Kreacijski paterni

Singleton patern – je kreacijski patern koji nam omogućava da klasa ima samo jednu instancu, dok nam pruža globalni pristup toj instanci.

Iskoristili smo ovaj patern prilikom logiranja korisnika, jer nam je potrebna jedna instanca koja je dostupna svim klijentima i da bismo imali imali jedinstveno upravljanje bazom u tom slučaju. Također se ovaj patern može iskoristiti i kasnije u projektu kada budemo radili sa bazom naprimjer pri registraciji korisnika da postoji jedinstvena klasa sa metodama za pristup bazi koja bi omogućila da se registriraju korisnici bez da svaki put se moramo posebno konektovati na bazu, i slično.

Implementira se na način da imamo privatni staticki atribut gdje cemo drzati instancu u klasi, privatni static konstruktor kojem će moći prostupiti samo ta(Singleton) klasa, i public metoda getInstance koja omogućava pristup instanci klase, koja će prvi put da pozove konstruktor ukoliko instanca nije kreirana, a svaki naredni put vraća instancu, na ovaj način smo i predvidjeli ovo korištenje u class diagramu.

Osigurali smo da se ova klasa ima jedinstvenu (single) instancu i globalni pristup toj instanci i jedinstveno pristupanje bazi podataka, te singleton objekt je inicijaliziran samo kada se ukazala potreba za njim prvi put.

Prototype patern – je kreacijski patern koji nam omogućava da kopiramo postojeće objekte bez da kod bude ovisan o klasama. Pošto moramo znati klasu objekta da bismo mogli kreirati kopiju, kod nam postaje ovisan o toj klasi, da bismo ovo izbjegli koristimo prototype patern. Također kada je trošak kreiranja novog objekra velik i kreiranje objekta je resursno zahtjevno, vrši se kloniranje postojećeg objekta.

Implementira se na način da klasa Client zahtijeva klonranje postojećeg objekta preko interfejsa Iprototype, to je interfejs kojim se omogućava kloniranje postojećih konkretnih objekata.

Ovaj patern ćemo iskoristiti u našem projeku za rad sa bazom podataka. Kako ćemo u bazi podataka držati sve potrebne informacije kao što su svi korisnici, sva putovanja, kodovi, ocjene itd. Ovaj patern cemo iskoristiti na način da samo jednom kopiramo podatke iz baze i uradimo analizu, npr za dohvatanje podataka o tome na kojim je sve putovanjima bio klijent koji pristupa aplikaciji. Kada korisnik želi da da ocjenu nekom putovanju na kojem je bio da se ne bi ponovo pristupalo bazi, čitanje podataka i enkapsuliranje podataka u novi objekat, klonirat ćemo postojeći objekat koji se već koristio i iskoristiti podatke za računanje nove prosječne ocjene i zapisivanje u bazu... Imat ćemo interface koji će nam omogućiti

kloniranje. Klasu za pristup bazi koja će implementirati ovaj interfejs i u njoj će se nalaziti sve metode koje su potrebne za rad sa bazom i konektovanje.

Upotrebom ovog paterna bismo spriječili ponovno pristupanje bazi podataka, ponovno čitanje podataka i kreiranje objekata za njihovo smiještanje, također ovaj patern omogućava da se izbjegne bespotrebni dio inicijalizacijskog koda, da možemo povoljnije koristiti kompleksne objekte.

Factory Method patern – uloga ovog paterna je da omogući kreiranje objekata na način da podklase odluče koju klasu instancirati. Različite podklase mogu na različite načine implementirati interfejs. Factory Method instancira odgovarajuću podklasu preko posebne moetode na osnovu informacije od strane klijenta ili na osnovu tekućeg stanja. Koristimo ga kada na početku ne znamo kojeg je tačno tipa objekat sa kojim radimo, jer odvaja dio koda konstrukcije objekta od dijela u kojem zapravo koristimo taj objekat. Također koristimo ovaj patern kada želimo da sačuvamo resurse sistema, ponovnim korištenjem postojećih objekata umjesto da ih „rebuild“ svaki put.

