sistem prikazuje **korisniku** poruku "Igra ne može biti startovana". (IA)



### 3.2.4. SK4: Slučaj korišćenja – Čuvanje rezultata igre

#### Naziv SK

Čuvanje rezultata igre

#### Aktori SK

Korisnik

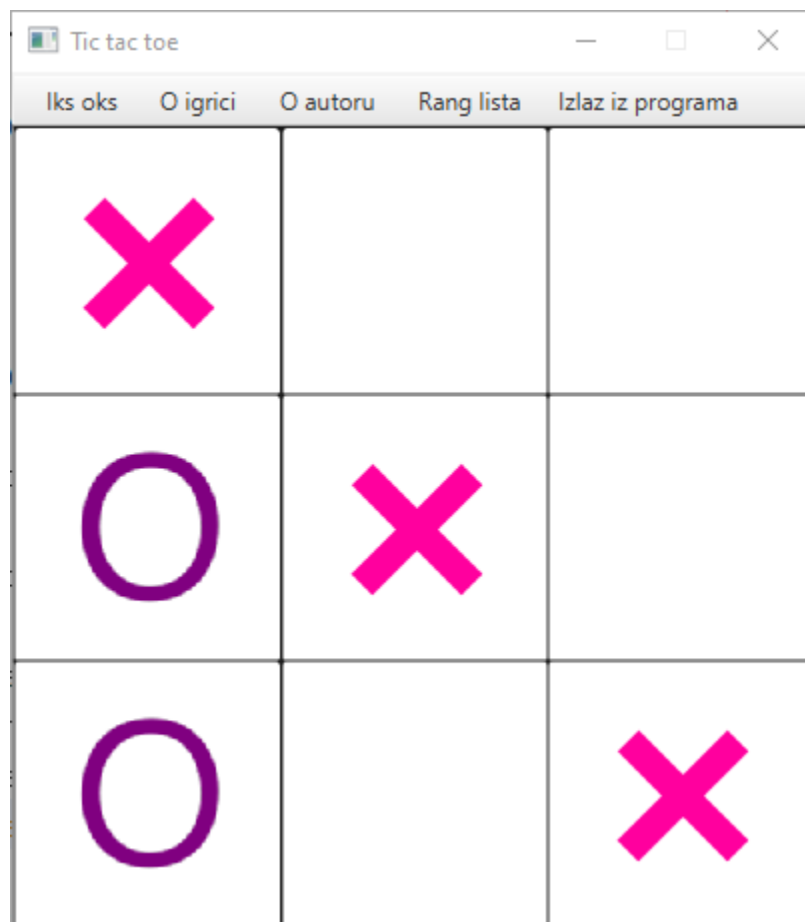
#### Učesnici SK

Korisnik i sistem (program)

**Preduslov:** **Sistem** je uključen, korisnik je ulogovan svojim korisničkim imenom i šifrom i igra je uspešno započeta.

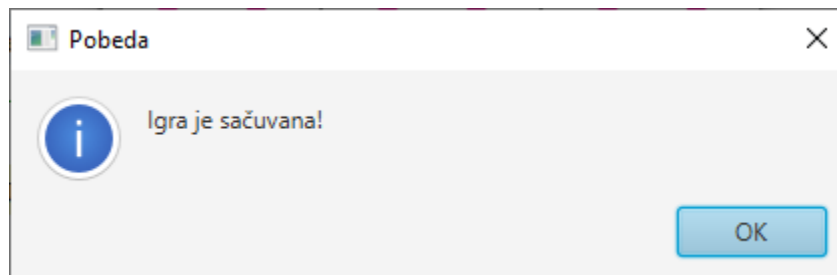
#### Osnovni scenario SK

1. **Korisnik** poziva **sistem** da sačuva rezultat igre. (APSO)



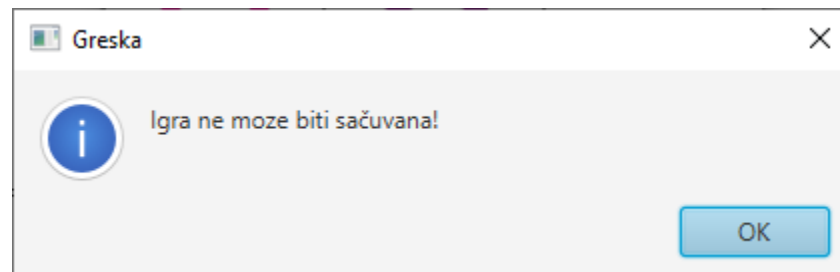
2. **Sistem** čuva novi rezultat. (SO)

3. **Sistem prikazuje korisniku** poruku: "Igra je sačuvana!". (IA)



#### Alternativna scenarija

- 3.1. Ukoliko **sistem** ne može da sačuva rezultat, sistem prikazuje **korisniku** poruku "Igra ne može biti sačuvana". (IA)



### 3.2.5. SK5: Slučaj korišćenja – Prikazivanje rang liste

#### Naziv SK

Prikazivanje rang liste

#### Aktori SK

Korisnik

#### Učesnici SK

Korisnik i sistem (program)

**Preduslov:** **Sistem** je uključen, korisnik je ulogovan svojim korisničkim imenom i šifrom i igra je uspešno započeta.

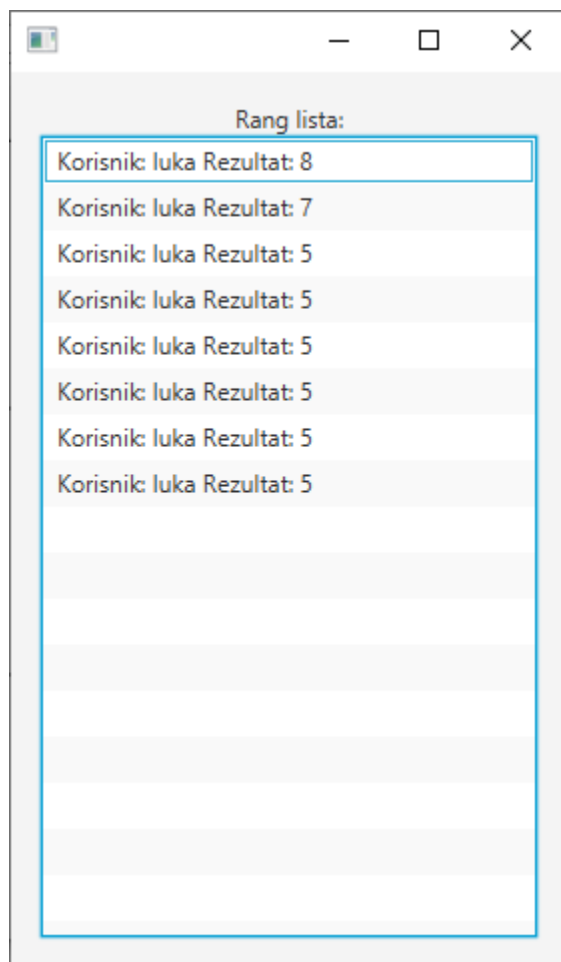
#### Osnovni scenario SK

1. **Korisnik** poziva **sistem** da prikaže rang listu. (APSO)



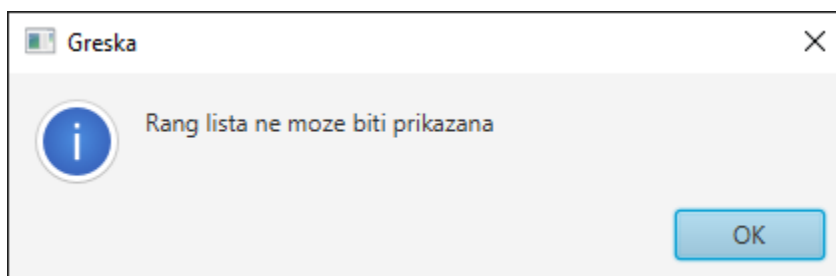
2. **Sistem** vrši nalaženje i kreiranje rang liste. (SO)

3. **Sistem prikazuje korisniku** rang listu. (IA)



**Alternativna scenarija**

- 3.1. Ukoliko **sistem** ne može da prikaže rang listu, sistem prikazuje **korisniku** poruku "Rang lista ne može biti prikazana". (IA)





### 3.3. Komunikacija server – klijent

Deo za komunikaciju podiže serverski soket, kao i veb servis koji će da osluškuju mrežu. Kada klijentski soket uspostavi konekciju sa serverskim soketom, ili se klijent poveže sa serverom preko veb servisa, tada server generiše nit koja će uspostaviti dvosmernu vezu sa klijentom.

Slanje i primanje podataka od klijenta se obavlja razmenom objekata klase DtoRequest i DtoResponse.

Klijent šalje zahtev za izvršenje neke od sistemskih operacija do odgovarajuće niti koja je povezana sa tim klijentom. Ta nit prihvata zahtev i prosleđuje ga do kontrolera aplikacione logike. Nakon izvršenja sistemske operacije rezultat se, preko kontrolera aplikacione logike, u obliku novog objekta klase DtoResponse, i vraća do niti klijenta koja taj rezultat šalje nazad do klijenta.

```
public class RequestDto implements Serializable{
```

```
    private Korisnik objekat;  
    private Igra igra;  
    private Rang rang;  
    private int operacija;
```

```
    public Korisnik getObjekat() {  
        return objekat;  
    }  
}
```

```
    public void setObjekat(Korisnik objekat) {  
        this.objekat = objekat;  
    }  
}
```

```
    public Igra getIgra() {  
        return igra;  
    }  
}
```

```
    public void setIgra(Igra igra) {  
        this.igra = igra;  
    }  
}
```

```
    public Rang getRang() {  
        return rang;  
    }  
}
```

```
    public void setRang(Rang rang) {
```

```

        this.rang = rang;
    }

    public int getOperacija() {
        return operacija;
    }

    public void setOperacija(int operacija) {
        this.operacija = operacija;
    }
}

```

```

public class ResponseDto implements Serializable{

    private Korisnik objekat;
    private Igra igra;
    private List<? extends GeneralIDObject> rangLista;
    private boolean done;

    public Korisnik getObjekat() {
        return objekat;
    }

    public void setObjekat(Korisnik objekat) {
        this.objekat = objekat;
    }

    public Igra getIgra() {
        return igra;
    }

    public void setIgra(Igra igra) {
        this.igra = igra;
    }

    public List<? extends GeneralIDObject> getRangLista() {
        return rangLista;
    }

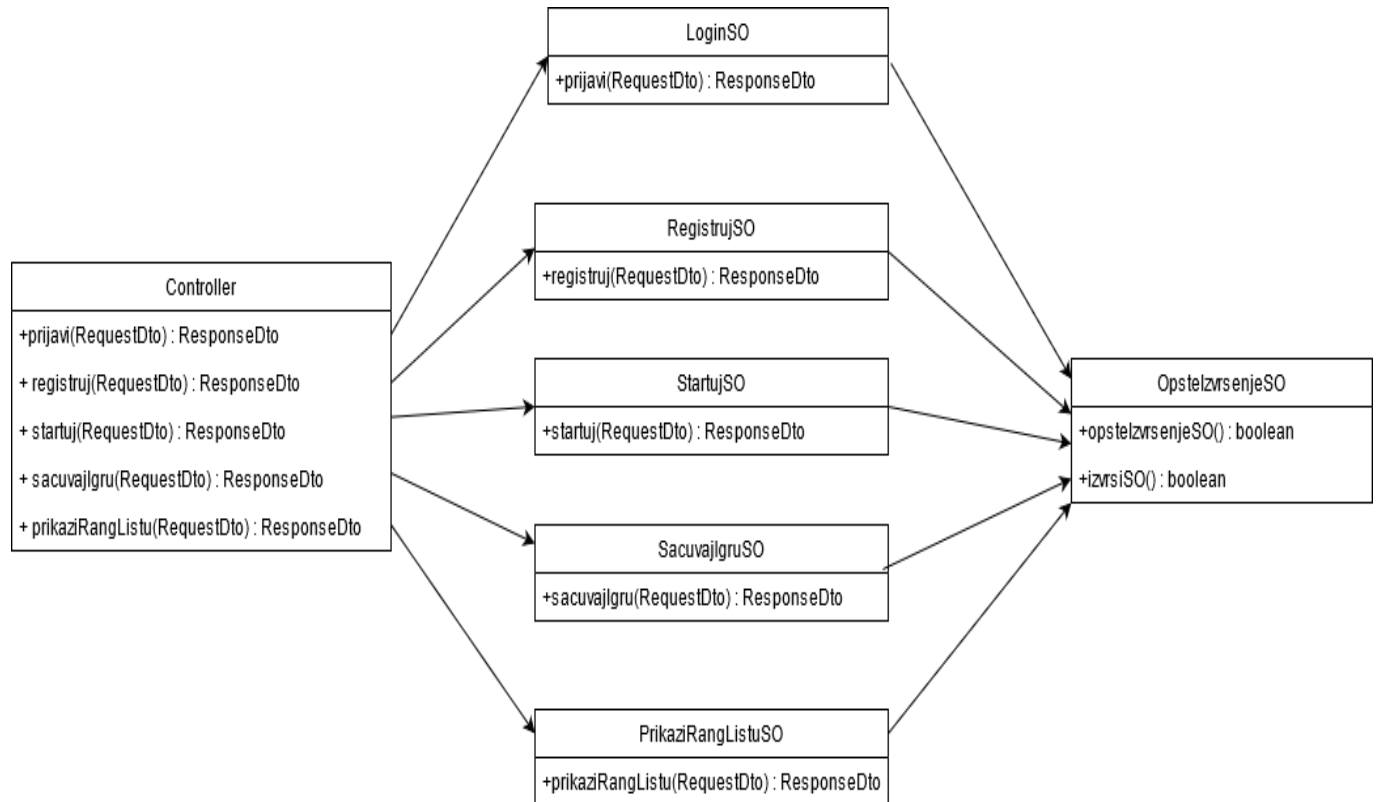
    public void setRangLista(List<? extends GeneralIDObject> rangLista) {
        this.rangLista = rangLista;
    }
}

```

```
public boolean isDone() {  
    return done;  
}  
  
public void setDone(boolean done) {  
    this.done = done;  
}  
}
```

### 3.4. Kontroler aplikacione logike

Kontroler aplikacione logike prihvata zahtev za izvršenje sistemske operacije od niti klijenta i dalje ga preusmerava do klasa koje su odgovorne za izvršenje sistemskih operacija. Nakon izvršenja sistemske operacije kontroler aplikacione logike prihvata rezultat i prosleđuje ga pozivaocu (niti klijenta).



