SOLID PRINCIPLES

Single-Responsibility Principle (SRP)

Single-Responsibility Principle (SRP) je prvi SOLID princip koji navodi da bi svaka klasa trebala imati samo jednu odgovornost. To znači da bi svaka klasa trebala biti zadužena za samo jedan dio funkcionalnosti i ta odgovornost treba biti potpuno enkapsulirana u klasi.

Na osnovu našeg dijagrama klasa, SRP princip se primjenjuje na sljedeći način:

- **Apstraktna klasa 'Korisnik'**: Klasa 'Korisnik' čuva osnovne podatke o korisniku, dok su izvedene klase 'Donor', 'Zaposlenik', i 'Administrator' zadužene za čuvanje podataka specifičnih za svaku od ovih uloga.

- **Klase 'Zahtjev', 'Termin', i 'Zaliha'**: Svaka od ovih klasa ima tačno jednu odgovornost. Klasa 'Zahtjev' je zadužena za upravljanje zahtjevima, klasa 'Termin' za upravljanje terminima, a klasa 'Zaliha' za upravljanje zalihama.

Svaka klasa u dijagramu ima svoj skup odgovornosti bez preklapanja funkcionalnosti, što ukazuje na to da se Single-Responsibility Principle poštuje u cjelokupnom dijagramu klasa.

Open-Closed Principle (ORP)

Open-Closed Principle (OCP) je drugi SOLID princip i on navodi da bi "softverski entiteti (klase, module, funkcije, itd.) trebali biti otvoreni za proširenje, ali zatvoreni za modifikaciju". Ovo znači da bi trebalo biti moguće dodati nove funkcionalnosti bez mijenjanja postojećeg koda.

Na osnovu našeg dijagrama klasa, OCP princip se primjenjuje na sljedeći način:

- **Apstraktna klasa 'Korisnik'**: Ova klasa je "zatvorena za modifikaciju" jer definira osnovne atribute kao što su 'korisnickoIme', 'email', i 'lozinka' koje su zajedničke za sve korisnike. Međutim, ona je "otvorena za proširenje" jer omogućava izvođenje novih klasa ('Donor', 'Zaposlenik', 'Administrator') koje mogu dodati specifične atribute i metode relevantne za svoje uloge.

- **Klase 'Zahtjev', 'Termin', i 'Zaliha'**: Svaka od ovih klasa je zatvorena za modifikaciju jer imaju jasno definisane odgovornosti. Međutim, one su otvorene za proširenje jer se mogu dodati nove metode ili atributi koji proširuju njihove funkcionalnosti bez mijenjanja postojećeg koda.

Dakle, na osnovu dijagrama klasa, vrijedi Open-Closed Principle.

Liskov Substitution Principle (LSP)

Liskov Substitution Principle (LSP) je treći SOLID princip koji navodi da bi podklase trebale biti zamjenjive za svoje nadklase bez utjecaja na ispravnost programa. To znači da bi svaka klasa koja koristi određenu nadklasu trebala moći koristiti bilo koju od njenih podklasa bez da to utječe na ispravnost programa.

Na osnovu našeg dijagrama klasa, LSP princip se primjenjuje na sljedeći način:

- **Apstraktna klasa 'Korisnik'**: Klasa 'Korisnik' je nadklasa za klase 'Donor', 'Zaposlenik', i 'Administrator'. Svaka od ovih podklasa može biti zamijenjena klasom 'Korisnik' bez utjecaja na ispravnost programa. Na primjer, ako postoji funkcija koja koristi klasu 'Korisnik', ta funkcija bi trebala raditi ispravno bez obzira koristi li se instanca klase 'Donor', 'Zaposlenik', ili 'Administrator'.

Primjena Liskov-og Substitution Principle-a osigurava da dizajn softvera ostane čist, sa jasno definiranim odnosima između klasa. Zaključujemo da za naš dijagram klasa vrijedi Liskov Substitution Principle.

Interface Segregation Principle (ISP)

Interface Segregation Principle (ISP) je četvrti SOLID princip koji navodi da klijenti ne bi trebali biti prisiljeni ovisiti o interfejsima koje ne koriste. Ovaj princip promoviše upotrebu više specifičnih interfejsa umjesto jednog općeg interfejsa.

Na onovu našeg dijagrama klasa, ISP princip se primjenjuje na sljedeći način:

- **Interfejs 'Korisnik'**: Umjesto da imamo jedan opći interfejs 'Korisnik' koji koriste sve klase ('Donor', 'Zaposlenik', 'Administrator'), možemo imati više specifičnih interfejsa. Na primjer, možemo imati interfejs 'IDonor' za klasu 'Donor', interfejs 'IZaposlenik' za klasu 'Zaposlenik', i interfejs 'IAdministrator' za klasu 'Administrator'. Svaki interfejs bi imao metode koje su specifične za datu klasu. Trenutno nije implementiran niti jedan interfejs.

Primjena Interface Segregation Principle-a pomaže u održavanju čistog dizajna softvera, smanjuje zavisnosti između klasa i olakšava održavanje koda. Na osnovu navedenog objašnjenja, Interface Segregation Principle bi vrijedio za naš konkretni projekat.

Dependency Inversion Principle (DIP)

Dependency Inversion Principle (DIP) je peti SOLID princip koji predstavlja specifičnu metodologiju za slabo povezane softverske klase. Kada se ovaj princip primjenjuje, konvencionalni odnosi zavisnosti između visokih klasa koji postavljaju politike i niskih klasa na kojima ovi prvi zavise se preokreću, čime se visoke klase čine nezavisnim od detalja implementacije niskih klasa.

Princip navodi:

- Visoke klase ne bi trebale zavisiti o niskim klasama. Oba bi trebala zavisiti o apstrakcijama (npr. interfejsima).
- Apstrakcije ne bi trebale zavisiti o detaljima. Detalji (konkretne implementacije) bi trebali zavisiti o apstrakcijama.

Na onovu našeg dijagrama klasa, DIP princip se primjenjuje na sljedeći način:

- **Apstraktna klasa 'Korisnik'**: Kako su klase 'Donor', 'Zaposlenik' i 'Administrator' izvedene iz apstraktne klase 'Korisnik', navedene klase zapravo zavise od apstrakcije.

Kao i kod ISP-a, primjena Dependency Inversion Principle-a pomaže u održavanju čistog dizajna softvera, smanjuje zavisnosti između klasa i olakšava održavanje koda. Dakle, za naš dijagram klasa vrijedi Dependency Inversion Principle.