PATERNI

Strukturalni paterni

- **Adapter pattern**

  Trenutno u našem sistemu nismo uočili mjesto za implementaciju ovog paterna. Međutim nakon razmatranja rada našeg sistema, zaključili bismo da bi se s proširenjem našeg sistema, javile i prilike za implementaciju ovog paterna. Eventualno mjesto za korištenje **adapter** paterna, bi bilo ako bismo proširili naš sistem, da blagajna ima mogućnost naplate preko kartice. Na ovaj način, dovoljno bi bilo da student priloži karticu, nakon čega bi se "skinula" sredstva sa kartice studenta. Poslije toga, sačekao bi se neki vid potvrde da je transakcija uspješna, te bi se onda pozvala standardno metoda *uplatiDomZaOdabraniMjesec*, koja bi izvršila ažuriranje stanja bonova za studenta.

- **Fasadni pattern**

  Zbog ne baš izražene kompleksnosti našeg sistema, nemamo trenutno mogućnosti da implementujemo **fasadni** patern. Naime, za korišenje ovog paterna bi nam trebalo više objekata koji rade kao jedan, dok u našem sistemu nije lako uočiti jednu takvu cjelinu. Ukoliko bismo se odlučili kasnije implementovati **fasadni** patern, moguće mjesto bi bilo prilikom proširenja kompleksnosti npr. Restorana. Proširivanjem kompleksnosti rada Restorana, došlo bi do potrebe za dodavanjem novih klasa, npr. neki tipovi klasa za jela, složenijeg sistema namirnica, pored dnevnog menija dodati cjenovnik itd. Proširivanjem sistema sa navedenim klasama, došlo bi do potrebe za nekim jednostavnijim načinom korištenja samog rada Restorana. Međutim taj problem bi bio riješen, upravo implementacijom **fasadnog** patern. Na ovaj način, Restoran bi koristio pozive nekih jednostavnih metoda, tipa *dajCjenovnik()*, dok bi se u pozadini odvijao složeniji postupak pravljenja tog cjenovnika, spajanja svih stvari u cjelinu, a da Restoran ne bi toga ni bio svjestan.

- **Dekoracijski pattern**

  U radu našeg sistema, trenutno ne uočavamo mogućnost implementacije ovog paterna. Nakon razmatranja sistema, primijetili

smo mjesto gdje bismo kasnije mogli primijeniti ovaj patern, a to je dio oko klase DnevnogMenija. Pošto se mi trenutno nismo toliko fokusirali na sam koncept Restorana u pozadini, njegovom nadogradnjom bismo mogli primijeniti i ovaj patern. Primjer upotrebe bi bio, ako bi nam se DnevniMeni sastojao od nekih cjelina, a ne samo od liste ručaka/večera, odnosno ako bi nam se meni sastojao od Rucka, Vecere, Napitaka i sl. Ovako bismo mogli korištenjem **dekoracijskog** paterna, na već postojeći DnevniMeni dodati npr. desert, ukinuti predjelo i sl. Ovo bi čak došlo do izražaja ako bi Restoran radio u nekom posebnom režimu rada, npr. tokom Ramazana, praznika i sl.

- **Bridge pattern**

  Gledajući naš sistem, mogućnost upotrebe ovog paterna bi bila ako bismo proširili naš sistem, da unutar Uprave/Restorana imamo različite uposlenike. Tako bismo npr. unutar StudentskogDoma, možda čak i Uprave/Restorana čuvali listu uposlenika. Međutim, uposlenici bi imali različite pozicije, npr. glavni kuhar, kuhar, čistačica itd. Kao i u stvarnom životu, postojale bi neke metode koje su zajedničke za sve uposlenike, međutim razlikovale bi se od implementacije do implementacije za svakog uspolenika. Jedan od primjera bi bio, da se izda platna lista za sve uposlenike, pozivala bi se ista metoda za izračun plate, međutim ona bi se razlikovala u zavisnosti od toga o kojem se uposleniku radi.