Implementira se na način da imamo interfejs Iproduct, zatim konkretne klase Product koje implementiraju interfejs. Klasu Creator koja posjeduje FactoryMethod() metodu koja odlučuje koju klasu instancirati. I klijent može imati više od jednog kreatora za različite tipove produkata.

Mogli bismo ga iskoristiti u našem projektu da smo imali više Agencija, te da se na osnovu grada u kojem je klijent odluči koja će se agencija instancirati.

Ovaj patern nam omogućava da izbjegnemo usku povezanost između kreatora i pojedinih produkata i slijedi single responsibility princip, jer možemo kreiranje „proizvoda“ premjestiti na jedno mjesto u programu. Također i open/closed princip jer se mogu dodavati novi tipovu „proizvoda“ u program bez da se mijenja postojeći kod klijenta.

Abstract Factory patern – omogućava da se kreiraju familije povezanih objekata/produkata bez da specificiramo njihovu konkretnu klasu. Na osnovu apstraktne familije produkata kreiraju se konkretne fabrike (factories) produkata različitih tipova. Patern odvaja definiciju klase od klijenta. Familije produkata se mogu jednostavno izmjenjivati ii ažurirati bez narušavanja strukture klijenta. Koristimo ovaj patern kada kod treba da tadi sa različitim familijama povezanih produkata, ali ne želimo da ovisi o konkretnoj klasi tih produkata. Također i za generiranje različitih izgleda i korisničkih interfejsa i za pisanje portabilnog koda za različite operativne sisteme za iste operacije.

Implementira se na način da imamo Ifactory apstraktni interfejs sa Create operacijama za svaku fabriku, zatim klase Factory koje implementiraju operacije kreiranja za pojedinačne

fabrike, apstraktne interfejse za pojedinačne grupe produkata, te klase koje implementiraju interfejse produkata i definiraju obekte koji se kreiraju za odgovarajuću fabriku. Tu je i Client klasa koja preko interfejsa koristi objekte(produkte) fabrike.

U našem projektu bismo ga mogli iskoristiti da imamo omogućeno putovanje različitim vrstama prijevoznih sredstava, i da imamo izbor let avionom i vožnju autobusom, apstraktne interfejse koji bi predstavljali ove dvije opcije. Te da imamo Factory-je Aviokompaniju i neku kompaniju koja pruža usluge vožnje autobusom, te za svako putovanje bi imali pojdinačne vožnje avionom i autobusom.

Ovaj patern omogućava da budemo sigurni da proizvodi koje dobivamo iz factory su kompaktibilni jedni sa drugima, omogućava nam da izbjegnemo usku vezu između konkretnih objekata(produkata) i klijent koda. i slijedi single responsibility princip, jer možemo kreiranje „proizvoda“ premjestiti na jedno mjesto u programu. Također i open/closed princip jer se mogu dodavati novi tipovu „proizvoda“ u program bez da se mijenja postojeći kod klijenta.

Builder – omogućava nam da kreiramo kompleksne objekte korak po korak. Ovaj patern nam omogućava da proizvedemo različite tipove i reprezentacije objekata koristeći isti konstrukcijski kod.

Osnovni elementi ovog paterna su Ibuilder interfejs koji definira pojedinačne dijelove koji se koriste za izgradnju produkta, Director kllasa koja sadrži neophodnu sekvencu operacija za izgradnju produkata. Builder klasu koja se poziva od strane direktora da se izgradi produkt, te Product klasa na osnovu koje se kreira objekat koji se gradi preko dijelova.

U našem projektu nismo koristili ovaj patern, ali da smo imali funkcionalnost da klijent odabere da li želi puni pansion tokom putovanja, sa doručkom, ručkom i večerom. Mogli bismo iskoristiti ovaj patern da se naprave ti obroci od nekih dijelova tj namirnica, te bismo imali klasu Kuhar koja bi implementirala interfejs IBuilder. Neku klasu Šef koja bi omogućavala da se konstruira objekat iz više djelova. Dijelovi bi bili namirnice, a sam proizvod(produkt) taj obrok.

Ovaj patern nam omogućava da ponovo koristimo kod(reuse) kada gradimo različite reprezentacije produkata, također slijedi princip Single Responsibility, možemo odvojiti kompleksni konstrukcijski kod od biznis logike produkta(objekta).