Slika 15 – Prikaz zavisnosti kontrolera i klasa odgovornih za izvršenje sistemskih operacija

Sve klase koje su zadužene za izvođenje sistemskih operacija nasleđuju klasu **OpstelzvršenjeSO**, pa samim tim i implementiraju apstraktnu metodu **izvrsiSO()**. Za metodu **opstelzvršenjeSO()** je implementacija data i ona nije apstraktna.

```

public abstract class OpstelzvrjenjeSO {

    static public BrokerBazePodataka bbp = new BrokerBazePodataka1();
    GeneralDObject gdo;

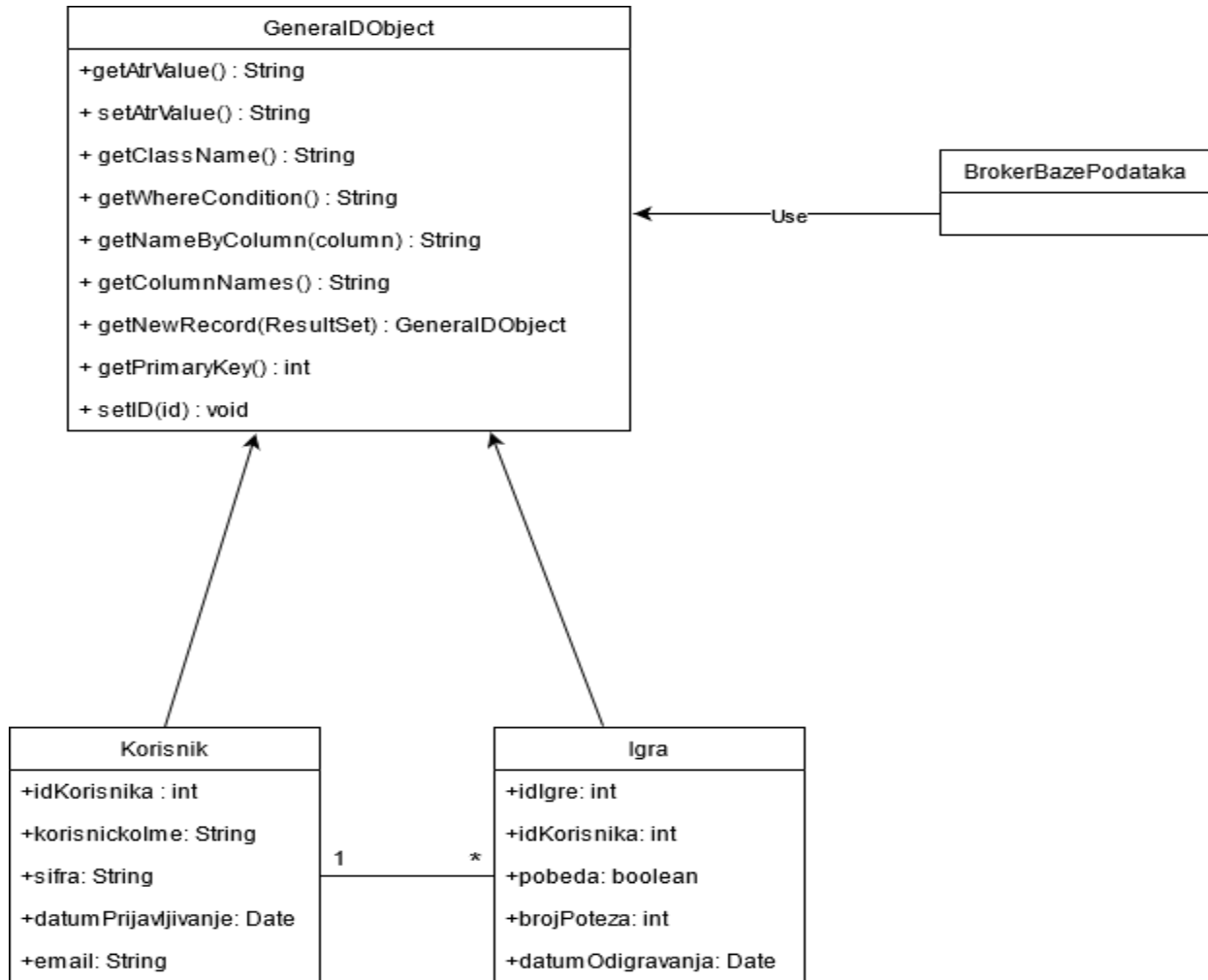
    synchronized public boolean opstelzvrjenjeSO() {
        bbp.makeConnection();
        boolean signal = izvrsiSO();
        if (signal == true) {
            bbp.commitTransation();
        } else {
            bbp.rollbackTransation();
        }
        bbp.closeConnection();
        return signal;
    }

    abstract public boolean izvrsiSO();
}

```

### 3.5. Projektovanje strukture softverskog sistema – Domenske klase

Sve domenske klase implementiraju interfejs **GeneralIDObject**, koji se koristi kao parametar u metodama brokera baze podataka, kako bi se obezbedila inverzija zavisnosti i omogućila implementacija generičkih metoda.



Slika 16 – Domenske klase i zavisnost Brokera od GeneralIDObjecta

```

public class Korisnik extends GeneralIDObject {

    private int idKorisnika;
    private String korisnickolme;
    private String sifra;
    private Date datumPrijavljivanja;
    private String email;

    public Korisnik(){
        this.idKorisnika = 0;
        this.korisnickolme = "";
        this.sifra = "";
        this.datumPrijavljivanja = new Date();
        this.email = "";
    }

    public Korisnik(int idKorisnika, String korisnickolme, String sifra, Date
datumPrijavljivanja, String email) {
        this.idKorisnika = idKorisnika;
        this.korisnickolme = korisnickolme;
        this.sifra = sifra;
        this.datumPrijavljivanja = datumPrijavljivanja;
        this.email = email;
    }

    public int getIdKorisnika() {
        return idKorisnika;
    }

    public void setIdKorisnika(int idKorisnika) {
        this.idKorisnika = idKorisnika;
    }

    public String getKorisnickolme() {
        return korisnickolme;
    }

    public void setKorisnickolme(String korisnickolme) {
        this.korisnickolme = korisnickolme;
    }

    public String getSifra() {
        return sifra;
    }

    public void setSifra(String sifra) {

```

```

        this.sifra = sifra;
    }

    public Date getDatumPrijavljivanja() {
        return datumPrijavljivanja;
    }

    public void setDatumPrijavljivanja(Date datumPrijavljivanja) {
        this.datumPrijavljivanja = datumPrijavljivanja;
    }

    public java.util.Date getDatumRodjenjaSQLDate(java.util.Date datumRodjenja) {
        SimpleDateFormat sm = new SimpleDateFormat("yyyy-MM-dd");
        this.datumPrijavljivanja = java.sql.Date.valueOf(sm.format(datumRodjenja));
        return this.datumPrijavljivanja;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    @Override
    public String getAtrValue() {
        return (idKorisnika + ", " + korisnickolme + ", " + sifra + ", " + datumPrijavljivanja
+ ", " + email + "");
    }

    @Override
    public String setAtrValue() {
        return "idKorisnika=" + idKorisnika + ", " + "korisnickolme=" + korisnickolme + ", "
+ "sifra=" + sifra
        + ", datumRodjenja=" + getDatumRodjenjaSQLDate(datumPrijavljivanja) + ",
" + "email=" + email + "";
    }

    @Override
    public String getClassName() {
        return getClass().getSimpleName();
    }

```



```

@Override
public String getWhereCondition() {
    return "korisnickolme=" + korisnickolme + " and sifra=" + sifra;
}

@Override
public String getNameByColumn(int column) {
    Field[] fields = getClass().getDeclaredFields();
    return fields[column].getName();
}

@Override
public GeneralIDObject getNewRecord(ResultSet rs) throws SQLException {
    return new Korisnik(rs.getInt("idKorisnika"), rs.getString("korisnickolme"),
rs.getString("sifra"), rs.getDate("datumPrijavljivanja"), rs.getString("email"));
}

@Override
public int getPrimaryKey() {
    return this.idKorisnika;
}

@Override
public void setID(int id) {
    this.idKorisnika = id;
}
}

```

```

public class Igra extends GeneralIDObject{

    private int idIgre;
    private int idKorisnika;
    private boolean pobeda;
    private int brojPoteza;
    private Date datumOdigravanja;

    public Igra() {
        idIgre = 0;
        idKorisnika = 0;
        pobeda = false;
        brojPoteza = 0;
        datumOdigravanja = new Date();
    }

    public Igra(int idIgre, int idKorisnika, boolean pobeda, int brojPoteza, Date
datumOdigravanja) {
        this.idIgre = idIgre;
        this.idKorisnika = idKorisnika;
        this.pobeda = pobeda;
        this.brojPoteza = brojPoteza;
        this.datumOdigravanja = datumOdigravanja;
    }

    public int getIdIgre() {
        return idIgre;
    }

    public void setIdIgre(int idIgre) {
        this.idIgre = idIgre;
    }

    public int getIdKorisnika() {
        return idKorisnika;
    }

    public void setIdKorisnika(int idKorisnika) {
        this.idKorisnika = idKorisnika;
    }

    public boolean isPobeda() {
        return pobeda;
    }

    public void setPobeda(boolean pobeda) {

```

```

        this.pobeda = pobeda;
    }

    public int getBrojPoteza() {
        return brojPoteza;
    }

    public void setBrojPoteza(int brojPoteza) {
        this.brojPoteza = brojPoteza;
    }

    public Date getDatumOdigravanja() {
        return datumOdigravanja;
    }

    public void setDatumOdigravanja(Date datumOdigravanja) {
        this.datumOdigravanja = datumOdigravanja;
    }

    public java.util.Date getDatumRodjenjaSQLDate(java.util.Date datumOdigravanja) {
        SimpleDateFormat sm = new SimpleDateFormat("yyyy-MM-dd");
        this.datumOdigravanja = java.sql.Date.valueOf(sm.format(datumOdigravanja));
        return this.datumOdigravanja;
    }

    @Override
    public String getAtrValue() {
        return (idIgre + ", " + idKorisnika + ", " + pobeda + ", " + brojPoteza + ", " + datumOdigravanja + "");
    }

    @Override
    public String setAtrValue() {
        return "idIgre=" + idIgre + ", " + "idKorisnika=" + idKorisnika + ", " + "pobeda=" + pobeda
            + ", brojPoteza=" + brojPoteza + ", " + "datumOdigravanja=" + getDatumRodjenjaSQLDate(datumOdigravanja) + "";
    }

    @Override
    public String getClassName() {
        return getClass().getSimpleName();
    }

    @Override
    public String getWhereCondition() {

```

```

        return "idlgre=" + idlgre + " and idKorisnika=" + idKorisnika;
    }

    @Override
    public String getNameByColumn(int column) {
        Field[] fields = getClass().getDeclaredFields();
        return fields[column].getName();
    }

    @Override
    public GeneralDBObject getNewRecord(ResultSet rs) throws SQLException {
        return new Igra(rs.getInt("idlgre"), rs.getInt("idKorisnika"), rs.getBoolean("pobeda"),
rs.getInt("brojPoteza"), rs.getDate("datumOdigravanja"));
    }

    @Override
    public int getPrimaryKey() {
        return this.idlgre;
    }

    @Override
    public void setID(int id) {
        this.idlgre = id;
    }
}

```

### 3.6. Projektovanje poslovne logike

U fazi analize odredili smo ugovore o sistemskim operacijama, pri čemu smo rekli da jedan ugovor opisuje ponašanje jedne sistemske operacije, tako što opisuje **ŠTA** operacija treba da radi, ali ne i **KAKO**.

U fazi projektovanja za svaki od ugovora se projektuje konceptualno rešenje (realizacija) sistemske operacije. To znači da ćemo za svaku klasu odgovornu za izvršenje sistemske operacije definisati **KAKO** će se sistemska operacija izvršiti.

Aspekti realizacije koji se odnose na konekciju sa bazom, perzistentnost i transakcije treba izbeći u početku projektovanja sistemske operacije.