- **Proxy pattern**

  Uočili smo potrebu za implementacijom ovog paterna, prilikom posmatranja dijela sistema, vezanog za dodavanje studenata u studentski dom i općenito pristup metodama koje se koriste kod klase StudentskiDom. Ideja je u tome, da na neki način osiguramo da nije moguće da npr. korisnik Restoran koristi neke od metoda, za koje on nema ovlašten pristup, kao npr. metodu upisiStudenta, te se idealno rješenje ovog problema, upravo ogleda u implementaciji **Proxy** paterna. Prvobitna zamisao prije same implementacije ovog paterna je da se kao atribut čuva password korisnika. Zatim u slučaju da korisnik želi da pozove neku metodu, prvo bi se provjerilo da li on ima pristup toj metodi, tj. provjerili bismo da li se radi o Upravi/Restoranu/Studentu te onda odlučili da li korisnik ima pravo korištenja te metode. Na ovaj način bismo riješili eventualne sigurnosne propuste, koji bi se javili u našem sistemu.

- **Composite pattern**

  Posmatrajući naš sistem, prvobitno smo uočili mogućnost implementacije ovog patterna na par mjesta, npr. kod implementacije interfejsa *AzurirajStanjeBonova*. Međutim detaljnijom analizom smo

zaključili da bismo ipak imali dosta poteškoća prilikom realizacije na prvobitno planiranim mjestima, pa smo se ipak odlučili za neki drugi vid implementacije. Naš sistem smo proširili sa 2 nove klase, koje su naslijeđene iz klase Student, a to su **StudentPonovac** i **RedovanStudent**. Ovim proširenjem smo uveli jedan novi pristup prilikom rada Blagajne sa studentima. Sada možemo dodati interfejs tipa *uplatiDom*, čija bi se implementacija razlikovala u zavisnosti od toga koji tip studenta uplaćuje. Na ovaj način smo omogućili ako bi se eventualno javila neka još složenija podjela studenata, jednostavno bi koristili svi isti interfejs za uplatu, pri čemu bi se implentacija naravno razlikovala.

- **Flyweight pattern**

  Ovaj patern bismo također mogli primijeniti, prilikom proširenja kompleksnosti našeg sistema, konkretno načina rada Restorana. Ako se odlučimo o proširenju sistema, uvođenjem klasa tipa Jelo, koja bi čuvala atribut sastojci, u kojem bi se nalazili svi sastojci. Onda upotrebom ovog paterna, na nekom mjestu bi se čuvala lista svih sastojaka, te bi se prilikom dodavanja sastojaka u neko jelo, prvo provjerilo da li taj sastojak već postoji u našem sistemu/listi sastojaka. Ako postoji, jednostavno bi se on upotrijebio, a ako ne, onda bi se dodao te omogućio na raspolaganje u daljnjem radu Restorana i sistema općenito. Na ovaj način bismo izbjegli bespotrebno dupliciranje objekata, koji bi zapravo predstavljali jedan te isti objekat, odnosno izbjegla bi se potreba za kreiranjem nove instance neke namirnice, prilikom kreiranja/modifikovanja svakog jela.

Kreacijski paterni

- **Singleton pattern**

  Potreba za korištenjem ovog patterna se javlja u slučaju da je tokom rada nekog sistema, potrebna samo jedna instanca neke klase, koja će biti dostupna svim klasama u sistemu. Nakon razmatranja ovog patterna, uočili smo da je idealna klasa koja će predstavljati **singleton** klasu, zapravo već postojeća klasa **StudentskiDom**. Ova klasa bi se instancirala jednom prilikom pokretanja našeg sistema i nakon toga bi bila dostupna sa bilo kojeg dijela programa. Na ovaj način, spriječeno je da se negdje tokom kreiranja i korištenja ovog sistema, kreira neka druga instanca iste ove klase. Sada prilikom poziva bilo koje metode ove klase, znamo da će uvijek isti objekti biti korišteni.

