Paterni ponašanja

1. **Strategy**
   Ovaj patern ponašanja iako veoma jednostavan za implementirati je vod velike važnosti za poboljsanje koda u nasem sistemu. Kako se koristi da olaksa dodavanje nekih „strategija“ kako ce se nesto ponasati u sistemu mi cemo to iskoristiti za prikaz (*tačnije sortirani prikaz*) sportskih centara u Sportu.

   To ćemo uraditi tako što ćemo u kontrolersoj klasi Sporta dodati metodu Sortiraj(), pored toga moramo dodati interfejs ISortiranje iz koje je potrebno naslijediti neke vrste sortiranja (*npr. Abecedno, OpadajuciOcjena, VelicinaCentra,...*). Ovim smo pristupom omogućili mnogo lakse dodavanje novih „strategija“ u budućnosti.

2. **State**
   Iako ovaj patern ima veliku slicnost sa vec gore uradjenim Strategy paternom, ova dva paterna su razlicita. Najkraće možemo reći da je State patern ustvari ***dinamički*** Strategy patern.

   Imajući ovo u vidu mi možemo dodatno poboljšati naš sistem. Kako smo u prethodnom razmatranju kreacijskih paterna vidjeli da bi mogli dodati i SportskiCentar koji je „outdoor“, na njegov view bi bilo vrlo funkcionalno dodati i vremensku prognozu. Tu nam može pomoći ovaj patern.
   U Sport kontroleru bi morali dodati instancu novog Interfejsa koji cemo dodati „IState“. Dodali bi novu klasu npr. PrikazCentra u kojoj bi imali dvije metode koje otvaraju indoor i outdoor sportski centar respektivno, dok bi ta klasa implementirala novonapravljeni interfejs.

3. **Template Method**
   Ovaj patern omogućava izdvajanje određenih koraka nekog kompleksnog ili dugog algoritma u odredjene podklase. Time se omogućilo da se i ti dijeli mogu koristiti u ostatku sistema ili da u nasem velikom novom algoritmu možemo slagati dijelove (*kao Lego kockice*).

   U našem sistemu (*budući da je vrlo jednostavno konstruiran*) nemamo potrebu za implementiranjem ovog paterna, ali kada bi se nekada u budućnosti htjela dodati funkcionalnost pamcenja svakog komentara i ocjene za pojedini centar, tu bi se moglo također dodati da dodje do nekog sortiranja ili filtriranja podataka.
   Kako smo već koristili sortiranje podataka u Strategy paternu ovdje bi bilo dobro, pošto će sortiranje ili filtriranje biti mnogo komplikovanije da se rastavi na dijelove. Tako bi „ubili dvije muhe jednim udarcem“, mogli bi koristiti neke dijelove algoritma u nasem sistemu za neke jednostavnije stvari dok bi

jednostavnim spajanjem u veci algoritam dobili sortiranje ili filtriranje kakvo nam je potrebno u novim podacima. (*moglo bi se zamisliti grafički kao odabir opcija za filter na nekim stranicama kao na OLX-u, samo se slažu kao kockice jedna na drugu*)

4. **Observer**
   Ovaj patern, iako veoma zanimljiv, zbog nedostatka vremena na predmetu ga necemo implementirati. Njegova glavna uloga je da obavjestava zainteresovane objekte o promjeni stanja drugog objekta.

   Znamo da u svakodnevnom životu većina stanovništva voli sport i sportske događaje, ako bi neki korisnik odabrao opciju da dobije pozivnicu na neki sportski događaj (*u našem sistemu bi tu informaciju mogli dobiti ako npr. dva poznata kluba rezervisu fudbalski teren*), ili bi mogli dobiti obavijest svaki put kada se u sistem doda novi sportski centar nekog njihovog sporta po želji.

   Konkretno da bi pokazali moguću implementaciju, Sport klasa bi mijenjala stanje i obavijestila Observer klase (*to je u nasem sistemu Korisnik*). Morali bi dodati i interfejs IObserver koji bi implementirala nasa Korisnik klasa.

5. **Iterator**
   Ovaj patern omogućava sekvencijalno kretanje kroz elemente neke kolekcije bez poznavanja strukture same kolekcije.
   U našem sistemu bi mogli iskoristiti ovaj patern jer imamo vise atributa u određenim klasama koji su List-e.