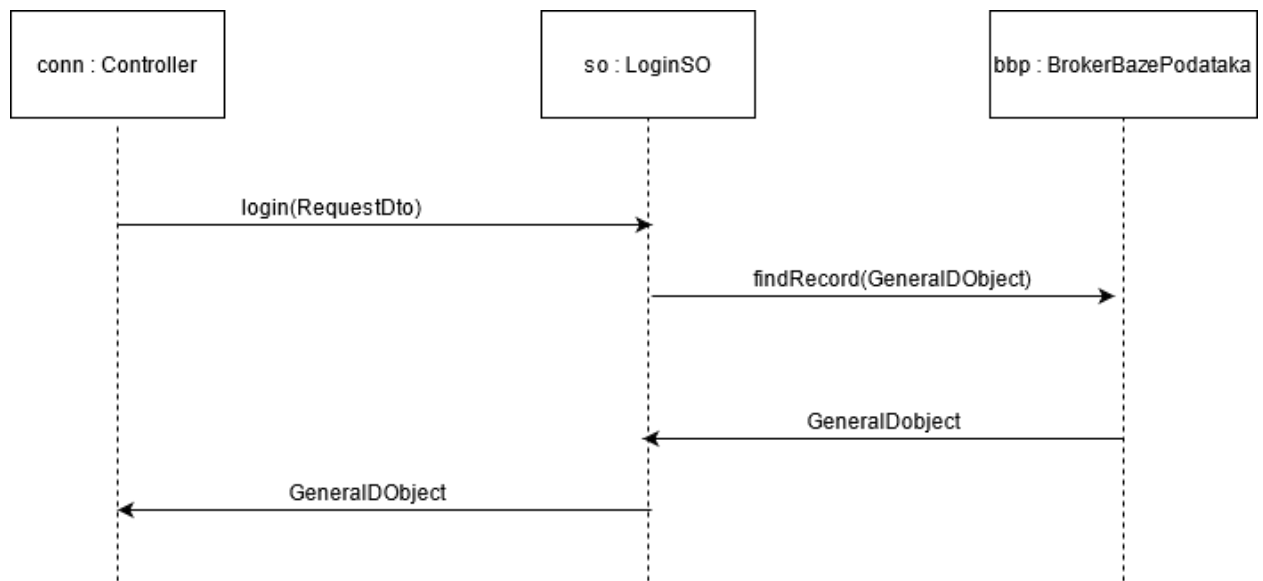
#### 1. Ugovor UG1: Prijavi

**Operacija:** Prijavi(Korisnik): signal

**Veza za SK:** SK1

**Preduslovi:** /

**Postuslovi:** /



Slika 17 - Ugovor UG1

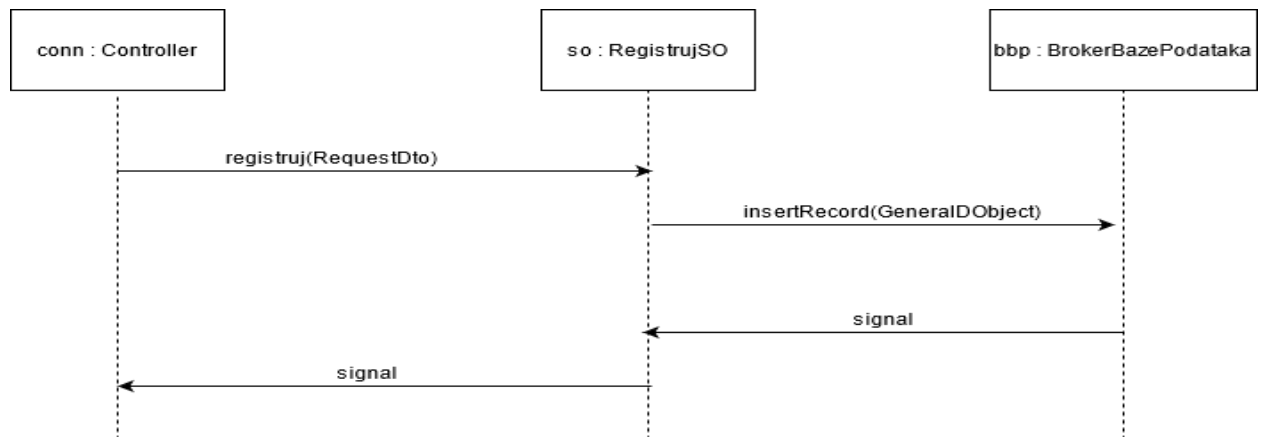
## 2. Ugovor UG2: Registruj

**Operacija:** Registruj(Korisnik): signal

**Veza za SK:** SK2

**Preduslovi:** Vrednosna i strukturna ograničenja nad objektom Korisnik moraju biti zadovoljena

**Postuslovi:** /



Slika 18 - Ugovor UG2

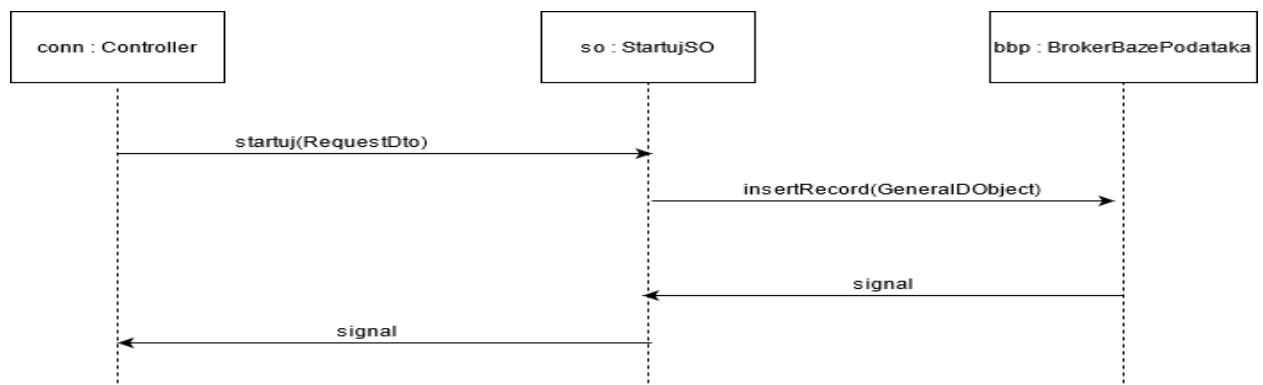
## 3. Ugovor UG3: Startuj

**Operacija:** Startuj(Igra): signal

**Veza za SK:** SK3

**Preduslovi:** Vrednosna i strukturna ograničenja nad objektom Igra moraju biti zadovoljena

**Postuslovi:** Igra je započeta



Slika 19 - Ugovor UG3

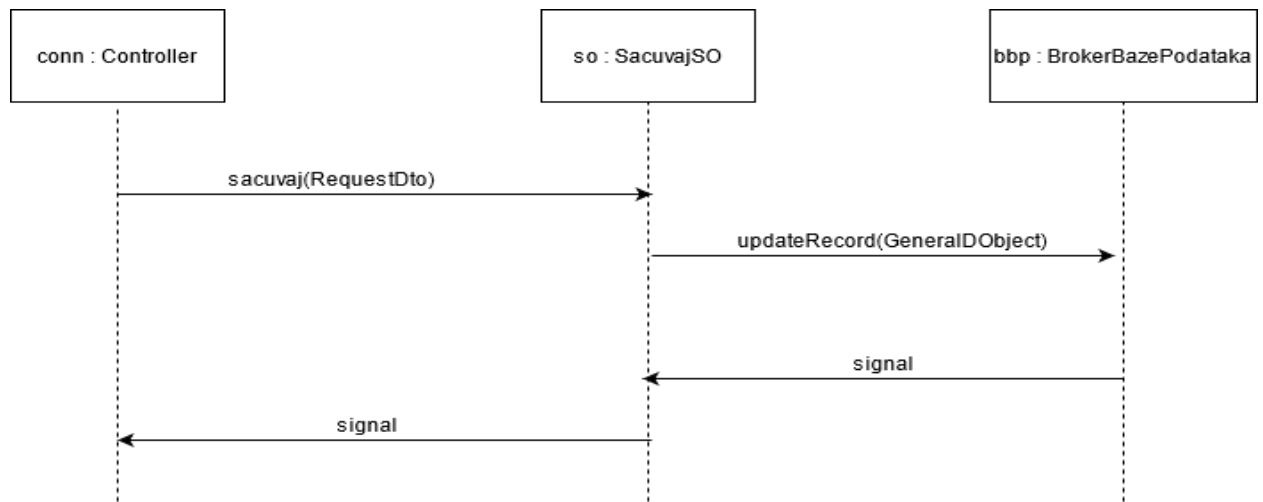
#### 4. Ugovor UG4: SacuvajIgru

**Operacija:** SacuvajIgru(Igra): signal

**Veza za SK:** SK4

**Preduslovi:** Vrednosna i strukturna ograničenja nad objektom Igra moraju biti zadovoljena

**Postuslovi:** Igra je sačuvana



Slika 20 - Ugovor UG4

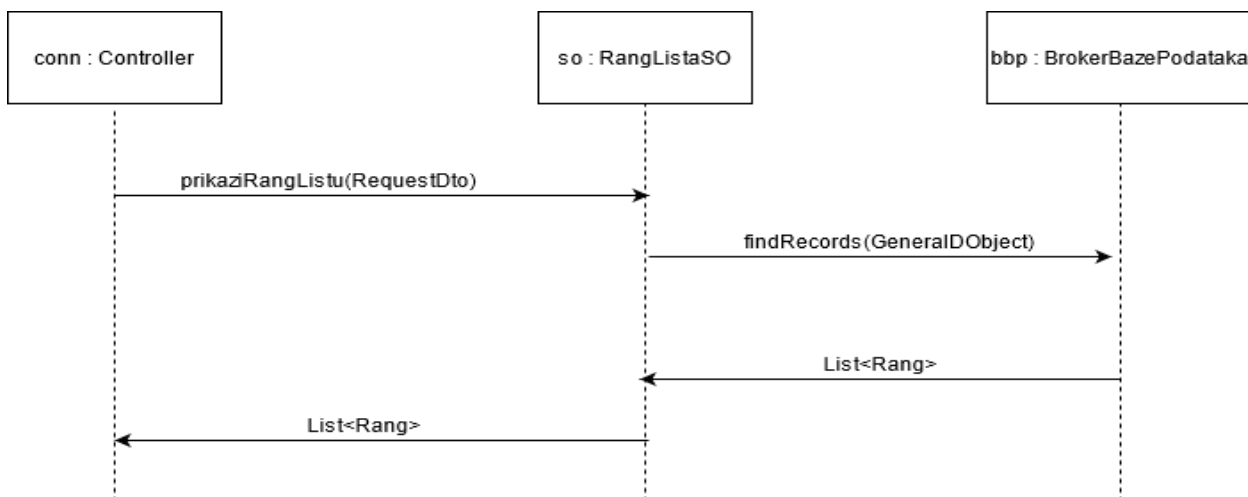
## 5. Ugovor UG5: PrikaziRangListu

**Operacija:** PrikaziRangListu(): signal

**Veza za SK:** SK5

**Preduslovi:** /

**Postuslovi:** /



Slika 21 - Ugovor UG5



### 3.7. Projektovanje brokera baze podataka

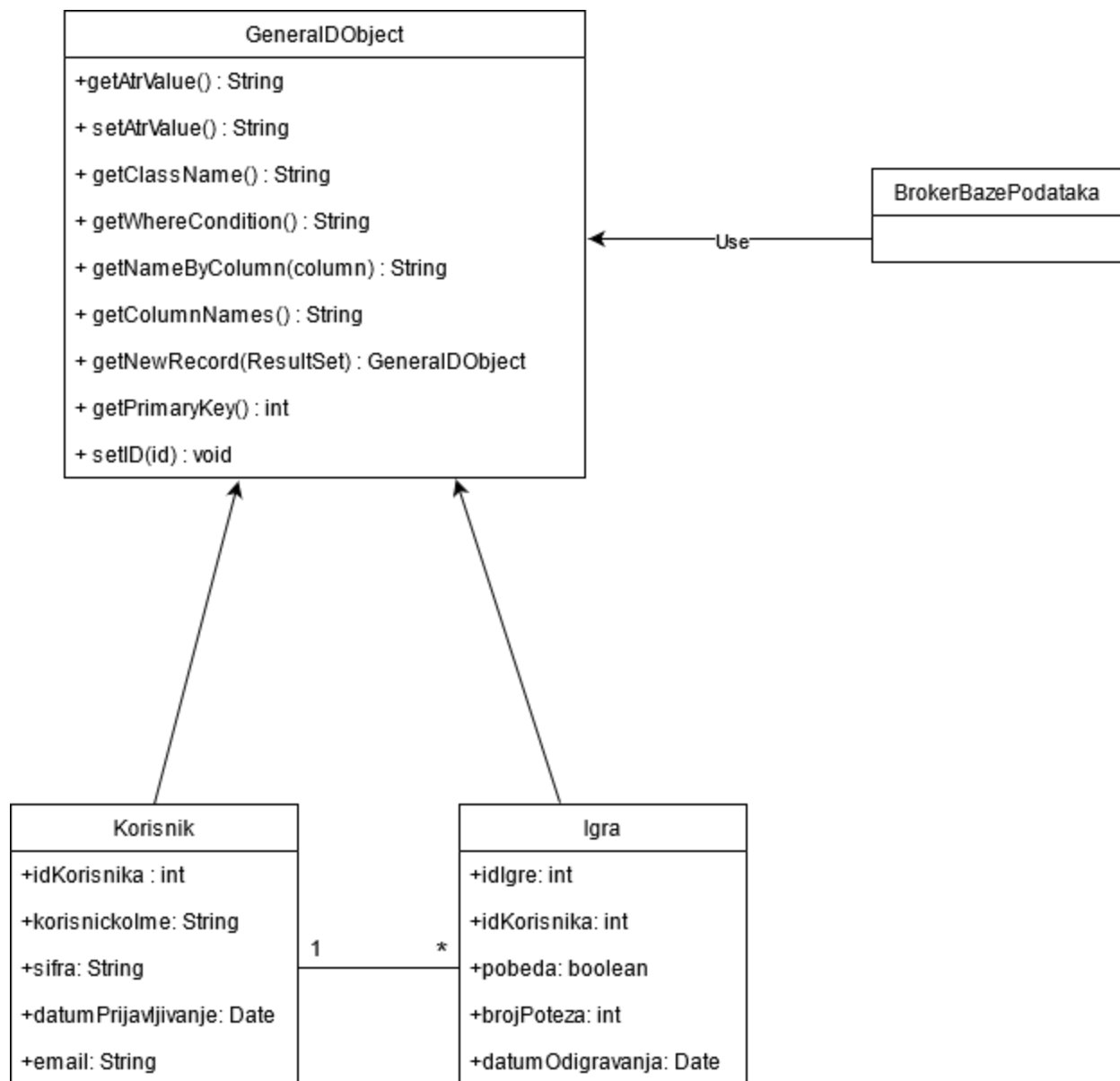
Klasa **BrokerBazePodataka** je projektovana kako bi se obezbedila perzistenciju objekata domenskih klasa i nakon završetka rada programa.

Metode ove klase su generički projektovane, jer kao parametar primaju **GeneralIDObject**, pa samim tim i sve njene podklase. Metode brokera baze podataka su:

```
public abstract boolean makeConnection();
public abstract boolean insertRecord(GeneralIDObject odo);
public abstract boolean updateRecord(GeneralIDObject odo,GeneralIDObject odoold);
public abstract boolean updateRecord(GeneralIDObject odo);
public abstract boolean deleteRecord(GeneralIDObject odo);
public abstract boolean deleteRecords(GeneralIDObject odo,String where);
public abstract GeneralIDObject findRecord(GeneralIDObject odo);
public abstract List<GeneralIDObject> findRecord(GeneralIDObject odo,String where);
public abstract boolean commitTransation();
public abstract boolean rollbackTransation();
public abstract boolean increaseCounter(GeneralIDObject odo,AtomicInteger counter);
public abstract void closeConnection();
public abstract GeneralIDObject getRecord(GeneralIDObject odo,int index);
public abstract int getRecordsNumber(GeneralIDObject odo);
```

Svaka od navedenih metoda pri izvršenju poziva metode klase **GeneralIDObject**, čije podklase obebeđuju sopstvene implementacije. Metode klase **GeneralIDObject**:

```
abstract public String getAtrValue();
abstract public String setAtrValue();
abstract public String getClassName();
abstract public String getWhereCondition();
abstract public String getNameByColumn(int column);
abstract public String getColumnNames();
abstract public GeneralIDObject getNewRecord(ResultSet rs) throws SQLException;
abstract public int getPrimaryKey();
abstract public void setID(int id);
```



Slika 22 - Veza brokera baze podataka sa domenskim klasama

### 3.8. Projektovanje skladišta podataka

Na osnovu softverskih klasa strukture, projektovane su tabele (skladišta podataka) relacionog sistema za upravljanje bazom podataka. U ovom radu korišćen je SUBP „MySQL“:

<input type="checkbox"/>	idKorisnika	int	11		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<input type="checkbox"/>	korisnickoIme	varchar	100		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<input type="checkbox"/>	sifra	varchar	15		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<input type="checkbox"/>	datumPrijavljivanja	datetime			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Slika 23 - Tabela Korisnik

<input type="checkbox"/>	Column Name	Data Type	Length	Default	PK?	Not Null?	Unsigned?	Auto Incr?	Zerofill?	On Update
<input type="checkbox"/>	idIgre	int	11		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<input type="checkbox"/>	idKorisnika	int	11		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<input type="checkbox"/>	pobeda	tinyint	1		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<input type="checkbox"/>	brojPoteza	int	11		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<input type="checkbox"/>	datumOdigravanja	datetime			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Slika 24 - Tabela Igra

## **4. Implementacija**

## 4.1. Mehanizam refleksije

Mehanizam refleksije predstavlja način ispitivanja ili modifikovanja ponašanja metoda, klasa ili interfejsa u vreme izvršenja programa. Ovaj mehanizam se realizuje pomoću metapodataka klasa.

