# Strukturalni paterni

## 1. Uvod

Naš postojeći sistem obuhvata entitete poput korisnika, termina, medicinskih usluga i nalaza, ali kako bismo postigli veću fleksibilnost i ponovnu iskoristivost koda, poboljšali modularnost, smanjili međuzavisnost klasa, uvodimo dva strukturalna paterna:

- Facade - koristi se kada sistem ima više podsistema (subsystems) pri čemu su apstrakcije i implementacije podsistema usko povezane
- Decorator - omogućava dinamičko dodavanje novih elemenata i ponašanja (funkcionalnosti) postojećim objektima

Pored ova 2 navedena postoji i niz ostalih paterna, koje ćemo također opisati u nastavku.

- Adapter – omogućava komunikaciju između nekompatibilnih interfejsa
- Bridge – razdvaja apstrakciju od implementacije radi veće fleksibilnosti
- Proxy – dodaje sigurnosnu ili kontrolnu logiku prije pristupa stvarnim objektima
- Composite – omogućava tretiranje pojedinačnih i grupnih objekata na isti način
- Flyweight – smanjuje upotrebu memorije dijeljenjem zajedničkih podataka

## 2. Potreba za strukturalnim paternima

Dizajn softverskih sistema u domenu zdravstva često podrazumijeva veliki broj povezanih klasa koje zajedno implementiraju poslovne procese kao što su zakazivanje termina, generisanje nalaza, upravljanje korisnicima i uslugama. S obzirom na to da zdravstveni informacioni sistem mora biti modularan, fleksibilan i lako proširiv, uočena su dva ključna nedostatka trenutnog rješenja:

### a) Problem kompleksnosti

U našem postojećem sistemu, klase kao što su Termin, MedicinskeUsluge, Korisnik i druge su direktno povezane i zavise jedna od druge.

Na primjer:

Klasa Termin direktno koristi atribute i metode iz Korisnik, Lokacija i MedicinskeUsluge.

Za zakazivanje jednog termina klijent mora upravljati nizom klasa i znati tačan redoslijed interakcija.

Ovakav dizajn uzrokuje:

- Teško održavanje i testiranje sistema.

- Veliku šansu za greške prilikom izmjena.
- Visoku međuzavisnost između klasa.

Rješenje – **Facade patern**:
Uvođenjem Facade paterna kreira se jedinstveni interfejs (ZakazivanjeFacade) koji predstavlja složenu logiku zakazivanja termina. Klijenti sistema sada koriste samo zakaziTermin(...) metodu bez potrebe za razumijevanjem svih unutrašnjih veza i logike. To pojednostavljuje korištenje i povećava modularnost.

## b) Problem ponovne upotrebe i proširivosti

U stvarnim uslovima rada, često se javlja potreba da se medicinskom nalazu dodaju nove funkcionalnosti kao što su:

Potpis doktora

Komentar doktora

Prevod nalaza na drugi jezik

Tradicionalni pristup bi zahtijevao izmjene unutar HistorijaGenerisanihNalaza klase ili stvaranje brojnih podklasa za svaku novu funkcionalnost.

Ovo dovodi do:

- Širenja hijerarhije klasa
- Dupliranja koda

Rješenje – **Decorator patern**:
Implementacijom Decorator paterna uvodi se interfejs NalazComponent koji definiše zajedničku metodu generisi(). Zatim se koristi osnovna klasa OsnovniNalaz, a dodatne funkcionalnosti (potpis, komentar) se dodaju kroz dekoratore: NalazSaPotpisom, NalazSaKomentarom.

Ovi dekoratori se mogu kombinovati po potrebi bez promjene postojećeg koda. Time se omogućava:

- Proširivost bez izmjena postojećih klasa
- Dinamičko dodavanje funkcionalnosti
- Fleksibilno sastavljanje ponašanja nalaza

## c) Problem kompatibilnosti eksternih i internih komponenti

Sistem se mora povezivati sa vanjskim servisima (npr. autentifikacija preko OAuth-a), ali trenutne klase nisu kompatibilne sa njihovim interfejsima.

Rješenje – **Adapter Pattern:**
Adapter omogućava konverziju interfejsa vanjskog servisa u oblik koji sistem već podržava, bez izmjena postojećeg koda.

Problem: `Korisnik` klasa koristi sopstvenu autentifikaciju (email + lozinka), ali želimo dodati i prijavu putem Google OAuth-a.
Dolazimo do toga da eksterni OAuth servis nema isti interfejs kao `Korisnik`.
Kreira se `AuthAdapter` koji implementira interfejs koji očekuje `Korisnik`, a interno koristi OAuth servis.

Primjer:

- Nova klasa: `AuthAdapter`
- Vezuje eksterni servis npr. `GoogleOAuthService` sa sistemskom klasom `Korisnik`

## d) Problem preklapanja apstrakcije i implementacije

Kod prikaza nalaza u različitim formatima (PDF, HTML) postoji velika zavisnost između prikaza i sadržaja.

Rješenje – **Bridge Pattern:**
Razdvajanjem apstrakcije (nalaz) i konkretne implementacije (format prikaza), omogućava se veća fleksibilnost i smanjuje zavisnost. Omogućava prikaz istog nalaza u više formata bez mijenjanja osnovne logike klase `Nalaz`.

## e) Problem kontrole pristupa osjetljivim podacima

Nisu svi korisnici ovlašteni da vide ili mijenjaju sve podatke (npr. nalaze). Trenutno ne postoji mehanizam kontrole pristupa.

