**Kreacijski dizajn paterni**

Kreacijski dizajn paterni su paterni koji se bave procesom kreiranja objekata. Vrste kreacijskih paterna su: Factory, Builder, Prototype i Singleton. Uvođenjem svakih od navedenih paterna, postižemo niz olakšanja tokom implementacije našeg projekta, a to su:

* Apstrakcija procesa kreiranja objekata
* Povećana fleksibilnost i skalabilnost
* Centralizovana kontrola nad instaciranjem
* Smanjena zavisnost između klasa
* Pogodnije okruženje za testiranje i mocking
* Lazy loading
* Mogućnost kreiranja složenih struktura

U ovom dokumentu, pokušati ćemo objasniti svaki od kreacijskih paterna i gdje bi one mogle biti implementirane

**Factory pattern**

Factory pattern je kreacijski dizajn pattern koji omogućava kreiranje objekata bez direktnog korištenja new operatora, već se objekt kreira preko specijalizovane metode - tzv. **fabrike (factory method)**. Ova metoda odlučuje **koju konkretnu klasu** treba instancirati, često na osnovu ulaznih parametara.

Factory pattern ima:

* **superklasu ili interfejs** koji definiše zajedničke osobine objekata koji se kreiraju.
* **konkretne klase** koje implementiraju tu superklasu.
* Ima **Factory klasu** koja sadrži metodu za kreiranje konkretnih objekata **na osnovu nekog kriterija** (npr. string, enum, tip korisnika…).

Factory pattern se može primjeniti kada imamo različite tipova korisnika poput:

* Gost
* Korisnik (registrovani korisnik, recenzent)
* Artist
* Administrator

Gdje svaki od ovih korisnika ima različite privilegije i funkcionalnosti

Bez factory paterna, morali bismo imati nešto poput:

User user = new Artist();

Ali, ako se doda još jedan novi tip korisnika, moramo mijenjati sve dijelove koda gdje se objekti instanciraju.

RJEŠENJE:

//Interfejs

public abstract class User {

public abstract void prikaziOpcije();

}

//Konkretne klase

public class Guest extends User {

public void prikaziOpcije() {

System.out.println("Pregled albuma bez recenzija.");

}

}

public class RegisteredUser extends User {

public void prikaziOpcije() {

System.out.println("Može ostaviti recenziju.");

}

}

public class Artist extends User {

public void prikaziOpcije() {

System.out.println("Može dodati albume i vidjeti analitiku.");

}

}

public class Admin extends User {

public void prikaziOpcije() {

System.out.println("Administrativni pristup.");

}

}

//Factory klasa

public class UserFactory {

public static User kreirajKorisnika(String tip) {

switch (tip.toLowerCase()) {

case "gost":

return new Guest();

case "korisnik":

return new RegisteredUser();

case "artist":

return new Artist();

case "admin":

return new Admin();

default:

throw new IllegalArgumentException("Nepoznat tip korisnika");

}

}

}

//Upotreba

User u = UserFactory.kreirajKorisnika("artist");

u.prikaziOpcije(); // Output: Može dodati albume i vidjeti analitiku.

**Builder Pattern**

Builder pattern je koristan u situacijama kada:

* Kreiramo **objekte sa mnogo opcionalnih ili složenih parametara**.
* Kreiramo **različite “verzije” istog objekta**, bez dupliranja koda.

U Revalb sistemu postoje korisnički nalozi koji mogu imati mnogo opcionalnih podataka poput: slika profila, biografija, ime, prezime, itd.

*Problem bez buildera:* Konstruktor sa 10+ parametara ili mnoštvo setter-a:

User user = new User("artist", "Eldar", null, "biografija", null, ...);

*Sa builderom:*

User artist = new UserBuilder("Eldar", "artist")

.withProfilePicture("eldar.png")

.withBio("Sarajevski hip-hop artist")

.withStats(102, 35)

.build();

Tokom kreiranja objekta albuma, album može imati sljedeće podatke: Naslov, žanr, datum izlaska, trajanje, lista pjesama, slika omota, itd  
  
*Builder bi omogućio:* dodavanje samo potrebnih polja i fleksibilnu gradnju:

Album album = new AlbumBuilder("Neon Nights")

.withGenre("Synthpop")

.withCover("neon.jpg")

.withTracks(listaPjesama)

.build();

**Prototype pattern**

**Prototype** omogućava **kreiranje novih objekata kloniranjem već postojećih objekata**, umjesto kreiranja "od nule". Koristi se kada:

* Kreiranje objekta je **skupo** (vremenski ili resursno).
* Imaš objekat koji se često koriste kao **predložak** (template).
* Želimo brzu **kopiju sa izmjenama**, bez ponavljanja logike izgradnje.

**Kod implementacije Prototype patterna:**

public class Album implements Cloneable {

private String title;

private String genre;

private List<Track> tracks;

public Album clone() {

try {

Album copy = (Album) super.clone();

copy.tracks = new ArrayList<>(this.tracks); // deep copy

return copy;

} catch (CloneNotSupportedException e) {

throw new RuntimeException(e);

}

}

}

U našem Revalb sistemu, Prototype pattern se može koristiti za Kloniranje albuma kao šablon, na primjer:  
Artist želi:

* Kreirati novi album **na osnovu već postojećeg**, ali s drugačijim imenom, pjesmama ili cover-om.
* Npr. "Deluxe Edition" albuma koji je skoro isti kao original.

Album original = new Album("Electric Soul", "Jazz", cover, trackList);

Album deluxe = original.clone();

deluxe.setTitle("Electric Soul – Deluxe");

**Singleton pattern**

**Singleton** osigurava da postoji **samo jedna instanca određene klase** u cijelom sistemu i omogućava globalni pristup toj instanci.

* Privatni konstruktor
* Statička metoda za dobijanje instance (getInstance())
* Jedna jedina instanca u memoriji

U našem Revalb sistemu, može se koristiti singleton pattern u sljedećoj situaciji:

### **Database konekcija (ako koristite JDBC ili sličan pristup)**

Samo jedna veza prema bazi, dijeljena kroz cijelu aplikaciju.

Connection conn = DatabaseConnection.getInstance().getConnection();