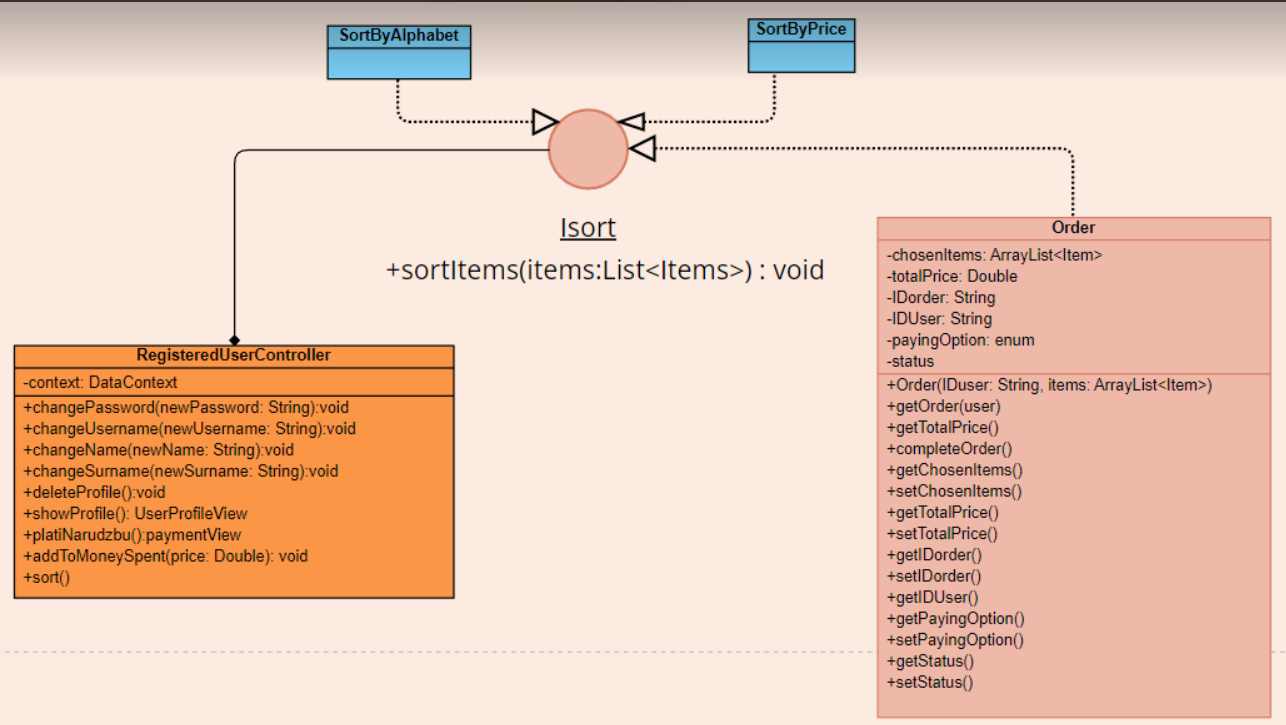
## Paterni ponašanja

1. **Strategy pattern** – koristi se za odabir jednog od više mogućih algoritama za rješavanje nekog problema, što olakšava dodavanje/brisanje novih algoritama. U našem sistemu, ovaj pattern bismo iskoristili pri sortiranju artikala (po cijeni – opadajuće/rastuće, po abecedi). Napravit ćemo interfejs iSortiranje iz kojeg ćemo naslijediti odgovarajuće vrste sortiranja.



1. **State** – možemo ga opisati kao dinamički Strategy pattern, koji omogućava mijenjanje stanja objekta, međutim ne na korisnikov zahtjev, već automatski. Kako bismo dodatno unaprijedili naš sistem, možemo običnog korisnika pretvoriti u VIP korisnika jednom kada zbir njegovih potrošnji na našem sistemu pređe 1000 KM. Dodali bismo u RegisteredUserController interfejs iState.
