Kreacijski paterni

1. **Singleton**
   Uloga ovog kreacijskog paterna je da osigura da se klasa samo jednom instancira i da ima globalni pristup prema njenoj instanci.

   Gledajuci naš sistem, možemo primijetiti da vecina korisnih funkcionalnosti zahtjeva upotrebu logiranja. Buduci da se upravo ovaj patern koristi kada se u sistemu desava mnogo logiranja, mozemo dodati novu klasu „LogTracker“. Ta klasa bi nam pomogla tako sto bi u cijelom sistemu vrsila logiranje na najefikasniji moguci nacin. LogTracker mora imati privatni staticki atribut, privatni konstruktor i getter.

2. **Prototype**
   Uloga ovog kreacijskog paterna je da klonira objekte koji zauzimaju mnogo resursa.

   U našem sistemu najkompleksnija klasa je „SportskiCentar“ koja ce biti i najvise koristena (*bice kreiran veliki broj instanci ove klase*). Kada bi htjeli napraviti (a ne naslijediti) klasu „SportskiCentarSaTerminima“, tu bi nam ovaj patern bio od velike pomoci. Mogli bi napraviti interfejs ISportskiCentar koji bi nam omogucio da kloniramo instacnu te klase te bi mogli jednostavno dodati atribut (*modifikovati*) listu rezervacija.

3. **Factory Method**
   Uloga ovog kreacijskog paterna je da omoguci kreiranje objekata na nacin da se odluci koja se podklasa instancira.

   Naš sistem bi se nekada u buducnosti mogao prosiriti tako da korisnik u pocetku moze izabrati da li zeli „*outdoor*“ ili „*indoor*“ aktivnost. U tom slucaju bi nam ovaj patern mogao pomoci. Bilo bi potrebno napraviti dvije nove klase npr. IndoorSport i OutdoorSport koje bi implementirale interfejs ISport. Pored ovih klasa moramo napraviti i klasu Kreator kako je definisano u samoj definiciji paterna. Ona bi morala imati metodu Factory koja bi instancirala odgovarajucu podklasu u sistemu.

4. **Abstract Factory**
   Ovaj kreacijski patern omogucava da se kreiraju familije povezanih objekata.

   Trenutno kako je dizajniran nas sistem on bi trebao da ispuni vecini korisnika sve potrebe i zahtjeve. Ali svjesni smo da postoji odredjeni broj ljudi koji vole i

neke ekstremne i visoko adrenalinske sportove. Tu dolazi Abstract Factory patern koji bi nam omogucio da napravimo dvije familije produkata (obicnih sportova i adrenalinskih/ekstremnih sportova). Morali bi kreirati dva interfejsa IAdrenalinskiSport i ISport. Zatim bi mogli kreirati posebne klase za ova dva interfejsa.

5. **Builder**

Buduci da ovaj patern pomaze kada imamo neku klasu koja je sklona nadogradjivanju i dodavanju parametara u konstruktor. Kod samim time postaje slozen i necitljiv. Jedan klasicni primjer koristenja ovog paterna je upotreba u kreiranju bankovnog racuna pa cemo i mi isti primjer iskoristiti u nasem sistemu.

Trenutno u nasem sistemu postoje samo osnovne informacije o korisnickom bankovnom racunu, ukoliko bi se nekada htjela dodati funkcionalnost da korisnik moze da vidi historiju svih transakcija koje je imao sa nasim sistemom morali bi dodati nove atribute i dodatno zakomolikovali kod. Tu dolazi Builder patern s kojim cemo taj problem rijesiti.

Prvo cemo napraviti klasu Builder koja ce sadrzavati sve atribute bivse klase BankovniRacun. Kreirati cemo settere za sve atribute i u svakom setteru cemo vratiti sam builder (*time smo napravili „fluentni“ interfejs*). Nakon toga moramo obrisati javni konstruktor klase BankovniRacun te ga zamijeniti sa privatnim tako da samo Builder moze kreirati instancu BankovnogRacuna.

Sada u kodu mozemo kreirati nove racune koristeci sve settere povezano (*kao što smo navikli u Javi*).