Strukturalni i kreacijski patterni

Predmet: Objektno orijentisana analiza i dizajn

Grupa: SeKEmin

Lazović Kemal

Suljević Semir

Šiljak Emin

Strukturalni patterni

1. Adapter pattern

   Ovaj pattern služi da se postojeći objekat prilagodi za korištenje na neki drugi način u odnosu na način kako se koristio. Mora se obezbijediti da objekat radi kako treba ono što je i prije radio kao i da radi dobro u novom načinu rada. U našem sistemu ne možemo navesti primjer ovog patterna. Međutim ukoliko bismo uveli neku klasu za plaćanje usluga studentskog doma na način da student plaća usluge u kešu, sada ukoliko bismo željeli dodati mogućnost plaćanja karticom mogli bismo uvesti interfejs IPlacanje koji ima metodu za plaćanje karticom. Dalje dodajemo klasu za plaćanje karticom (adapter klasa) koja je naslijeđena iz klase za plaćanje usluga doma i implementira interfejs IPlacanje. Sada korisnik ima mogućnost plaćanja usluga kako u kešu tako i karticom.

2. Facade pattern

   Ovaj pattern ima ulogu pojednostavljivanja korištenja korisnicima. Ukoliko koristimo više složenih klasa u našem sistemu i želimo pojednostaviti korištenje sistema uvedemo neku novu klasu koja će predstavljati fasadu i preko koje ćemo pristupati ostalim klasama sistema. Kao primjer fasadnog patterna u našem sistemu možemo navesti klasu StudentskiDom. Korisnik putem te klase može pristupiti studentima, upraviteljima kao i zahtjevima bez da zna kako su navedene klase implementirane što je i suština ovog patterna. Ovo je izvedeno tako što je klasa StudentskiDom povezana sa klasama Student, Zahtjev kao i interfejsom IUpravitelj, tj. klasa StudentskiDom kao svoje atribute ima liste navedenih objekata kojima može pristupiti.

3. Decorator pattern

   Pattern služi za omogućivanje različitih unapređenja objektima koji u osnovi predstavljaju istu vrstu objekta. Umjesto definisanja velikog broja izvedenih klasa dovoljno je da omogućimo dodavanja različitih detalja te tako i pojednostavljivanje objekata. U našem sistemu konkretno ne možemo navesti primjer ovog patterna. Međutim ukoliko uvedemo funkcionalnost za uređivanje soba u studentskom domu. Neka korisnik ima mogućnost kako estetskog uređivanja sobe tako i uređivanje čišćenjem, pa za ta dva uređivanja pravimo klase. Dalje pravimo interfejs ISoba koja od metoda ima urediSobu kao i dajSobu. Ovaj interfejs implementiraju klase za uređivanje sobe koje sadrže i atribut tipa soba koji predstavlja stanje sobe prije odrađenog uređivanja. S ovim klasama upravitelj zaduživanjem pored veze s klasom Soba ima i vezu sa interfejsom ISoba pa tako omogućujemo uređivanje soba od strane upravitelja.

4. Bridge pattern

   Pattern služi za odvajanje apstrakcije nekog objekta od njegove implementacije i pomaže nam u ostvarenju open-closed SOLID principa. Za primjer bridge patterna možemo navesti Upravitelje. U prethodnoj verziji dijagrama klasa smo imali jedan interfejs

IUpravitelj sa metodom obradiZahtjev i taj interfejs je implementiran od strane različitih vrsta upravitelja gdje oni na različite načine obrađuju primljene zahtjeve. Taj interfejs je povezan sa kontejnerskom klasom StudentskiDom. Izmjena koju smo napravili da implementiramo ovaj pattern jeste dodavanje bridge klase (BridgeUpravitelj) koja predstavlja vezu između interfejsa i klase StudentskiDom. U toj klasi kao atribut imamo instancu interfejsa i imamo jednu metodu koja daje različite rezultate u zavisnosti koji je dinamički tip atributa klase.