2. **Template metod** – ovim metodom izdvaja se dio kompleksnog ili dugog algoritma u određene podklase. Ti manji dijelovi se kasnije mogu koristiti u ostatku koda. Ukoliko u našem sistemu nekad budemo imali jako puno artikala, bilo bi dobro napraviti određeni algoritam koji će sortiranje vršiti “po preporuci” za svakog korisnika, odnosno, na osnovu artikala koji su se korisniku prethodno sviđali, da algoritam ustanovi koji bi mu se proizvodi moći još eventualno svidjeti, pa da ih izbacuje među prvim. S obzirom da bi takav algoritam bio dosta kompleksan, trebalo bi ga podijeliti u više nezavisnih dijelova koji će kasnije biti ponovo iskoristivi (npr. dio odlučivanja koji artikli se korisniku sviđaju moći ćemo kasnije iskoristiti u “pair with”).
3. **Observer pattern** – služi da, kada se promijeni stanje jednog objekta, drugi objekti, kojima to nešto znači, dobiju informaciju o promjeni stanja. Mi bismo ovaj pattern implementirali u našem sistemu tako što bismo obavještavali korisnika kada dođe nova kolekcija ili sezona sniženja. Dakle korisničke klase bi implementirale IObserver, dok bi se u ItemControlleru mijenjalo stanje itema.
4. **Iterator** – služi da bi omogućio kretanje kroz kolekciju čiji tip elemenata nam nije poznat. U našem sistemu bismo ovaj pattern mogli implementirati pri prolasku kroz kolekciju proizvoda, na osnovu cijene ili abecednog poretka. Napravili bismo npr. klase IteratorPoCijeni, IteratorPoAbecedi koje bi imale metodu prelaska na sljedeći element u kolekciji.
5. **Chain of responsibility pattern** – bihevioralni dizajn pattern za prenošenje zahtjeva kroz lanac handlera. Pri primanju zahtjeva, svaki handler odlučuje da li će procesirati taj zahtjev ili ga prenijeti narednom handleru u lancu. Ovaj pattern bismo mogli iskoristiti za autorizaciju podataka. Npr. kada korisnik želi izvršiti kupovinu, potrebno je da unese dosta dodatnih podataka (npr. adresa, broj telefona i sl.). Za sistem nije sigurno da direktno prima “raw data”, pa bi trebalo uvesti validaciju primanih request-ova. Da bi sistem bio još sigurniji, mogli bismo uvesti detektovanje velikog broja pokušaja unosa passworda s iste IP adrese (jer se vjerovatno radi o zloupotrebi). Da bismo sve ovo mogli ispuniti, koristimo chain of responsibility pattern, koji pretvara određena ponašanja u stand-alone objekte koji se zovu handleri. Odnosno, svaku provjeru bismo izdvojili u zasebnu klasu sa jedinom metodom koja vrši validaciju. Tek onda se zahtjev, zajedno sa provjerenim podacima, može proslijediti novoj metodi kao argument.
6. **Mediator pattern** – služi da se izbjegne povezivanje velikog broja objekata, odnosno uvodi se posrednik (međuobjekat, medijator), s kojim se povezuju objekti, a koji je zadužen za njihovu međusobnu komunikaciju. Dakle, dva objekta ne komuniciraju direktno, već se sve odvija preko navedenog medijatora. Recimo, ako uvedemo mogućnost da VIP korisnici ocjenjuju artikle i ostavljaju komentare ispod artikla ( o samom artiklu, ali i u sveukupnom dojmu koji uključuje i brzinu dostave, kvalitet proizvoda uživo i sl.), ti komentari bi morali prvo proći provjeru (da li sadrži osjetljiv ili uvredljiv sadržaj), pa se tek onda postaviti. Tada bi nam bio potreban IMediator sa metodom za provjeru sadržaja preko nekih ključnih riječi.