Na sledećem primeru se može videti izvršenje refleksije u klasi GameController.

```
for (Field f : docCtrl.getClass().getDeclaredFields()) {
    if (f.getType() == Button.class) {
        try {
            Button button = (Button) f.get(docCtrl);
            button.setOnMouseClicked(new OsluskivacDugmeUlgri(this, button));
        } catch (IllegalAccessException ex) {

            Logger.getLogger(FXMLDocumentController.class.getName()).log(Level.SEVERE, null,
            ex);

            popupPoruka("Start", "Igra ne moze biti startovana!");
        }
    }
}
```

Na ovaj način dobijamo objekte klase Button preko refleksije polja FXMLDocumentController.

## 4.2. *Generics* mehanizam

Po analogiji sa mehanizmom apstrakcije, o kojem je ranije bilo reči, ovaj mehanizam omogućava parametrizovanje metoda, klasa i interfejsa. Kada je potrebno kreirati metodu koja vrši operacije nad različitim tipovima podataka, a čije izvršenje nije zavisno od ulaznih parametara, moguće je kreirati generičku metodu.

Generička metoda predstavlja metodu koja je invarijantna u odnosu na parametre. Na sledećem primeru biće prikazan način na koji je moguće ubaciti elemente u grafički element ComboBox nezavisno od njihovog tipa.

```
private <T> VBox dodajUListu(List<T> elements) {  
    ObservableList<T> observableList = FXCollections.observableArrayList();  
    for (T t : elements) {  
        observableList.add(t);  
    }  
    Label label = new Label("Rang lista:");  
    ListView<T> listView = new ListView<>(observableList);  
    VBox vbox = new VBox();  
    vbox.setAlignment(Pos.CENTER);  
    vbox.setPadding(new Insets(15));  
    vbox.getChildren().addAll(label, listView);  
    return vbox;  
}
```

Na prethodnim primerima uočava se izdvajanje opšteg i uklanjanje specifičnosti. Pomoću ovih metoda moguće je popuniti grafički element objektima nezavisno od njihovog tipa.

## **5. Testiranje**

Svaki od implementiranih slučajeva korišćenja je testiran manuelno. Prilikom testiranja svakog od ovih slučajeva korišćenja, pored unetih pravilnih podataka, unošeni su i nepravilni podaci da bi se utvrdilo kakav će biti rezultat izvršenja ovih sistemskih operacija.

Na osnovu izvršenog testiranja, otklonjeni su uočeni nedostaci.

## **6. Zaključak**

Kroz ovaj rad je opisan postupak razvoja softverskog sistema pomoću uprošćene Larmanove metode, koja se sastoji od pet faza:

1. Prikupljanje zahteva od korisnika
2. Analiza
3. Projektovanje
4. Implementacija
5. Testiranje

Prilikom razvoja razmatrani su principi, metode kao i strategije projektovanja softvera. Da bi softverski sistem mogao lako da se održava i nadgrađuje, korišćeni su uzori projektovanja, koji razdvajaju generalne delove od specifičnih. Pokazan je primer MVC makro-arhitekture. Takođe, korišćen je i mehanizam refleksije.

Ovako izgrađen sistem predstavlja modularan sistem koji je nadgradiv, jednostavniji za održavanje i koji treba da omogući efikasno vođenje evidencije o igračima i samoj igri.

## **7. Principi, metode i strategije projektovanja softverskog sistema**

### **7.1. Principi projektovanja softverskog sistema**

#### **7.1.1. Apstrakcija**

“Apstrakcija je proces svesnog zaboravljanja informacija, tako da stvari koje su različite mogu biti tretirane kao da su iste”.

Pod apstrakcijom podrazumevamo izdvajanje opštih, generičkih osobina, ne obraćajući pažnju na detalje i specifičnosti.

U kontekstu projektovanja softvera, postoje dva ključna mehanizma apstrakcije:

1. Parametrizacija
2. Specifikacija

Apstrakcija specifikacijom vodi do tri glavne vrste apstrakcija:

1. Proceduralna apstrakcija
2. Apstrakcija podataka
3. Apstrakcija kontrolom

Parametrizacija je apstrakcija koja izdvaja iz nekog skupa elemenata njihova opšta svojstva koja su predstavljena preko parametara.

Postoji pet slučajeva parametrizacije:

1. Parametrizacija skupa elemenata prostog tipa
2. Parametrizacija skupa elemenata složenog tipa
3. Parametrizacija skupa operacija
4. Parametrizacija skupa procedura
5. Parametrizacija skupa naredbi



## 1. Parametrizacija skupa elemenata prostog tipa

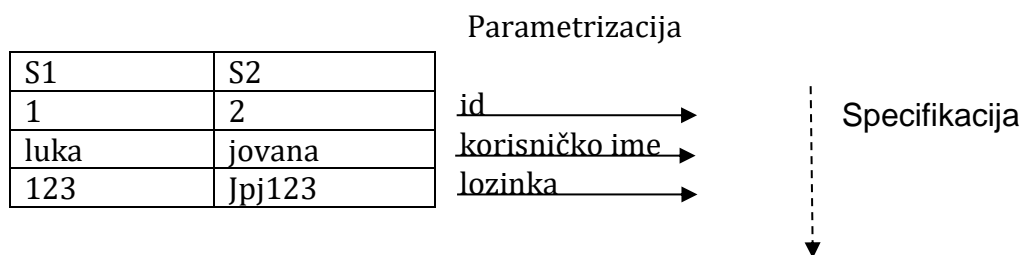
Ukoliko posmatramo skup celih brojeva 1,2,3... njih možemo predstaviti preko nekog opšteg predstavnika skupa celih brojeva. Na primer pomoću parametra x. U prikazanom slučaju, programski bi to zapisali kao: int x;



Slika 25 - Parametrizacija skupa elemenata prostog tipa

## 2. Parametrizacija skupa elemenata složenog tipa

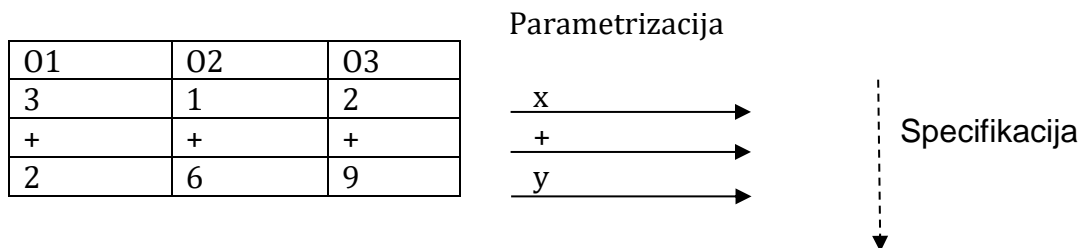
Ukoliko posmatramo neki skup složenih objekata, npr. skup korisnika: (1, "luka", "123" ), (2, "jovana", "jpj123"), tada parametrizacijom dobijamo opšta svojstva tog skupa, odnosno njihove attribute: ID, korisničko ime i lozinku. Navedeni skup objekata označićemo sa S, a elemente skupa sa s1,s2.



- Parametrizacijom dobijamo opšta svojstva elemenata skupa
- Primer je klasa u objektno orijentisanim programskim jezicima, jer predstavlja predstavnika nekog skupa, a parametrizacijom dolazimo do njegovih svojstava, odnosno atributa klase
- Specifikacija sledi parametrizaciju, jer navodi opšta svojstva elemenata skupa, odnosno koristiti rezultate parametrizacije
- Apstrakcija je definisana imenom i specifikacijom skupa:  
Korisnik (id, korisničko ime, lozinka)

### 3. Parametrizacija skupa operacija

Ukoliko posmatramo neki S, skup operacija:  $\{ (3+2), (1+6), (2+9) \}$ , uočavamo opštu operaciju sabiranja nad dva elementa i možemo je definisati sa  $x+y$ . Brojevi  $x$  i  $y$  predstavljaju operande nad kojima se operacija izvršava, dok  $+$  predstavlja ono što operacija radi. Navedeni skup elemenata označen je sa O1, O2 i O3:



- Specificiranjem opštih svojstava operacije, dobijamo njene elemente: operande i operator
- Specificiranjem nekog skupa operacija, iz njega dobijamo opštu operaciju.
- Specifikacijom operacije, za navedeni primer, se dobija opšta operacija za sabiranje dva broja.

### 4. Parametrizacija skupa procedura

Ukoliko posmatramo neki skup S procedura:

s1: *boolean insertRecord(GeneralDObject gdo)* – metoda koja čuva igru

s2: *boolean insertRecord(GeneralDObject gdo)* – metoda koja čuva korisnika.

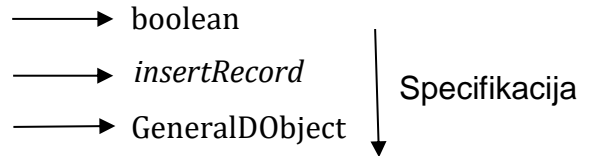
Uočavamo da parametrizacijom dobijamo opštu proceduru:

*boolean insertRecord(GeneralDObject gdo).*

Elementi koji čine proceduru su:

povratna vrednost procedure, naziv procedure, argumenti (parametri) procedure i telo procedure.

	S1	S2
Povratna vrednost procedure	boolean	boolean
naziv	<i>insertRecord</i>	<i>insertRecord</i>
argumenti	GeneralDObject	GeneralDObject
Telo procedure	Metoda koja čuva igru	Metoda koja čuva igrača



- Navođenje opštih svojstava procedure predstavlja specifikaciju procedure
- Specifikacijom procedure se dobija opšta procedura nad nekim skupom procedura
- Rezultat specifikacije procedure je njen potpis
- Kao takav, potpis procedure predstavlja proceduralnu apstrakciju nekog skupa procedura.

U našem primeru dobijamo sledeći potpis: *boolean insertRecord(GeneralDObject gdo)*.

## 5. Parametrizacija skupa naredbi

Ukoliko posmatramo skup S naredbi:

1. S1: `System.out.println("Korisnik "+users[0].getUsername()+" se prikljucio igri!");`
2. S2: `System.out.println("Korisnik "+users[1].getUsername()+" se prikljucio igri!");`

Uočićemo da se parametrizacijom ističu opšta svojstva ovih naredbi, a to su:

- `System.out.println(`
- `"Igrac "`
- `+igraci[i].getKorisnickolme()+`
- `" se prikljucio igri!"`
- `);`

S1	S2
System.out.println(	System.out.println(
"Korisnik "	"Korisnik "
+users[0].getUsername()+	+users[1].getUsername()+
" se prikljucio igri!"	" se prikljucio igri!"
);	);

Parametrizacija:

```

-----> System.out.println(
-----> "Korisnik "
-----> +users[0].getUsername()+
-----> " se prikljucio igri!"
-----> ");

```

Specifikacija



Navođenje opštih svojstava naredbe predstavlja specifikaciju naredbe.

Kao rezultat specifikacije naredbe dobija se:

```
for(int i=0;i<2;i++){ System.out.println("Korisnik "+users[i].getUsername()+" se prikljucio igri!");}
```

Specifikacijom naredbi iz skupa naredbi, dobijamo *opštu naredbu*.

Specifikacija naredbi predstavlja apstrakciju naredbi, jedan od oblika apstrakcije kontrolom.

Specifikacija je apstrakcija koja izdvaja iz nekog skupa elemenata njihova opšta svojstva, koja mogu biti predstavljena preko procedure, podatka ili kontrole.

## 1. Proceduralna apstrakcija

Proceduralnom apstrakcijom izdvajamo iz nekog skupa procedura ono što su njihova opšta svojstva:

- tip povratne vrednosti
- ime procedure
- argument procedure

Kao što je već rečeno, rezultat proceduralne apstrakcije je potpis procedure. Njen rezultat, pored potpisa, mogu biti dodatne informacije o proceduri, kao što su uslovi izvršenja i/ili dodatni opis šta ona radi.

Prednosti proceduralne apstrakcije su da se fokusira na to šta procedura radi, ne bavi se detaljima njene implementacije i načinom realizacije.

## 2. Apstrakcija podataka

- Apstrakcijom podataka izdvajamo opšta svojstva nekog datog skupa podataka

Ukoliko imamo npr. skup igrača {(1, "luka", "123"), (2, "jovana", "jpj123")}, Tada se parametrizacijom dobijaju opšta svojstva skupa: id, korisničko ime, lozinka. Parametrizaciju sledi specifikacija, pa se navođenjem ovih opštih svojstava dobija specifikacija skupa.

Apstrakcija podataka je definisana imenom (Korisnik) skupa i specifikacijom skupa: id, korisničko ime, lozinka.

Ukoliko elementi skupa imaju i strukturu i ponašanje – objekti, tada se nad njima radi i proceduralna apstrakcija i apstrakcija podataka. Rezultat primene ovih apstrakcija je klasa koja ima svoja stanja – atribut i ponašanje-metode. Ukoliko elementi skupa imaju samo ponašanje, rezultat proceduralne apstrakcije će biti interfejs koji sadrži skup potpisa procedura.

## 3. Apstrakcija kontrolom

Postoje dva vida apstrakcije kontrolom:

### 1. *Apstrakcija naredbi*

Ovom apstrakcijom se iz nekog skupa naredbi izdvaja ono što je opšte i predstavlja se pomoću kontrolne strukture i opšte naredbe.

### 2. *Apstrakcija struktura podataka*

Ovom apstrakcijom se iz nekog skupa struktura izdvajaju njihove opšte osobine i predstavljaju se pomoću iteratora koji kontroliše prolazak kroz strukturu podataka.

*Spojenost (coupling) i kohezija (cohesion)*

Ono čemu se teži pri razvoju softverskih sistema je ostvarivanje:

- Što veće kohezije – *high cohesion*
- Što manje povezanih klasa – *low coupling*

### **Kohezija**

Klasa X treba da obezbedi neko ponašanje `m1()`.

```
class X{  
    public m1(){  
    }  
}
```

Ukoliko se pri izvršenju `m1` pozivaju druge metode klase X, `m11`, `m12` i `m13`:

```
class X {  
    public m1() {  
        m11();  
        m12();  
        m13();  
    }  
    private m12(){...}  
    private m13(){...}  
}
```

onda se može reći da navedena klasa ima visoku koheziju. Kohezija nam govori o tome koliko su metode unutar klase međusobno povezane.

Gubitak kohezije znači da je neka klasa odgovorna da obezbedi više različitih ponašanja, koja između sebe nisu povezana.

```
class Y{  
    public m1(){  
        m11(); m12(); m13();  
    }  
    public m2(){  
        m21(); m22(); m23();  
    }  
}
```

```

    public m3(){
        m31(); m32(); m33();
    }
    private m12(){..}
    private m13(){..}
    private m21(){}
}

```

Ovakva klasa je teška za održavanje i nadgradnju.