5. Composite pattern

Ovaj pattern služi za kreiranje hijerarhije objekata. Ukoliko objekti imaju različite implementacije neke metode a svima je potrebno na isti način pristupiti. Ovaj pattern bismo mogli iskoristiti ukoliko bismo izmijenili klasu Zahtjev. Prva izmjena bi bila brisanje atributa u klasi jer u tom slučaju nam ne trebaju obrisani atributi. Dalje u klasu zahtjev dodamo metodu za slanje zahtjeva koja radi na različite načine u zavisnosti od vrste zahtjeva, tj. šalje zahtjev različitim upraviteljima. Pošto apstraktna klasa Zahtjev nema atributa možemo od nje napraviti interfejs koji će implementirati sve klase koje su bile izvedene iz klase Zahtjev kao i naša kontejnerska klasa StudentskiDom. Na taj način se kreirala hijerarhija objekata i iskoristio composite pattern.

6. Proxy pattern

Namjena proxy pattern-a je da omogući pristup i kontrolu pristupa stvarnim objektima. U općem obliku, proxy je klasa koja funkcionira kao interfejs za nešto drugo (neki drugi objekat).

Ovaj pattern bismo mogli koristiti u našem projektu da bismo osigurali da samo upravitelj zaduživanjem može odobriti smještaj. To bi bilo korisno u slučaju da se metoda odobriSmjestaj nalazi u interfejsu Iupravitelj. Tada bismo uveli novu klasu npr. ProxyUpraviteljZaduzivanjaSoba. Ta klasa bi kao atribut imala objekat tipa IUpravitelj (npr. imena upravitelj) koji bi se postavljao u konstruktoru. A u implementaciji metode odobriSmjestaj bi tada bilo potrebno provjeriti instanceof upravitelja. Ako je bilo šta osim ProxyUpraviteljZaduzivanjaSoba, zabraniti izvršavanje akcije.

7. Flyweight pattern

Osnovna namjena flyweight pattern-a je da se omogući da više različitih objekata dijele isto glavno stanje, a imaju različito sporedno stanje. Ovaj pattern se inače koristi kada bi ponavljanje određenog objekta zauzimalo prekomjernu količinu memorije. Za implementaciju ovog patterna se uvodi jedan zajednički objekat (stanje) koji više objekata međusobno dijeli pri čemu također svaki od tih objekata imaju atribute koji su karakteristični samo za njih.

U našem projektu nema nekog razloga za korištenje ovog patterna jer nemamo nekih objekata koji se ponavljaju više puta. Primjera radi, mogli bismo implementirat na sljedeći način. Pretpostavimo da želimo proširiti aplikaciju sa tim da u svakog dijelu

studentskog doma imamo razne radnike. Dakle, radnici u kuhinji, u sobi za zabavu, u biblioteci itd. Svaki od upravitelja tih soba bi tada morao evidentirati na neki način zaposlene radnike. Da ne bismo čisto držali listu radnika u svakom upravitelju, mogli bismo u upravitelj klasama držati objekat EvidencijaRadnika. A dalje da ne bismo kod svakog upravitelja ponavljali taj objekat (i time zadovoljili Flyweight pattern), možemo napraviti jednu zajedničku klasu EvidencijaRadnika koja bi u sebi mogla držati listu parova Radnika i broja sobe u kojoj radi. Na osnovu broja sobe se raspoznaje u kojoj sobi radnik radi i tada bi svaka Upravitelj klasa mogla koristiti ovu klasu. Umjesto liste parova, mogli bismo u klasi Radnik dodati atribut broj sobe u kojoj radnik radi, tada bi klasa evidencije radnika morala u sebi držati samo listu radnika.

Kreacijski patterni

1. Singleton pattern

   Singleton pattern je pattern koji omogućava postojanje samo 1 instance jedne klase. To se vrši stavljanjem konstruktora kao private, te kreiranjem posebne metode getInstance koja se poziva ako je instanca date klase null. Ovaj pattern ćemo implementirati u naš projekat, tačnije naša klasa StudentskiDom će biti singleton klasa. To smo odlučili zbog toga što korisnik sve akcije sa aplikacijom vrši direktno preko ove kontejnerske klase, te nema smisla da postoji više instanci klase StudentskiDom.

2. Prototype pattern

   Prototype pattern služi da kreira nove objekte klonirajući jednu od postojećih prototip instanci. Koristi se većinom kada je veliki trošak konstatno kreiranje novih objekata (korištenjem ključne riječi „new“).

