Grupa2 – Zamger

Patterni ponašanja

Studenti:

Hamzić Huso

Handžić Rijad

Džumhur Paša

Strategy pattern - izdvaja algoritam iz matične klase i uključuje ga u posebne klase. Pogodan je kada postoje različiti primjenjivi algoritmi (strategije) za neki problem. Strategy patern omogućava klijentu izbor jednog od algoritma iz familije algoritama za korištenje. Algoritmi su neovisni od klijenata koji ih koriste.

Kako u sistemu nemamo nekih procedura odnosno procesa na koje klijent sam može uticati kako će da se izvrše(odnoso koje se mogu vršiti na više različitih načina) zaključujemo da je sa trenutnim stanjem sistema(koji ima dosta procedura koje su jednoznačno izvodljive/ne na više načina) nemoguće primijeniti ovaj pattern. Klijent uopće nema toliko raznovrsnog izbora(odnosno nema uopće) prilikom pozivanja neke funkcionalnosti(nema više načina da postigne isti cilj) pa je ćemo u nastavku pokušati hipotetički nadopuniti naš sistem. Krenimo sa hipotetskim razmatranjem, odnosno šta nama trenutno fali ovdje kako bismo konkretno ovaj pattern mogli primijeniti na našem sistemu. Neka je potrebno sortirati studente na predmetu po broju njihovih ostvarenih bodova(kroz zadaće i ispite i ostale aktivnosti). To je moguće uraditi na više načina, od kojih su neki jednostavniji, a drugi kompleksniji, analogno neki su brži a neki su sporiji. Zašto ne bismo omogućili klijentu da bira način na koji želi da izvrši to sortiranje(odnosno da mu omogućimo više načina na koje može sortirati te studente). Da bismo ovo omogućili moramo imati neku kontekstualnu klasu i nek se ona zove **StudentContext** koja će interfejsu ***ISortAlgoritmi*** proslijediti neophodne podatke(odnosno u ovom slučaju samu kolekciju studenata koju je potrebno sortirati). Ovaj interface će se sastojati od metode sortiraj() koju će implementirati više klasa(npr klase: **MergeSort**, **ShellSort**, **BubbleSort** itd) na različite načine. StudentContext bi kao privatni atribut imao objekat tipa ISortAlgoritam gdje bi se pomoću polimorfizma mogli pozivati razni algoritmi sortiranja. Također klasa StudentContext bi posjedovala metodu koja bi omogućavala da se promijeni ova strategija npr promijeniAlgoritam(id: int). Znači omogućeno je da klijent(programer) u kodu može birati strategiju sortiranja studenata u ovisnosti kakve mu performanse trebaju(da li to sortiranje treba biti brzo itd...), odnosno po želji može da odabere razne

načine koji ga vode do istog cilja. Ovime završavamo hipotetičko razmatranje ovog patterna.

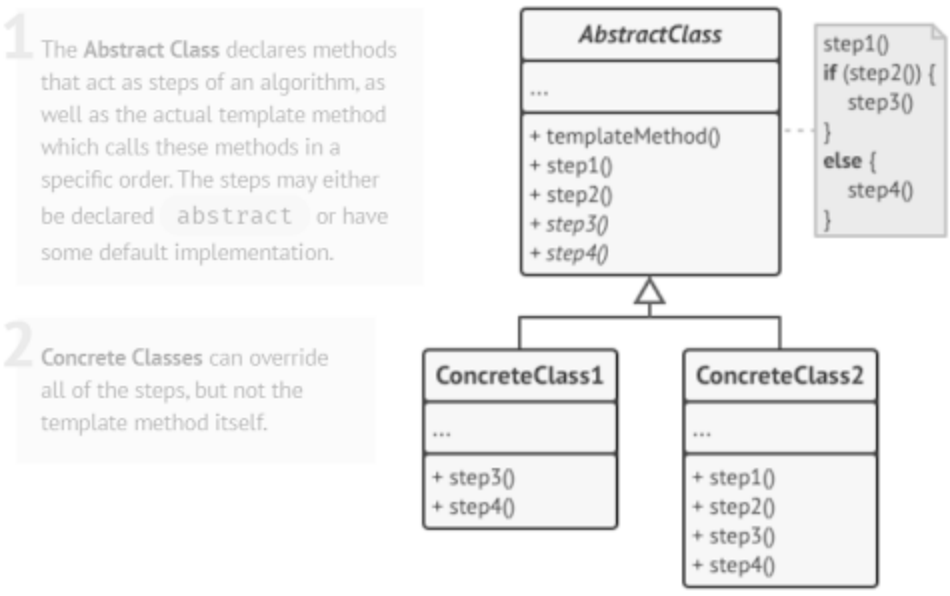
State pattern - je dinamička verzija Strategy paterna. Objekat mijenja način ponašanja na osnovu trenutnog stanja. Postiže se promjenom podklase unutar hijerarhije klasa.