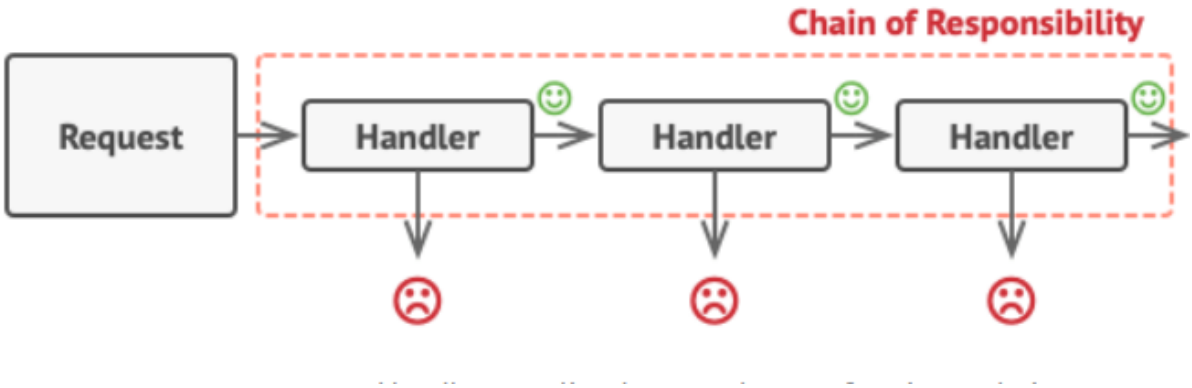
   Pošto imamo List<SportskiCentar> u Sportu, tu bi mogli iskorisitit ovaj patern. U klasi Sport bi imlementirali neku metodu gdje bi sa foreach petljom prolazili kroz elemente liste.
   Morali bi napraviti Interfejs IEnumerable koji bi sadrzavao metodu getEumerator koja se automatski uključuje u pozivu foreach petlje. Na kraju bi još morali da naslijedimo SportskiCentar iz interfejsa koji smo maloćas spomenuli.

6. **Chain of Responsibility**
   Ovaj patern je savršen za kompleksniju autorizaciju podataka.

   Znajući da je naša aplikacija za sada vrlo jednostavna ali nažalost i ranjiva jer radimo sa „raw data“ u našoj autentifikaciji kod provjere lozinke kada korisnik želi da potvrdi narudžbu ili otkaže istu, mogao bi se nekada u budućnosti iskoristiti ovaj patern da poboljšamo sigurnost i sam kod.

   Ove sve potencijalne probleme možemo riješiti tako što ćemo svaku novu provjeru (*modifikaciju, kodiranje, skrivanje,...*) transformisati u „stand-alone“

klase koje nazivamo ***handleri.*** Tada bi svaki novi zahtjev bio ustvari atribut skupa sa podacima s kojim radimo koji bi se slali iz jednog handlera u drugi kao što je prikazano na slici.



7. **Mediator**

Ovaj vrlo korisni patern se koristi kada u sistemu postoji mnogo povezanih klasa koje zavise jedna od druge, mediator ustvari definise objekat koji enkapsuliše kako upravo ovi povezani objekti komuniciraju. Kako bi ga jos bolje shvatili možemo se poslužiti primjerom iz svakodnevnog života, znamo da aerodromi imaju kontrolnu sobu iz koje posrednik komunicira sa avionima kada i gdje ce koji sletjeti. Taj posrednik je ustvari nas Mediator, jer bi nastao haos kada bi piloti međusobno pricali...

U našem sistemu, kako je već naglašeno ranije, nema kompleksnih vezivanja ni mnogo povezanih klasa tako da ovaj patern trenutno nije moguće iskoristiti. Ukoliko bi se nekada sistem zakomplikovao mogli bi ga iskoristiti, npr. kada bi dodali na profil korisnika neke dodatne informacije koje zavise jedna od druge (*kada bi se npr mogao dodati opis, spol, ...*) tada bi mogli dodati neku posrednu klasu koja bi za odgovarajuće poruke prosljedjivala odgovarajuce odgovor, na slici možemo vidjeti jedan takav klasični primjer gdje je klasa Dialog ustvari taj posrednik.