- **Prototype pattern**

  Mogućnost implementacije ovog paterna u našem sistemu, ogledamo kod instanciranja paviljona. Naime, kada kreiramo paviljon, potrebno je i kreirati listu svih soba koje se nalaze u tom paviljonu. Međutim, pošto su sobe uglavnom iste, prilikom kreiranja, razlikuju se samo u kapacitetu i brojuSobe, javlja se potreba za korištenjem ovog paterna. Na ovaj način smo omogućili našem sistemu da samo kloniramo već postojeću sobu, a onda vršimo eventualnu modifikaciju dobijene sobe.

- **Factory method pattern**

  Priliku za implementaciju ovog paterna smo uočili kod klase Zahtjev. Međutim pošto smo već ranije isplanirali taj dio, odlučili smo ipak da ne primijenimo ovaj patern na naš sistem, ali ako se javi potreba za tim, smatramo da nam neće biti prevelik problem. U slučaju da se iskoristi ovaj pattern, implementacija bi išla sljedećim tokom; napravila bi se klasa **Creator** kojoj bi na neki način rekli o kojem tipu zahtjeva se radi, a ona bi zatim vratila instancu tog zahtjeva. Nakon toga, nekom polimorfnom metodom tipa *popuniZahtjev* koja bi se nalazila u klasi Zahtjev bi se taj zahtjev popunio, a budući da je vraćena instanca odgovarajućeg zahtjeva, znalo bi se tačno o kojem zahtjevu se radi i način na koji se zahtjev popunjava. Na kraju, zahtjev bi bio popunjen i jednostavno bi se zahtjev poslao, odnosno dodao u listu zahtjeva studentskog doma.

- **Abstract Factory pattern**

  Trenutno u našem sistemu, nemamo mogućnost implementacije ovog paterna. Jedna od mogućih realizacija ovog paterna, bi se javila u

slučaju proširenja kompleksnosti našeg sistema, sa proširenjem i dodavanjem novih funkcionalnosti restoranu. Za primjer čemo uzeti generisanje Menija. Mogli bismo proširiti naš sistem, da unutar jedne **AbstractFactory** klase imamo metode, koje generišu meni za svaki dan u sedmici. Međutim koji meni će se kreirati, zavisio bi od perioda u godini i mogućih događaja koji se odvijaju unutar jedne godine. Kao što smo već ranije naveli, primjer bi bio poseban vid menija za Ramazan. Tako bi postojale dvije različite klase tipa RamazanskiMeni, DnevniMeni i sl., koje bi imale iste metode kao naša **AbstractFactory** klasa. Zatim bi se, kao što je već navedeno, u zavisnosti od perioda u godini i nekih događaja, kreirao odgovarajući meni.

- **Builder pattern**

  U našem sistemu, trenutno ne vidimo mogućnost korištenja ovog paterna. Međutim, kao i u prethodnim slučajevima, eventualnom nadogradnjom sistema, javila bi se prilika za implementaciju paterna. Ako bismo proširili naš Restoran, da radi sa Jelima, koji bi se sastojali od nekih sastojaka pri čemu bi se i sam koncept Menija promijenio, mogli bismo koristiti ovaj pattern. Npr. postupak kreiranja Jela odnosno Menija, bi se ogledao u korištenju nekog **Builder** interfejsa. Ako bismo koristili **Builder** interfejs za kreiranje Jela, u tom interfejsu bismo mogli dodati metode koje će se pozivati za dodavanje određenih sastojaka u navedeno jelo. Npr. u zavisnosti od toga koliku porciju jela želimo, mogli bismo neku metodu za dodavanje sastojaka pozivati više puta. Drugi primjer bi eventualno bi kod kreiranja Menija, gdje bismo koristili metode koje će dodavati određenu vrstu jela na Meni, tipa dodajPredjelo, dodajDesert i sl.