Univerzitet u Sarajevu
Elektrotehnički fakultet Sarajevo

Kreacijski patterni

OOAD 2020-2021

Naziv grupe: Hilbert’s Grand Hotel

Članovi: Nedim Badžak
Harun Alagić
Emil Fejzagić

1. Singleton

Singleton pattern osigurava da se klasa može instancirati samo jednom i da osigura globalni pristup kreiranoj instanci klase. Postoje objekti čija je samo jedna instanca potrebna i nad kojima je potrebna jedinstvena kontrola pristupa. Instanciranje više nego jednom može prouzrokovati probleme kao što su nekorektno ponašanje programa, neadekvantno korištenje resursa ili nekonzistentan rezultat.

Što se tiče primjene ovog kreacijskog patterna, u našem sistemu postoji klasa “Šef” koja ne bi smjela više puta biti instancirana jer je šef samo jedan, pa bi kreiranje statičke klase getInstance umjesto gettera znatno pomoglo da ne dođe do nepredviđenog ponašanja programa ili neadekvatnog korištenja resursa.

2. Prototype

Prototype pattern kreira nove objekte klonirajući jednu od postojećih prototip instanci. Dakle, ovaj pattern omogućava jednostavnije kreiranje novih instanci klasa kod kojih je veliki broj atributa identičan za većinu instanci, kako bi se uštedjelo na resursima i pojednostavio rad sa objektima.

U našem sistemu, ovaj kreacijski pattern bi se mogao iskoristiti na klasi “Soba” s obzirom da je većina atributa najćešće jednaka, pa samim time kreiranje kopije ima smisla. Način na koji bismo implementirali ovaj pattern bi bio da napravimo “Prototype” interface koji sadrži metodu kloniraj() i implementirati metodu kloniraj() u klasi “Soba”. Implementacija metode bi pokrivala klasično kopiranje, kao i neke rubne slučajeve i rješavanje ugnježdenih informacija koje bi plitkim kopijama izazvale zavisnost, što ovaj pattern pokušava da spriječi.

3. Factory Method

Factory Method pattern omogućava kreiranje objekata na način da podklase odluče koju klasu instancirati. Različite podklase mogu na različite načine implementirati interfejs.

S obzirom da ovaj pattern odlikuje olakšano naknadno dodavanje novog podtipa neke klase i zahtijeva samo preklapanje onoga što vraća interface koji vrši kreiranje, umjesto da se kod nalazi u konstruktoru postojeće klase pa kasnije zahtijeva znatne izmjene, a u našem sistemu postoji mogućnost događaja da se pojavi još neki tip “Zaposlenika”, smatramo da bi primjena ovog patterna u našem slučaju bila izvedena tako što bismo samog “Zaposlenika” kreirali pomoću kreacijskog interface-a umjesto new operatora.

4. Abstract Factory

Abstract Factory pattern se koristi pri radu sa familijama različitih objekata. Na osnovu apstraktne familije objekata kreiraju se konkretne fabrike produkata različitih tipova i različitih kombinacija. Familije produkata se mogu jednostavno izmjenjivati ili ažurirati bez narušavanja strukture klijenta.

Budući da se ovaj pattern odlikuje sa mogućnosti grupisanja sličnih tipova da bi se spriječilo narušavanje strukture, a ujedno i pojednostavio dalji rad sa njima, mjesto na kojem bismo mogli iskoristiti ovaj pattern je u slučaju da proširimo sistem tako da sadrži i “Smještaj” klasu koja je nadklasa “Apartman” i “Soba” klasama. Navedene dvije klase bi predstavljale prvu podjelu i bile apstraktne klase koje bi se dalje mogle proširivati i nasljeđivati u polu-pansion sobe, polu-pansion apartmane, pansion sobe i pansion apartmane. S obzirom da ove podjele imaju dosta zajedničkog, možemo kreirati familije polu-pansion i pansion koje bi bile fabrike za naše dvije klase i njihove dalje tipove.

5. Builder

Builder pattern omogućava konstrukciju kompleksnih objekata korak po korak i odvaja specifikaciju kompleksnih objekata od njihove stvarne konstrukcije. Isti konstrukcijski proces može kreirati različite reprezentacije i tipove.

Svojstva ovog patterna su smanjenje veličine konstruktora i delegiranje poslova konstruktora u manje builder klase gdje se na taj način omogućava lakše kreiranje dalje izvedenih klasa koje zahtijevaju još neku dodatnu informaciju u odnosu na njihovu roditeljsku klasu. Primjer za to u našem sistemu bi bio “Zaposlenik” iz kojeg su naslijeđeni “Šef” i “Osoblje”, kao i u krajnjem slučaju i “Recepcioner” koji imaju većinu zajedničkih atributa, i svaki od njih još par individualnih i specifičnih samo za njih. Način na koji bismo mogli implementirati ovaj pattern je da napravimo “Direktor” klasu koja bi imala metode napraviSefa(), napraviOsoblje() i napraviRecepcionera(). Dalje bi bilo potrebno kreirati builder interface koji bi u sebi sadržavao specifične metode za dodavanje svih informacija. Nakon toga bismo napravili par builder klasa koje bi implementirale builder interface. Na ovaj način bi se “Direktor” klasa pobrinula za raspoređivanje kojem builderu se prosljeđuje zahtjev za kreiranje datog tipa.