   Objekti koji se javljaju u našem projektu nisu toliko memorijski zahtjevni, tako da nema neke velike potrebe za primjenom ovog pattern-a. Primjera radi, prototype pattern bismo u našem projektu implementirali tako što bismo omogućili kopiranje soba. Bitno je napomenuti da bismo vršili duboko kopiranje jer klasa soba u sebi, kao atribute, ima reference (tačnije reference na klasu Student). Ovo bi bilo korisno npr. u slučaju da oba studenta moraju da vrše određene operacije sa klasom Soba.

3. Factory Method pattern

Factory Method omogućava kreiranje objekata na način da podklase odluče koju klase instancirati. To je ostvareno koristeći interfejse jer klase na različite načine implementiraju metode interfejsa.

Ovo je iskorišteno kod hijerhije Osoba – Student. Klasa Student je naslijeđena iz apstraktne klase Osoba, dok su klase StudentHrana, StudentStanovanje, StudentZabava i StudentBiblioteka naslijeđene iz klase Student.

Odlučeno je korištenje zbog broja klasa uključenoj u hijerarhiji. Metoda dajStudenta u klasi Studenta u zavisnosti od parametra koji je pobrojanog tipa, vraća IStudent. Interfejs IStudent ima metodu posaljiZahtjev koja u slučaju StudentStanovanje će odigrati ulogu produzivanja i razduzivanja smještaja. U klasi StudentZabava rezervaciju termina u sobi za zabavu. StudentHrana – trošenje bonova i StudentBiblioteka – slanje zahtjeva za podizanje knjige.

P.S. napisano je i na class diagramu, ali evo i ovdje: klase StudentBiblioteka i StudentHrana u metodi posaljiZahtjev primaju objekte tipa Knjiga i Jelo, respektivno. Zbog toga je stavljen parametar tipa Object, koji u prve dvije klase neće biti korišten.

Moglo je s dvije metode istog imena, ali bi bila jedna prazna u sve 4 klase.

4. Abstract Factory Pattern

Omogućava enkapsuliranje grupe individualnih *tvornica* koje imaju zajedničku značajku, bez da definisanja funkcionalnosti klasa. Na osnovu apstraktne *tvornice*, kreiraju se konkretne *tvornice*.

Na neki način je već implementirano u projektu, kod obrade zahtjeva od strane upravitelja. Moguća implementacija jeste da se kreira interfejs IUpravitelj i interfejs IUpraviteljFactory. S metodama obradiZahtjev, void, i kreirajUpravitelja koja vraća IUpravitelj, respektivno. Nakon toga kreirale bi se specifične klase za svaku vrstu upravitelja (da ne navodim imena), koje implementiraju interfejs IUpraviteljFactory.

Kako interfejs IUpraviteljFactory ima metodu kreirajUpravitelja, svaka od tih klasa bi kreirala specifičnu vrstu upravitelja.

Nakon toga, kreiraju se specifične klase za tipove zahtjeva, koje implementiraju IUpravitelj interfejs. Kako taj interfejs posjeduje metodu obradiZahtjev, tako bi sve vrste zahtjeva posjedovale različit način obrade.

5. Builder pattern

Uloga Builder paterna je odvajanje specifikacije kompleksnih objekata od njihove stvarne konstrukcije. Isti konstrukcijski proces može kreirati različite reprezentacije.

Njegova mogućnost biti će prikazana na primjeru kreiranju osobe.

Imamo klasu StudentStanovanje s potrebnim atributima.

Interfejs IStudentStanovanjeBuilder s metodom GetStudentStanovanje() koja vraća objekat tipa StudentStanovanje.

Klasa StudentBuilder koja implementira IStudentStanovanjeBuilder. Metodu GetOsoba implementira na način da vraća objekat tipa StudentStanovanje s postavljenim vrijednostima atributa.

Klasa StudentStanovanjeDirector koja posjeduje privatni atribut IStudentStanovanjeBuilder, konstruktor u kojem prima objekat tipa IStudentStanovanjeBuilder i inicijalizira atribut. Metodu Kreiraj koja postavlja sve atribute na željeni način, primanjem parametara i slično.

Kreiranje bi teklo na način da neki upravitelj posjeduje metodu u kojoj kreira varijablu tipa StudentBuilder (builder), koju šalje kao parametar konstruktoru varijabli tipa StudentStanovanjeDirector director. Poziva metodu Kreiraj nad varijablom director.

Sada je lagano kreirati objekat tipa Osoba na način StudentStanovanje student = builder.GetResult();