U ovom primeru vidimo da među privatnim metodama koje spadaju pod izvršenje jedne operacije postoji povezanost (m11,m12,m13), ali da između metoda koje pripadaju različitim izvršenjima (m12, m23), ne postoji povezanost.

Samim tim, zaključujemo da treba praviti klase koje imaju visoku koheziju, radi lakše nadgradnje i održavanja.

### ***Povezanost – Coupling***

Povezanost – kuplovanje znači da klase međusobno zavise jedna od druge. To znači da klasa ne može da obavi neku operaciju bez prisustva druge klase.

Na primer, klasa X, ima metodu m1, koja pri izvršenju poziva metodu m2, klase Y.

U tom slučaju, klasa X zavisi od klase Y.

```

class X{
    Y y;
    public m1(){
        y=new Y(); y.m2();
    }
}
class Y{
    public m2(){..}
}

```

Kuplovanje predstavlja meru povezanosti klase sa drugim klasama, kojom se utvrđuje koliko je jedna klasa zavisna od drugih.

Klasa koja jako zavisi od drugih klasa je klasa sa snažnom povezanošću – *High/strong coupling*.

Zaključak je da treba praviti klase sa slabom povezanošću sa drugim klasama, iako je prihvaćeno da je međusobna povezanost klasa neizbežna.

Primer međusobne povezanosti klasa je klasa *OpstelzvršenjeSO*, koja u svojoj *opstelzvršenjeSO* metodi, iako dobro projektovanoj, prikazuje zavisnost od klase *BrokerBazePodataka*.

```
public abstract class OpstelzvršenjeSO {  
  
    static public BrokerBazePodataka bbp = new BrokerBazePodataka1();  
    GeneralDBObject gdo;  
  
    synchronized public boolean opstelzvršenjeSO() {  
        bbp.makeConnection();  
        boolean signal = izvršiSO();  
        if (signal == true) {  
            bbp.commitTransation();  
        } else {  
            bbp.rollbackTransation();  
        }  
        bbp.closeConnection();  
        return signal;  
    }  
  
    abstract public boolean izvršiSO();  
}
```

Slika 26 - Povezanost klase *OpstelzvršenjeSO* i *BrokerBazePodataka*

Međutim, ovakav nivo povezanosti je nizak, prihvatljiv i neizbežan, jer klasa *OpstelzvršenjeSO* zavisi samo od jedne druge klase.



## Dekompozicija i modularizacija

Dekompozicija predstavlja proces raščlanjivanja, proces koji polazni problem deli u skup potproblema, koji se nezavisno rešavaju. Kada su potproblemi rešeni na taj način, omogućavaju da se polazni problem lakše reši.

Ukoliko primenjujemo princip dekompozicije pri razvoju softverskog sistema, doći će do modularizacije softverskog sistema.

Zaključak je da modularizacija softverskog sistema nastaje kao posledica procesa dekompozicije.

Dekompozicija može biti objašnjena sa više aspekata pri razvoju softvera:

### 1. Dekompozicija kod prikupljanja zahteva

Korisnički zahtev se kod prikupljanja dekomponuje na skup zahteva koji se opisuju preko slučajeva korišćenja: Prijavljivanje i registrovanje korisnika, započinjanje nove igre, čuvanje rezultata igre, prikazivanje rang liste.

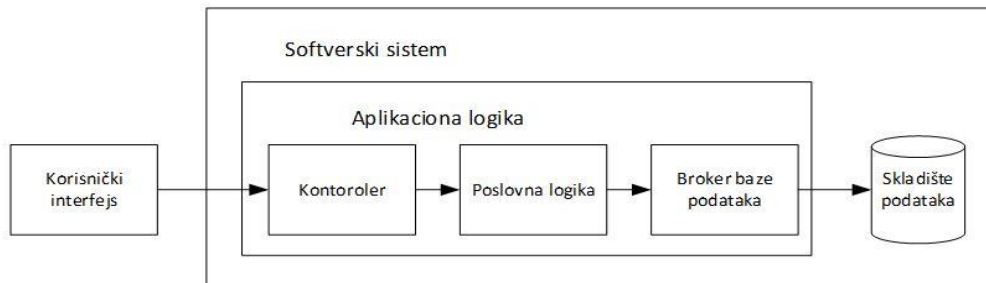


Slika 27 - Dekompozicija kod prikupljanja zahteva

## 2. Dekompozicija kod projektovanja softvera

Pri razvoju softverskog sistema, primenom Larmanove metode se, u trećoj fazi - fazi projektovanja, vrši projektovanje arhitekture softverskog sistema koja je modularna:

1. Modul – korisnički interfejs
2. Modul – aplikaciona logika
3. Modul – skladište podataka



Slika 28 - Dekompozicija kod projektovanja softvera

Svaki modul je dobijen dekompozicijom, može se nezavisno projektovati i implementirati.

Postoje principi koji moraju biti ispunjeni da bi softverski sistem mogao da se dekomponuje u module:

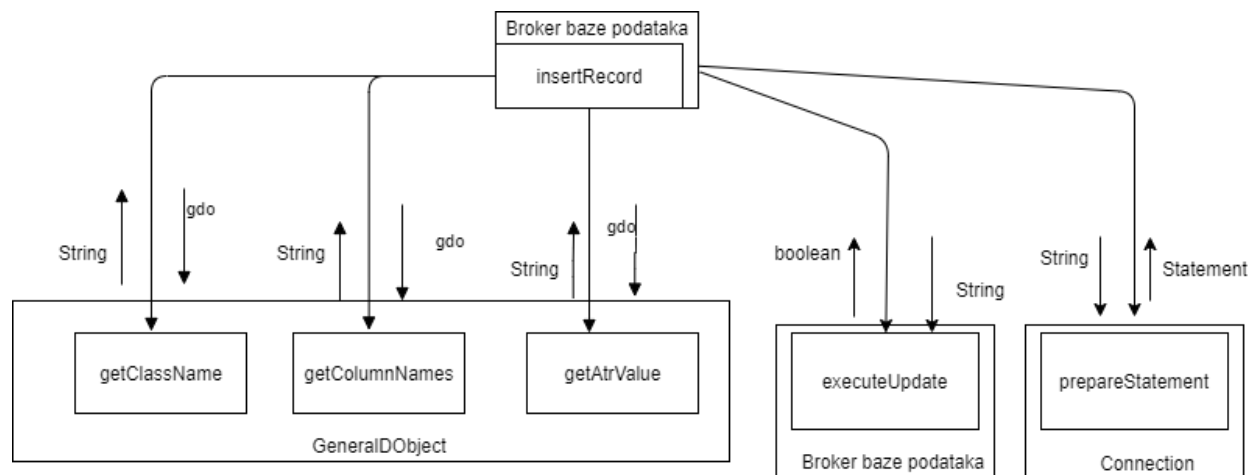
- Moduli treba da imaju jaku koheziju (high cohesion)
- Moduli treba budu što je moguće slabije vezani (low coupling)
- Svaki modul treba da čuva svoje interne informacije (information hiding)

Rezultat procesa dekompozicije je softverski sistem koji je modularizovan.

### 3. Dekompozicija funkcija (metoda)

Funkcija – metoda neke klase koja je složena, može biti dekomponovana na više podfunkcija. Ove podfunkcije se nezavisno rešavaju da bi se na kraju sve integrisale u jednu celinu kako bi se realizovala početna funkcija.

U primeru, metoda brokera baze podataka *insertRecord*.



Slika 29 - Dekompozicija funkcije

### Učaurjenje – enkapsulacija / Sakrivanje informacija – information hiding

Učaurjenje je proces kojim se razdvajaju:

- Osobine modula – klase, koje su javne za druge module
- Od osobina modula koje su skrivene za druge module sistema

Osobine koje su javne, mogu koristiti drugi moduli, za razliku od onih koje nisu javne.

Sakrivanje informacija predstavlja rezultat enkapsulacije, jer se sakrivaju informacije koje drugi moduli ne smeju koristiti.

Kao primer će poslužiti klasa *Kontroler*.

```

public class Kontroler {

    private static Kontroler instance;
    private OpsteIzvršenjeSO so;

    public static Kontroler getInstance() {
        if(instance == null){
            instance = new Kontroler();
        }
        return instance;
    }

    public ResponseDto startuj(RequestDto request){
        so = new StartujSO();
        return ((StartujSO) so).startuj(request);
    }

    public ResponseDto sacuvaj(RequestDto request){
        so = new SacuvajSO();
        return ((SacuvajSO) so).sacuvaj(request);
    }

    public ResponseDto rangLista(RequestDto request){
        so = new RangListaSO();
        return ((RangListaSO) so).prikaziRangListu(request);
    }

}

```

Slika 30 - Enkapsulacija

Ova klasa je implementirana kao *Singleton* klasa, što znači da će samo jednom biti instancirana pri prvom pozivu njene javne metode *getInstance()*.

Naime, ona stavlja na raspolaganje svoju javnu metodu drugim klasama kojima je potrebna njena instanca, ali im ne dozvoljava kreiranje. Kao što je prikazano na slici 30, konstruktor ove klase je privatn, što znači da druge klase kojima je potrebna instanca klase Kontroler, ne znaju da li je ona tog trenutka bila kreirana ili ne. Ono što je sigurno, je, da će tu instancu dobiti i da će instanca biti jedinstvena tokom celog njenog postojanja.

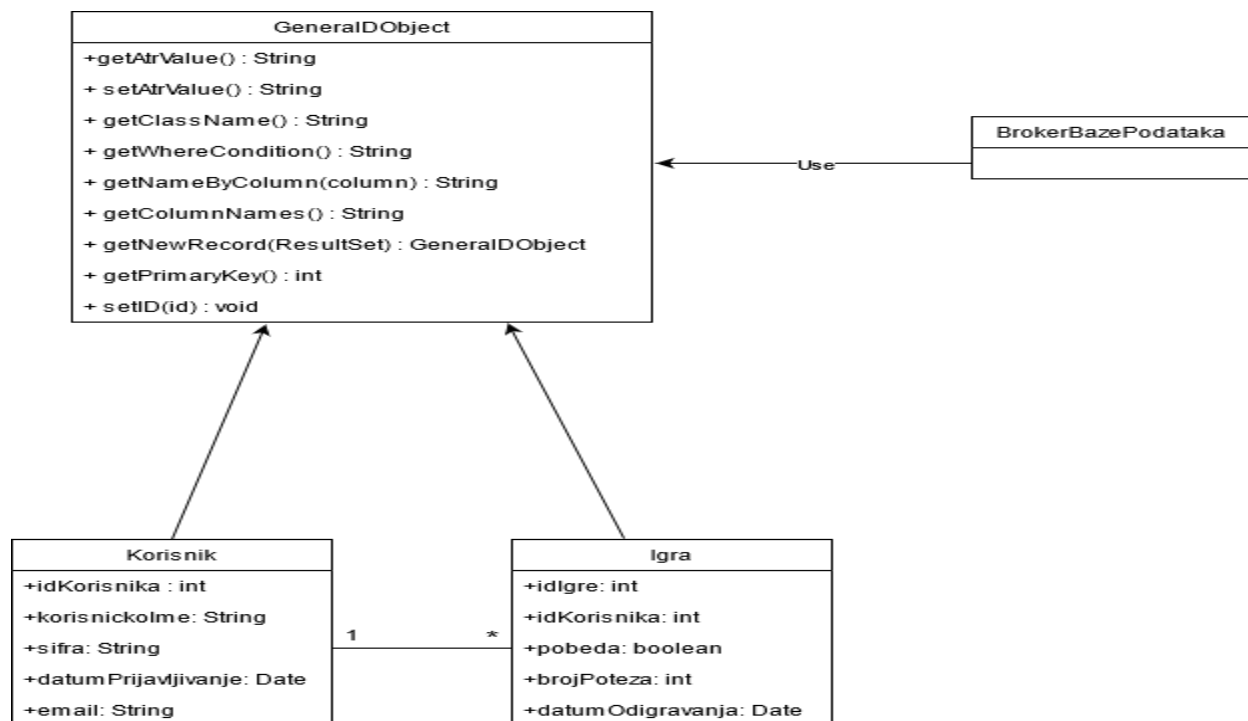
Za enkapsulaciju možemo reći da je takođe apstrakcija, jer razdvaja nešto što je opšte (javno) od specifičnog (privatno).

Takođe, primenom Larmanove metode u fazi analize opisujemo sistemske operacije na visokom nivou apstrakcije, odnosno enkapsuliramo način implementacije koji u tom momentu još uvek ne znamo, od onoga što u tom momentu znamo, a to je – šta ta sistemska operacija treba da radi.

### *Odvajanje interfejsa od implementacije*

Interfejs se odvaja od implementacije i izlaže se klijentu.

Klijent ne treba da zna kako su operacije implementirane. U primeru, klijent je klasa *DatabaseBroker*, a server je klasa *GeneralDObject*.



Slika 31 - Odvajanje interfejsa od implementacije

Broker je taj koji poziva metode klasa koje implementiraju klasu *GeneralDObject*, ne znajući kako su metode implementirane.

### *Dovoljnost, kompletnost i jednostavnost*

Dovoljnost, kompletnost i jednostavnost ukazuju na osobine softverske komponente koju karakteriše jednostavan način održavanja kao i nadgradnje, a sa druge strane je dovoljna i kompletna da obezbedi funkcionalnost.

## 7.2. Strategije projektovanja

Najpoznatije strategije projektovanja su:

1. Podeli i pobedi
2. S vrha na dole
3. Odozdo na gore
4. Iterativno-inkrementalni pristup

### *Podeli i pobedi – divide and conquer*

Ova strategija se zaniva na prethodno pomenutom principu dekompozicije koji razlaže polazni problem na potprobleme, koji se nezavisno rešavaju kako bi se polazni problem lakše rešio.

Ova strategija se najčešće primenjuje u prve tri faze razvoja softvera:

1. Prikupljanje korisničkih zahteva – zahtevi se opisuju preko skupa nezavisnih slučajeva korišćenja
2. Analiza – struktura se opisuje pomoću konceptualnog modela, a ponašanje preko skupa nezavisnih sistemskih operacija.
3. Projektovanje – arhitektura softverskog sistema se deli na tri dela: korisnički interfejs, aplikacionu logiku i skladište podataka. Korisnički interfejs se dalje deli na ekranske forme i kontroler korisničkog interfejsa, a aplikaciona logika na: kontroler aplikacione logike, poslovnu logiku i skladište podataka.

### *S vrha na dole – top down*

Ova strategija se zasniva na principu dekompozicije funkcija i deli početnu funkciju na više podfunkcija.

Ove podfunkcije se nezavisno rešavaju kako bi olakšale rešavanje polazne funkcije.