Rješenje – **Proxy Pattern:**
Proxy klasa može kontrolisati pristup stvarnom objektu i provjeriti prava korisnika prije nego što se omogući pristup. Uvede se NalazProxy koji kontroliše da li korisnik ima pravo da vidi konkretan Nalaz.

## f) Problem grupisanja i upravljanja složenim strukturama

Neke medicinske usluge mogu biti dio paketa (npr. "sistematski pregled"), što zahtijeva da se više objekata tretira kao jedan.

Rješenje – **Composite Pattern:**
Omogućava tretiranje pojedinačnih i složenih objekata (paketa) na isti način. Ukoliko želimo kreirati pakete usluga (npr. “Sistematski pregled” = ultrazvuk + EKG + krvna slika). Uvedemo UslugaComponent interfejs. PojedinacnaUsluga i PaketUsluga implementiraju isti interfejs. Na primjer PaketUsluga sadrži listu više MedicinskaUsluga instanci

### g) Problem višestrukog korištenja identičnih podataka

Podaci kao što su lokacije, odjeli i vrste usluga često se ponavljaju kroz sistem i zauzimaju puno memorije.

Rješenje – **Flyweight Pattern:**
Dijeljenjem zajedničkih instanci, ovaj patern smanjuje memorijsku potrošnju. Na primjer `Lokacija` se ponavlja u svakom `Terminu`, a sistem može imati mnogo termina dnevno. Zbog toga se uvede `LokacijaFactory` koja vraća istu instancu `Lokacija` za identične adrese.

## 3. Primjena paterna

### 3.1 Facade Pattern

**- Lokacija primjene:**

Kombinacija više podsistema za zakazivanje termina (Korisnik, Termin, MedicinskeUsluge, Lokacija) u jedinstven interfejs: ZakazivanjeFacade.

**- Klase:**

ZakazivanjeFacade: pruža jednostavne metode poput zakaziTermin() i otkaziTermin(). Interno koristi klase Termin, MedicinskeUsluge, Korisnik, Lokacija.

**- Prednosti:**

Klijent ne mora poznavati kompleksnost zakazivanja termina.

Pojednostavljuje interakciju sa sistemom.

### 3.2 Decorator Pattern

**- Lokacija primjene:**

Dodavanje dodatnih funkcionalnosti rezultatima nalaza (HistorijaGenerisanihNalaza)

npr. dodavanje digitalnog potpisa, dodatnih komentara itd.

**- Klase:**

NalazComponent (interfejs)

OsnovniNalaz (implementira NalazComponent)

NalazSaPotpisom, NalazSaKomentarom

- **Prednosti:**

Fleksibilno dodavanje novih ponašanja bez mijenjanja postojeće logike.

Poštuje Open/Closed princip.

### 3.3 Adapter Pattern

**Lokacija primjene:** Povezivanje sistema sa eksternim autentifikacionim servisima (npr. OAuth)
**Klase**: AuthAdapter, EksterniAuthServis
Prednosti: Omogućava spajanje nekompatibilnih interfejsa bez promjene postojećeg koda

### 3.4 Bridge Pattern

**Lokacija primjene**: Razdvajanje prikaza nalaza od konkretnih formata (PDF, HTML, itd.)
**Klase:** NalazPrikaz, FormatImplementor, PDFFormat, HTMLFormat
Prednosti: Omogućava nezavisni razvoj prikaza i formata, veću fleksibilnost

### 3.5 Proxy Pattern

**Lokacija primjene:** Kontrola pristupa nalazima i osjetljivim podacima korisnika
**Klase:** NalazProxy, PraviNalaz
Prednosti: Dodaje sigurnosni sloj bez potrebe za izmjenom osnovne klase

### 3.6 Composite Pattern

**Lokacija primjene:** Upravljanje paketima medicinskih usluga (npr. “Pregled srca” sadrži više pojedinačnih usluga)
**Klase:** UslugaComponent, PojedinacnaUsluga, PaketUsluga
Prednosti: Tretiranje pojedinačnih i grupnih objekata na isti način

### 3.7 Flyweight Pattern

**Lokacija primjene**: Optimizacija memorijske potrošnje za često korištene objekte kao što su Lokacija, Odjel
**Klase:** FlyweightFactory, Flyweight, Kontekst
Prednosti: Smanjuje memorijsku potrošnju dijeljenjem zajedničkih instanci

## 4. Zaključak

Primjena strukturalnih paterna omogućava:

- Redukciju kompleksnosti i bolju organizaciju sistema (Facade)
- Modularno i fleksibilno proširivanje funkcionalnosti (Decorator)
- Sigurniji i integrabilniji sistem (Proxy, Adapter)
- Skalabilnost i optimizaciju resursa (Composite, Flyweight, Bridge)

Ovakav pristup povećava kvalitet dizajna, smanjuje tehnički dug i čini sistem spremnim za buduće zahtjeve bez potrebe za velikim izmjenama postojećeg koda.

Uvođenjem Facade i Decorator paterna sistem postaje modularniji i lakši za održavanje. Ovo je posebno važno za našu web stranicu zdravstva gdje su česte izmjene zahtjeva i funkcionalnosti. Na ovaj način krajnji korisnici (doktori, administratori i pacijenti) ostvaruju brži, sigurniji i intuitivniji pristup funkcionalnostima sistema.