Paterni ponašanja

1. *Strategy*

Ovaj patern ponašanja iako veoma jednostavan za implementirati je od velike važnosti za poboljšanje koda u našem sistemu. Kako se koristi da olakša dodavanje nekih "strategija" kako će se nešto ponašati u sistemu mi ćemo to iskoristiti za prikaz (tačnije sortirani prikaz) artikala u klasi Artikli. To ćemo uraditi tako što ćemo u kontrolerskoj klasi Artikl dodati metodu Sortiraj(), pored toga moramo dodati interfejs ISortiranje iz koje je potrebno naslijediti neke vrste sortiranja (npr. Abecedno, OpadajucaCijena, RastucaCijena, ...). Ovim smo pristupom omogućili mnogo lakše dodavanje novih "strategija" u budućnosti.

2. *State*

Iako ovaj patern ima veliku sličnost sa već spomnutim Strategy paternom, ova dva paterna su različita. 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 Korpu koja je namjena za kupovinu u BiH i u inostranstvu, na njehov view bi bilo vrlo funkcionalno dodati i popuste koji će se zavisno od države razlikovati u odnosu na period u kojim korisnik pristupa. Tu nam može pomoći ovaj patern. U Korpa kontroleru bi morali dodati instancu novog Interfejsa koji cemo dodati „IState". Dodali bi novu klasu npr. PrikazKorpe u kojoj bi imali dvije metode koje otvaraju korpu u BiH i korpu u inostranstvu 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 dijelovi mogu koristiti u ostatku sistema ili da u našem 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 slanja mail-ova. Mogli bismo napraviti template mail-a, kao npr. mail dobrodošlice, obavijesti za popust i slično, gdje bi se mijenjalo ime i prezime korisnika, datum slanja.

4. *Observer*

Ovaj patern, iako veoma zanimljiv, zbog nedostatka vremena na predmetu ga nećemo implementirati. Njegova glavna uloga je da obavještava zainteresovane objekte o promjeni stanja drugog objekta. Znamo da u svakodnevnom životu sve je veća upotreba online kupovine, korisnik bi mogao odabrao opciju da dobije obavijesti za određeni popust, novu kolekciju i slično.

Konkretno da bi pokazali moguću implementaciju, klasa Narudzba bi mijenjala stanje i obavijestila Observer klase. Morali bi dodati i interfejs IObserver koji bi implementirala naša klasa Osoba.

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 u Narudzbi, tu bi mogli iskorisitit ovaj patern. U klasi Narudzba bi imlementirali neku metodu gdje bi sa foreach petljom prolazili kroz elemente liste. Morali bi napraviti Interfejs IEnumerable koji bi sadržavao metodu getEumerator koja se automatski uključuje u pozivu foreach petlje.

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 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. Prilikom provjere narudzbe mogli bismo imati jedan handler za potvrdu narudzbe, drugi za provjeru uplate iznosa, treći za provjeru broja artikala na stanju i slično.

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.

7. *Mediator*

Ovaj vrlo korisni patern se koristi kada u sistemu postoji mnogo povezanih klasa koje zavise jedna od druge, mediator ustvari definiše objekat koji enkapsuliše kako upravo ovi povezani objekti komuniciraju. Kako bi ga još 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, npr. kao što broj telefona zavisi od države. Mogli bismo dodati Mediator koji bi vršio validaciju tih međuovisnih relacija. Uloga Mediatora je da centralizira pravila međuovisnosti i ukoliko bi se javila potreba za još jednom međuovisnom relacijom, mogli bismo to vršiti na jednom mjestu.