Primer koji je dat se odnosi na klasu *OpstelzvršenjeSO*:

```
public abstract class OpstelzvršenjeSO {

    static public BrokerBazePodataka bbp = new BrokerBazePodataka1();
    GeneralDObject gdo;

    synchronized public boolean opstelzvršenjeSO() {
        bbp.makeConnection();
        boolean signal = izvrsiSO();
        if (signal == true) {
            bbp.commitTransation();
        } else {
            bbp.rollbackTransation();
        }
        bbp.closeConnection();
        return signal;
    }
    abstract public boolean izvrsiSO();
}
```

### *Odozdo na gore*

Ova strategija se zasniva na principu generalizacije, koji u nekoj složenoj funkciji uočava jednu ili više logičkih celina, koje proglašava za funkcije.

Na taj način je složena funkcija dekomponovana na više nezavisnih funkcija.

Kompozicija tih funkcija treba da obezbedi istu funkcionalnost kao i složena funkcija.

U ovoj strategiji kao i u prethodnoj, dobićemo isto rešenje. Ono što se razlikuje je način dolaska do rešenja. U prethodnoj strategiji uočavamo odmah generalne delove, odnosno složene funkcije koje moramo razložiti, dok u ovakvom pristupu prvo polazimo od veoma specifičnih delova.

Nakon što uočimo logičke delove koji su specifični, možemo ih proglasiti za funkcije.

### *Iterativno-inkrementalni pristup*

U iterativno-inkrementalnom pristupu je cilj da se relativno brzo realizuje rešenje, koje nije neophodno da bude sasvim kompletno. Cilj je da korisnik može da vidi gotov proizvod, pojedine funkcionalnosti i da izrazi svoje zadovoljstvo odnosno nezadovoljstvo, što bi značilo da se ta funkcionalnost mora menjati.

Ovakav pristup je svakako bolja opcija od toga da se stigne do kraja projekta i da se kasno shvati da možda korisnički zahtevi nisu dobro prikupljeni ili shvaćeni, što bi rezultiralo nezadovoljstvom korisnika i velikim problemom u smislu izmene celog projekta.

Kako bi se problemi izbegli, sistem koji se razvija se deli na više delova (potprojekata), koji mogu predstavljati sistemske operacije koje se razvijaju ili potpuno nezavisne slučajeve korišćenja. Svaki deo sistema, podprojekat, prolazi kroz više iteracija i kao rezultat jedne iteracije, dobija se **inkrement** za sistem.

Na kraju se svi potprojekti integrišu u jedan softverski sistem.

Primer: Prikupljanjem korisničkih zahteva, uočili smo sistemske operacije koje treba projektovati tako da su potpuno nezavisne jedna od druge. Na ovakav način omogućavamo i pojavu novih sistemskih operacija koje neće uticati na postojeće, kao i izmenu postojećih, a da se izmene ne odraze na druge sistemske operacije.



### 7.3. Metode projektovanja

Najvažnije metode projektovanja su:

1. Funkciono orijentisano projektovanje
2. Objektno orijentisano projektovanje
3. Projektovanje zasnovano na strukturi podataka
4. Projektovanje zasnovano na komponentama

1. Funkciono orijentisano projektovanje:

Problem se posmatra iz perspektive njegovog ponašanja, funkcionalnosti.

Na ovaj način se prvo uočavaju funkcije sistema, zatim se određuju strukture podataka nad kojima se izvršavaju te funkcije.

2. Objektno orijentisano projektovanje:

Zasnovano je na objektima (klasama). Objekti mogu da predstavljaju i strukturu i ponašanje softverskog sistema. Kod objektno orijentisanog projektovanja paralelno se razvijaju i struktura i ponašanje.

Teži se razdvajanju strukture i ponašanja, jer se želi postići efekat nezavisnog izvršenja sistemskih operacija nad nezavisnom strukturom sistema.

3. Projektovanje zasnovano na strukturi podataka:

Problem posmatra iz perspektive strukture. Prvo se uočava struktura sistema, a zatim se definišu funkcije koje se izvršavaju nad tom strukturom.

4. Projektovanje zasnovano na komponentama:

Problem posmatra iz perspektive postojećih komponenti koje se mogu (ponovo) koristiti u rešavanju problema. Prvo se uočavaju delovi problema koji se mogu realizovati postojećim komponentama, a nakon toga se implementiraju oni delovi za koje nije postojalo rešenje.

## 7.4. Principi objektno orijentisanog projektovanja

Postoje sledeći principi kod objektno orijentisanog projektovanja klasa:

1. Princip otvoreno zatvoreno
2. princip zamene Barbare Liskov
3. princip inverzije zavisnosti
4. princip umetanja zavisnosti
5. princip izdvajanja interfejsa

### 1. **Princip otvoreno zatvoreno (Open – closed principle) :**

Modul treba da bude otvoren za proširenje, ali i zatvoren za modifikaciju.

Primer je klasa *OpstelzvršenjeSO*, koja kao što je već rečeno, ima *opstelzvršenjeSO* metodu, u kojoj daje redosled izvršenja operacija na višem nivou apstrakcije, a detalje izvršenja ostavlja podklasama.

Na ovaj način, redosled izvršenja operacija je nepromenljiv, a ono što je otvoreno za promene je apstraktna metoda koja dozvoljava podklasama da je prošire. Primetićemo da postoji direktna veza između Template Method patterna i open-closed principa, odnosno možemo reći da je template method pattern zasnovan na open-closed principu.

```
public abstract class OpstelzvršenjeSO {  
  
    static public BrokerBazePodataka bbp = new BrokerBazePodataka1();  
    GeneralDObject gdo;  
  
    synchronized public boolean opstelzvršenjeSO() {  
        bbp.makeConnection();  
        boolean signal = izvrsiSO();  
        if (signal == true) {  
            bbp.commitTransation();  
        } else {  
            bbp.rollbackTransation();  
        }  
        bbp.closeConnection();  
        return signal;  
    }  
  
    abstract public boolean izvrsiSO();  
}
```

## 2. ***Princip zamene Barbare Liskov (The Liskov substitution principle):***

Podklase treba da budu zamenljive sa njihovim nadklasama.

Primer se odnosi na domenske klase *Igra* i *Korisnik*, koje mogu biti zamenljive njihovom nadklasom *GeneralDObject*. Ovo se konkretno vidi u metodama brokera baze podataka koji kao parametar prima *GeneralDObject*.

Obezbeđivanjem ovakvog parametra, obezbeđujemo da nam klasa brokera baze podataka ne bude čvrsto vezana za sve domenske klase, već za njihovu apstrakciju, što je osnovna ideja svih paterna projektovanja.

### ***Broker baze podataka:***

```
public boolean insertRecord(GeneralDObject odo) {  
    String upit = "INSERT INTO " + odo.getClassName() + " VALUES (" + odo.getAtrValue()  
        + ")";  
    return executeUpdate(upit);  
}
```

### ***Sistemska operacija StartujSO:***

```
public class RegistrujSO extends OpstelzvrjenjeSO{
    private RequestDto request;
    private ResponseDto response;
    public ResponseDto registruj(RequestDto request){
        this.request = request;
        response = new ResponseDto();
        //set response code boolean
        this.opstelzvrjenjeSO();
        return this.response;
    }

    @Override
    public boolean izvrsiSO() {
        boolean uspeh = bbp.insertRecord(((GeneralDObject) this.request.getObjekat()));
        response.setDone(uspeh);

        if(uspeh){
            response.setObjekat(((Korisnik) bbp.findRecord(((GeneralDObject)
request.getObjekat())));
            return true;
        }
        else return false;
    }
}
```

### ***Kontroler igre:***

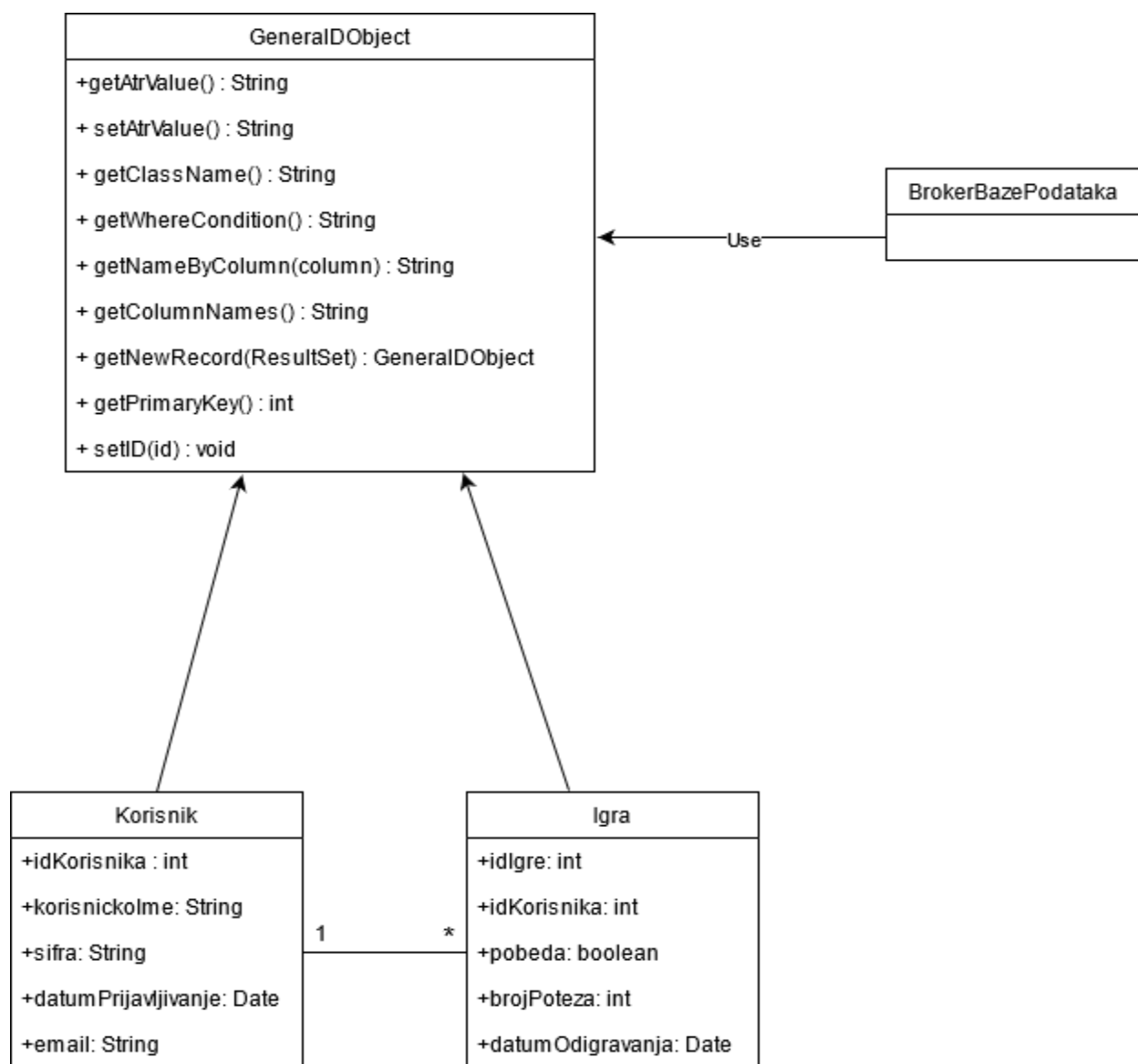
```
RequestDto request = new RequestDto();
trenutnaIgra.setIdKorisnika(MainMenuFX.korisnik.getIdKorisnika());
request.setIgra(trenutnaIgra);
request.setOperacija(Operacija.STARTUJ);

Communication.getInstance().sendRequest(request);
ResponseDto response = Communication.getInstance().readResponse();
```

### 3. *Princip inverzije zavisnosti – The Dependency inversion principle*

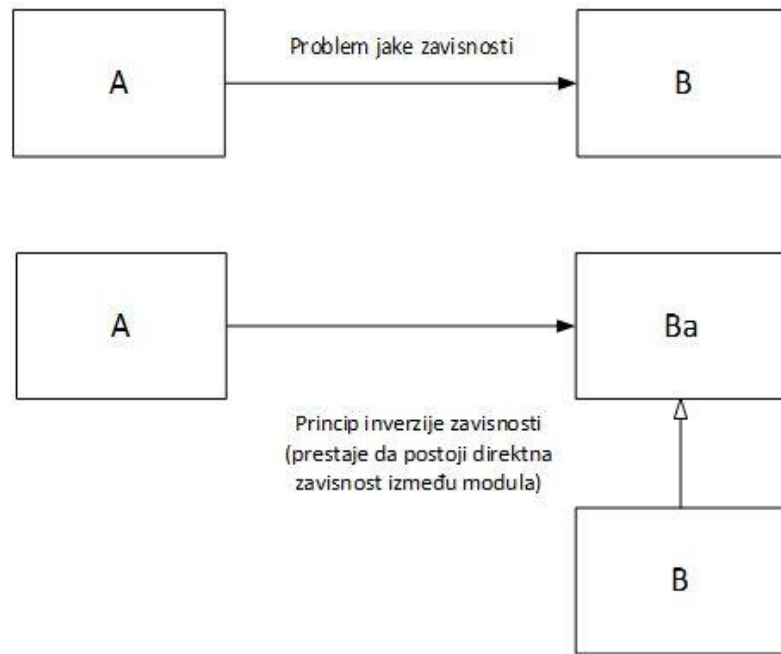
- Princip inverzije zavisnosti: zavisi od apstrakcije a ne od konkretizacije.
- Moduli višeg nivoa ne treba da zavise od modula nižeg nivoa
- Oba treba da zavise od apstrakcije
- Apstrakcije ne treba da zavise od detalja
- Detalji treba da zavise od apstrakcije

U našem primeru se izbegava zavisnost modula višeg nivoa: Brokera baze podataka od modula nižeg nivoa – domenskih klasa. Povezanost se ostvaruje preko klase *GeneralDObject*, koju realizuju sve domenske klase. Na taj način se broker baze podataka posredno povezuje sa svim domenskim klasama koje realizuju klasu *GeneralDObject*.



Slika 32 - Primer principa inverzije zavisnosti

U opštem slučaju:



Slika 33 - Princip inverzije zavisnosti - opsti slucaj

Modul A je modul višeg nivoa i on u prvom delu slike zavisi od modula nižeg nivoa – modula B, što je protivno ovom principu.

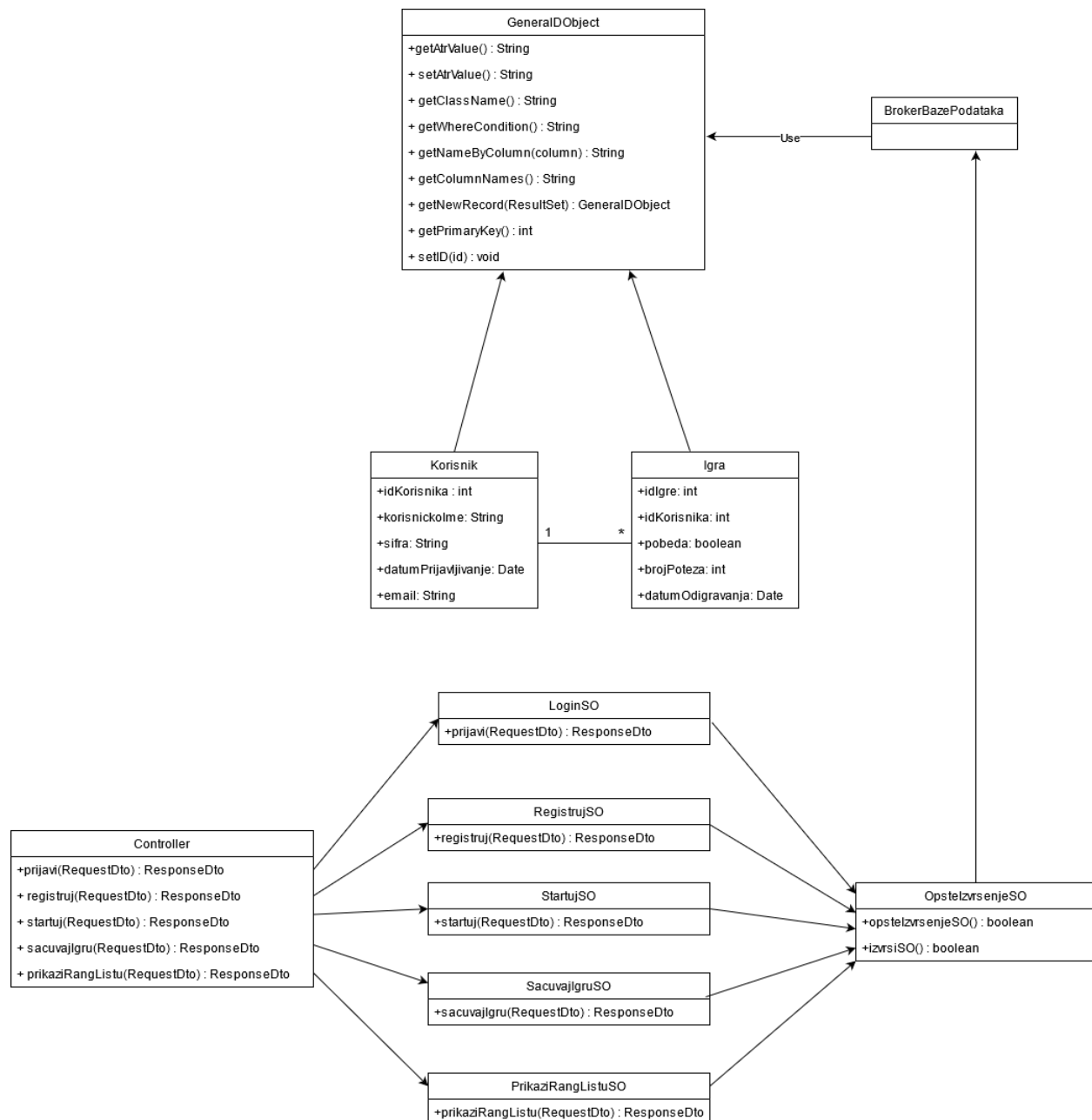
Direktna veza između modula A i B se prekida i između njih se postavlja modul višeg nivoa – modul Ba.

Možemo da zaključimo da postoji velika sličnost između principa inverzije zavisnosti i opšte definicije paterna.

#### 4. *Princip umetanja zavisnosti – The dependency injection principle :*

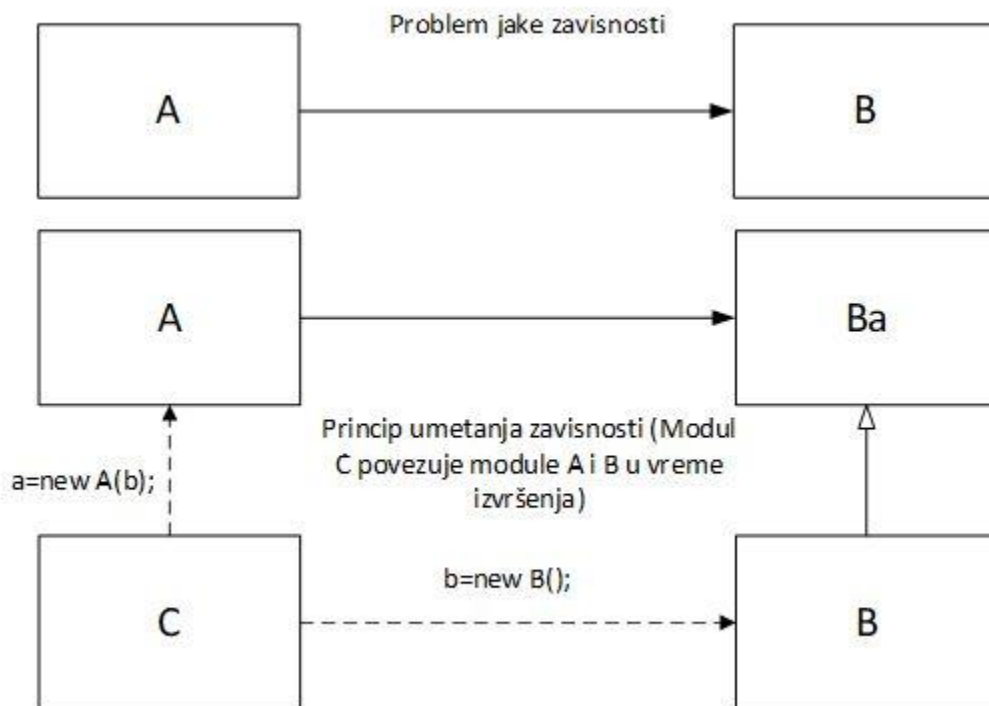
Zavisnosti između dve komponentne programa se uspostavljaju u vreme izvršenja programa preko neke treće komponente.

U navedenom primeru, veza između brokera baze podataka i konkretnih domenskih klasa se uspostavlja u vreme izvršenja programa, preko klasa koje realizuju klasu *OpstelzvršenjeSO*.



Slika 34 - Princip umetanja zavisnosti

U opštem slučaju:



Slika 35 - Opsti slucaj umetanja zavisnosti



## **8. Primena paterna u projektovanju**

### **8.1. Uvod u paterne**

Prve definicije paterna nastaju opažanjem struktura gradova i građevina, dakle u građevinarstvu, zaslugom Kristofera Aleksandera koji je i dao prvi značajan doprinos u definisanju paterna.

Jedna od njegovih definicija glasi: "Svaki patern je trodelno pravilo, koje uspostavlja relaciju između nekog problema, njegovog rešenja i njegovog konteksta.

Patern je u isto vreme i stvar, koja se dešava u stvarnosti, i pravilo koje govori kada i kako se kreira navedena stvar".

Ono što je Aleksander otkrio je da se za svaki problem koji se više puta ponavlja na različit način, postoji neko rešenje koje je primenljivo za ceo skup sličnih problema.

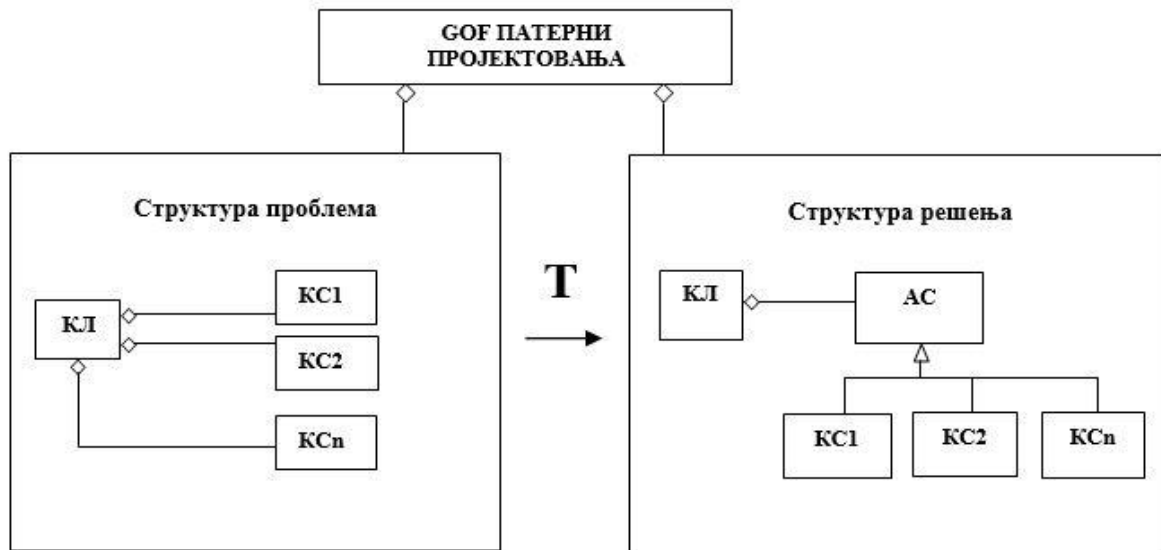
Ono što izvodimo kao zaključak je da je patern "trodelno pravilo" koje uspostavlja korelaciju između određenog konteksta kao sistema ograničenja, problema koji deluje u tom kontekstu i rešenja, kao strukture softverskog sistema koja omogućava da odnosi između elemenata tog sistema budu ponovo upotrebljivi.

### **8.2. Opšti oblik GOF paterna projektovanja**

Kada govori o prethodno navedenim definicijama paterna, Kristofer Aleksander insistira i na osobini ponovne upotrebljivosti paterna i kaže: "Svaki patern opisuje problem koji se javlja iznova (neprestano) u našem okruženju, a zatim opisuje suštinu rešenja tog problema na takav način da vi možete koristiti ovo rešenje milion puta a da nikada to ne uradite na dva puta na isti način "[2], što znači da se već jednom pronađeno rešenje može primeniti na više različitih problema, koji su iz istog konteksta.

Patern kao proces, transformacijom strukture problema u strukturu rešenja obezbeđuje dugoročnost, stabilnost, fleksibilnost i mogućnost daljeg razvoja, što i jeste primarna definicija održivosti, u kontekstu softverskih sistema.

Ovakvom transformacijom, razdvajaju se specifičnosti koje obezbeđuju različitosti u softverskom sistemu, i koje su na samom početku razvoja softverskog sistema bile pomešane sa generalnim delovima, koje programu obezbeđuju univerzalnost, što i jeste nešto čemu vremenom teži svaki softverski sistem.



Slika 36 - Opsti oblik GOF paterna projektovanja

Postoje paterni *makro* arhitekture i *mikro* arhitekture.

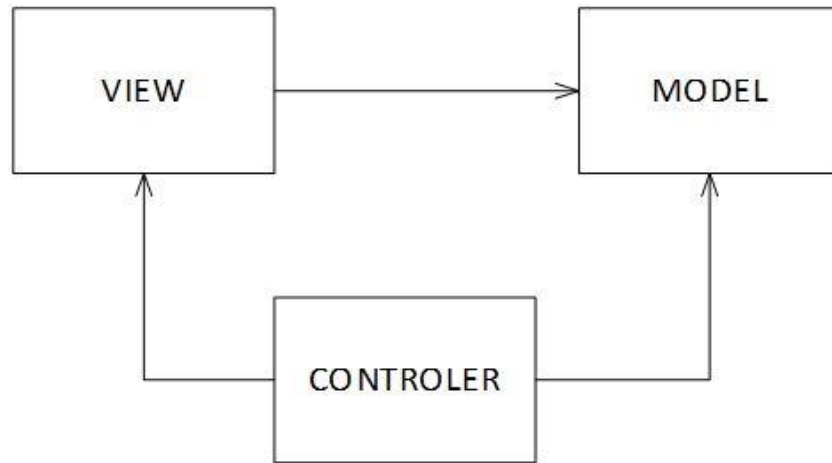
Makro arhitektura opisuje strukturu i organizaciju softverskog sistema na najvišem nivou.

*MVC* je makroarhitekturni patern koji deli softverski sistem na tri dela:

1. View – obezbeđuje korisniku interfejs (ekransku formu) pomoću koje će korisnik da unese podatke i poziva odgovarajuće operacije koje treba da se izvrše nad modelom
2. Controller – osluškuje i prihvata zahtev od klijenta za izvršenje operacije.

Nakon toga poziva operaciju koja je definisana u modelu. Ukoliko model promeni stanje, controller izveštava view o tome.

3. Model – predstavlja stanje sistema. Stanje modela menjaju neke od operacija modela.



Slika 37 - MVC patern

*Pogled (view)* je zadužen da omogući korisniku poziv operacija nad modelom, ima informacije o modelu, dok kontroler ima informaciju i o pogledu i o modelu. On izvršava promene nad modelom koje pogled zahteva i obaveštava pogled o tim promenama.

Model ne mora da ima bilo kakvu informaciju o kontroleru i pogledu, on samo služi da čuva informacije o svojim stanjima. Može se i reći da model predstavlja aplikacionu logiku, pogled ekransku formu, a kontroler predstavlja “lepak” između njih.

Ukoliko govorimo u kontekstu klijenta i servera, kod MVC paterna, model bi predstavljao server, dok su kontroler i pogled klijenti.

Mikro arhitekturni paterni se svrstavaju u tri kategorije:

- Kreacioni paterni
- Paterni strukture
- Paterni ponašanja

Kreacioni paterni apstrahuju proces kreiranja objekata. Oni daju veliku fleksibilnost u tome šta će biti kreirano, ko će to kreirati, kako i kada.

U ovu podvrstu paterna se svrstavaju: *Abstract Factory*, *Builder*, *Factory Method*, *Prototype*, *Singleton*.

Strukturni paterni opisuju složene strukture međusobno povezanih klasa i objekata. U njih se svrstavaju: *Adapter*, *Bridge*, *Composite*, *Decorator*, *Facade*, *Flyweght*, *Proxy* *patern*.

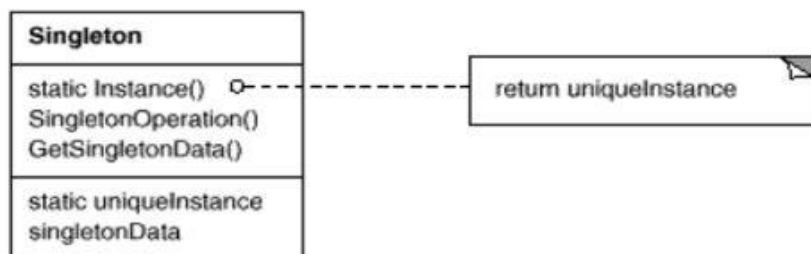
Paterni ponašanja opisuju način na koji klase i objekti sarađuju i raspoređuju odgovornosti. U njih se svrstavaju: *Chain of responsibility*, *Command*, *Interpreter*, *Iterator*, *Mediator*, *Memento*, *Observer*, *State*, *Strategy*, *Template method*, *Visitor* *patern*.

U nastavku će biti dati primeri primena nekih od navedenih paterna:

## Singleton patern

Zahtev: Potrebno je obezbediti jedinstvenu komunikaciju između klijenta i servera. Neophodno je da se klijent samo jednom poveže sa serverom, ostvarujući komunikaciju sa serverom putem klase *Kontroler*. Onemogućiti ponovno povezivanje u toku jednog pokretanja programa.

Rešenje: *Singleton* patern obezbeđuje klasi samo jedno pojavljivanje i jedinstven pristup do nje. To omogućava statička metoda *Instance()*.



Slika 38 - Singleton patern

U primeru je data implementacija klase *Kontroler*.

```

public class Kontroler {

    private static Kontroler instance;
    private OpsteIzvršenjeSO so;

    public static Kontroler getInstance() {
        if(instance == null){
            instance = new Kontroler();
        }
        return instance;
    }

    public ResponseDto startuj(RequestDto request) {
        so = new StartujSO();
        return ((StartujSO) so).startuj(request);
    }

    public ResponseDto sacuvaj(RequestDto request) {
        so = new SacuvajSO();
        return ((SacuvajSO) so).sacuvaj(request);
    }

    public ResponseDto rangLista(RequestDto request) {
        so = new RangListaSO();
        return ((RangListaSO) so).prikaziRangListu(request);
    }
}

```

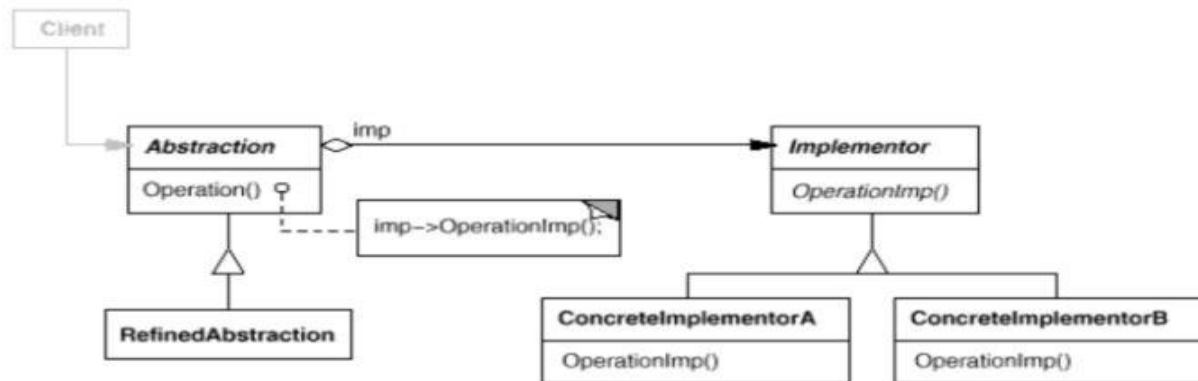
Slika 39 - Singleton patern – Kontroler

Ova klasa je implementirana preko Singleton paterna i na taj način je omogućeno da se pri svakom pozivu metode *getInstance()*, dobije jedna ista instanca ove klase, koja će biti kreirana pri prvom pozivu a pomoću koje će se vršiti pozivanje sistemskih operacija.

## Bridge patern

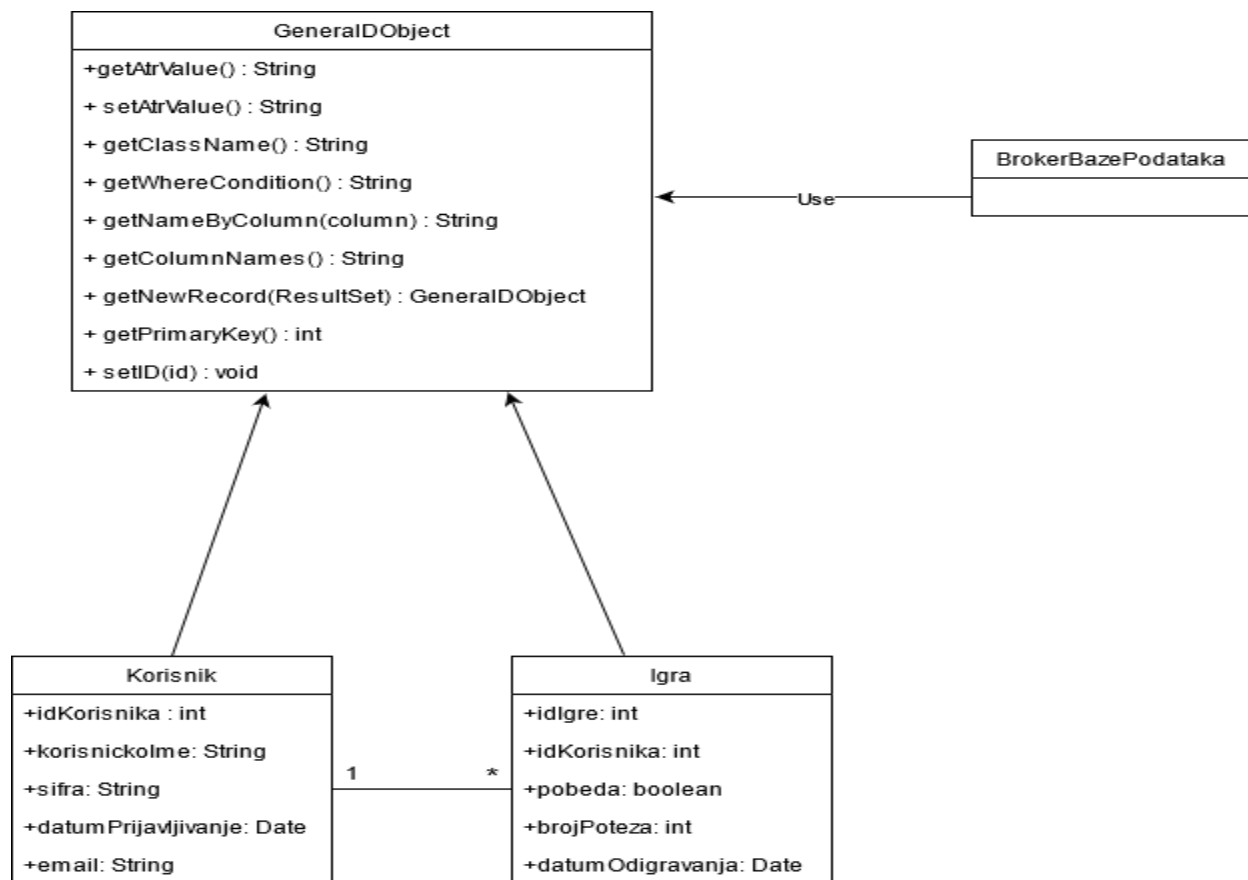
Zahtev: Potrebno je obezbediti apstraktne operacije u klasi brokera baze podataka, za koje će se tek u vreme izvršenja programa vezati konkretne implementacije. Obezbediti generičke metode za sve domenske klase.

Rešenje: Bridge patern - dekupluje (odvaja) apstrakciju od njene implementacije, tako da se one mogu menjati nezavisno.



Slika 40 - Bridge patern

U primeru navodimo klasu *DatabaseBroker*, koja kao parametar u svojim metodama prima generički objekte klase *GeneralIDObject*, nad kojima poziva određene metode pri izvršenju.



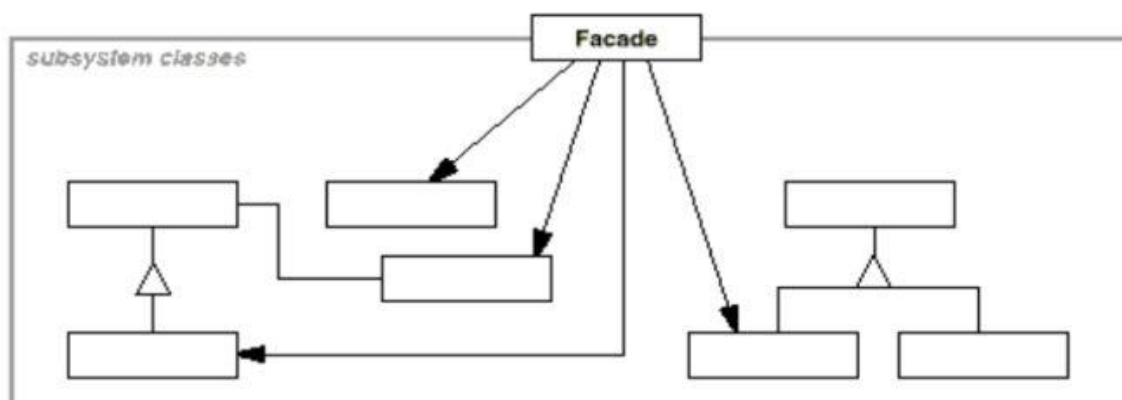
Slika 41 - Bridge patern - Broker baze podataka

Sa priloženih slika možemo zaključiti da je odnos između brokera baze podataka i klase *GeneralDObject*, zapravo odnos između *Abstraction*-a i *Implementora*, odnosno zaključujemo da broker baze podataka ima ulogu *Abstraction*-a, a domenske klase ulogu *ConcreteImplementora*-a.

## Facade patern

**Zahtev:** Potrebno je obezbediti korisnicima da pomoću ekranskih formi pozivaju različite operacije sistema. Server treba da osluškuje povezivanje klijenta i kreira posebnu klijentsku nit koja će nastaviti da osluškuje zahteve od povezanog klijenta. Na ovaj način je potrebno obezbediti da server samo usmerava zahteve ka klijentskoj niti, ne da ih obrađuje.

**Rešenje:** Facade patern – obezbeđuje jedinstven interfejs za skup interfejsa nekog podsistema. Facade patern definiše interfejs visokog nivoa koji omogućava da se sistem lakše koristi.



Slika 42 - Facade patern

Ovaj patern omogućava da se obezbedi jedinstvena tačka ulaska u neki podsistem i na taj način olakšava klijentu da ne mora voditi računa o onome što se dešava iza te tačke. Sve što je iza fasade, klijent ne treba da zna.

Primer koji je dat se odnosi na klasu *ServerStart*, koja predstavlja jedinstvenu tačku ulaza u sistem i u kojoj se delegiraju zahtevi ka odgovornim klasama za njihovu obradu, tj. klasi *Client*.



```

public class ServerStart extends Thread {

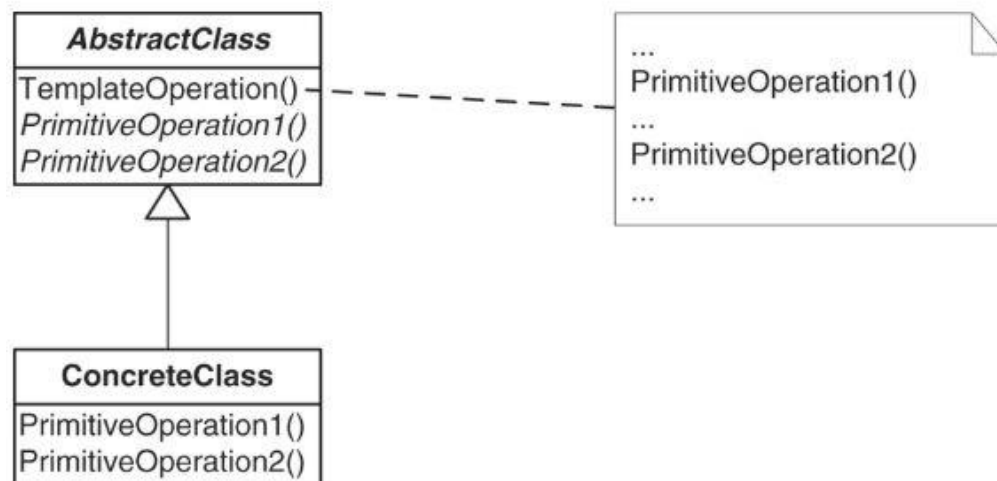
    private ServerSocket ss;
    private boolean active = true;
    @Override
    public void run() {
        try {
            ss = new ServerSocket(3724);
            System.out.println("Server je poceo sa radom.");
            while (active) {
                Socket clientSocket = ss.accept();
                System.out.println("Server je docekao novog klijenta.");
                KlijentNit klijentNit = new KlijentNit(clientSocket);
                klijentNit.start();
            }
        } catch (IOException ex) {
            Logger.getLogger(ServerStart.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}

```

## Template method patern

**Zahtev:** Potrebno je logički grupisati sve metode koje se ponavljaju pri izvršenju svake systemske operacije u jednu apstraktnu metodu i definisati redosled izvršenja. Neke metode je potrebno implementirati, jer je ponašanje isto za sve systemske operacije, a neke metode je potrebno ostaviti podklasama da implementiraju.

**Rešenje:** Template method patern- Definiše skelet algoritma u operaciji, prepuštajući izvršenje nekih koraka operacija podklasama. Ovaj patern omogućava podklasama da redefinišu neke od koraka algoritma bez promene algoritamske strukture.



Slika 43 - Template method patern

Na ovaj način, apstraktna klasa definiše neku template (šablon) operaciju koja sadrži druge primitivne operacije, koje se proglašavaju apstraktnim kako bi se omogućilo da ih podklase redefinišu. Primećujemo da na ovaj način obezbeđujemo da nijedna podklasa ne može promeniti redosled operacija – zatvoren je za promenu, ali može proširiti primitivne operacije - otvorene za proširenja, što je zapravo pravi pokazatelj direktne veze između open-closed principa i template method paternu.

U primeru je prikazana klasa *OpstelzvršenjeSO* i njena template metoda - *opstelzvršenjeSO()*:

```

public abstract class OpstelzvrjenjeSO {

    static public BrokerBazePodataka bbp = new BrokerBazePodataka1();
    GeneralDObject gdo;

    synchronized public boolean opstelzvrjenjeSO() {
        bbp.makeConnection();
        boolean signal = izvrsiSO();
        if (signal == true) {
            bbp.commitTransation();
        } else {
            bbp.rollbackTransation();
        }
        bbp.closeConnection();
        return signal;
    }

    abstract public boolean izvrsiSO();
}

```

Класе које представљају конкретне системске операцију редефинишу само методу `izvrsiSO()` – примитивну операцију.

```

public class LoginSO extends OpstelzvrjenjeSO {
    Request request;
    Response response;

    public Response login(Request request) {
        this.request = request;
        this.response = new Response();
        opstelzvrjenjeSO();
        return response;
    }

    @Override
    public boolean izvrsiSO() {
        System.out.println(request.getUser().getAtrValue());
        User user = (User) bbp.findRecord(request.getUser());
        response.setUser(user);
        return true;
    }
}

```

## **9. Literatura**

[1] dr Siniša Vlajić, Softverski proces (Skripta), Beograd, 2016

[2] Siniša Vlajić: Softverski paterni, Izdavač Zlatni presek, ISBN: 978-86-86887-30-6, Beograd, 2014.