Ovaj pattern iz sličnih razloga(jednoznačnih procedura) nećemo koristiti u našem sistemu jer bi ga to previše zakomplikovalo. Razmotrimo hipotetsko mjesto upotrebe ovog patterna. Zamislimo da studentska služba ima više „stanja“ rada, odnosno što se dešava u normalnom životu da studentska služba mijenja svoj „ritam“ rada onda kada se pojavi više gužve sa zahtjevima i to. Eh sada zamislimo da studentska služba u zavisnosti u kom stanju je rada, na različite načine obrađuje studentske zahtjeve(tako da zahtjevi imaju neku oznaku kao npr „hitan“ ili „normalan“, odnosno da zahtjeve nekako možemo razlikovati po toj „hitnoći“). Neka imamo neku klasu **StudentskaContext** koja kao svoj atribut ima trenutni broj zahtjeva koji čekaju na obradu, i neki svoj limit. Ako broj zahtjeva pređe određeni limit tada studentska treba da promijeni svoj režim obrate tih podataka odnosno da prvo na red dolaze oni zahtjevi koji su označeni sa „hitno“ sve dok se broj takvih zahtjeva ne svede na 0 ili se ukupan broj zahtjeva ne spusti ispod dozvoljenog limita. Također ista klasa ima i neku kolekciju svih zahtjeva. Moramo imati i interface ***IStanje*** koji definiše metodu obradi(StudentskaContext context) i kog će implementirati dvije klase: **HitnoStanje** i **NormalnoStanje**. **HitnoStanje** će u svojoj implementaciji ove metode uporediti limit sa trenutnim brojem zahtjeva te ako je ukupan broj zahtjeva manji od limita prebacit će se u **NormalnoStanje** koje zahtjeve obrađuje redom onakvi kakvi su, dok suprotno implementacija ove metode u klasi **NormalnoStanje** provjerava da li je ukupan broj zahtjeva prema studentskoj veći od određenog limita, te ako jest prebacuje se u **HitnoStanje** koje obrađuje samo zahtjeve sa naznakom „hitno“ sve dok se ne ispuni ranije navedeni uslov koji će ponovno rad(obradu zahtjeva) studentske vratiti u **NormalnoStanje**. Vidimo da se načini obrade zahtjeva dinamički

mijenjaju od situacije u kojoj se sistem trenutno nalazi pa je ovaj pattern dobro pojašnjen, odnosno mogao bi se na ovaj način implementirati/iskoristiti, ali kako je već navedeno, sistem nije toliko komplikovan te nam ovo zaista ne treba, pa ovdje možemo završiti sa hipotetičkim razmatranjem ovog patterna.

Template Method pattern - Omogućava izdvajanje određenih koraka algoritma u odvojene podklase. Struktura algoritma se ne mijenja - mali dijelovi operacija se izdvajaju i ti se dijelovi mogu implementirati različito.

Ovaj pattern možemo primijentiti u našem sistemu bez nekih većih izmjena. Očigledno je da će nam u nekom dijelu/vremenu biti potrebno sortirati npr studente po nekom kriteriju, profesore itd... Zašto bi duplicirali naš kod kada on očigledno ima istu bazu, samo bi se razlikovao u kriterijima po kojima se sortira dok je sami algoritam sortiranja isti.



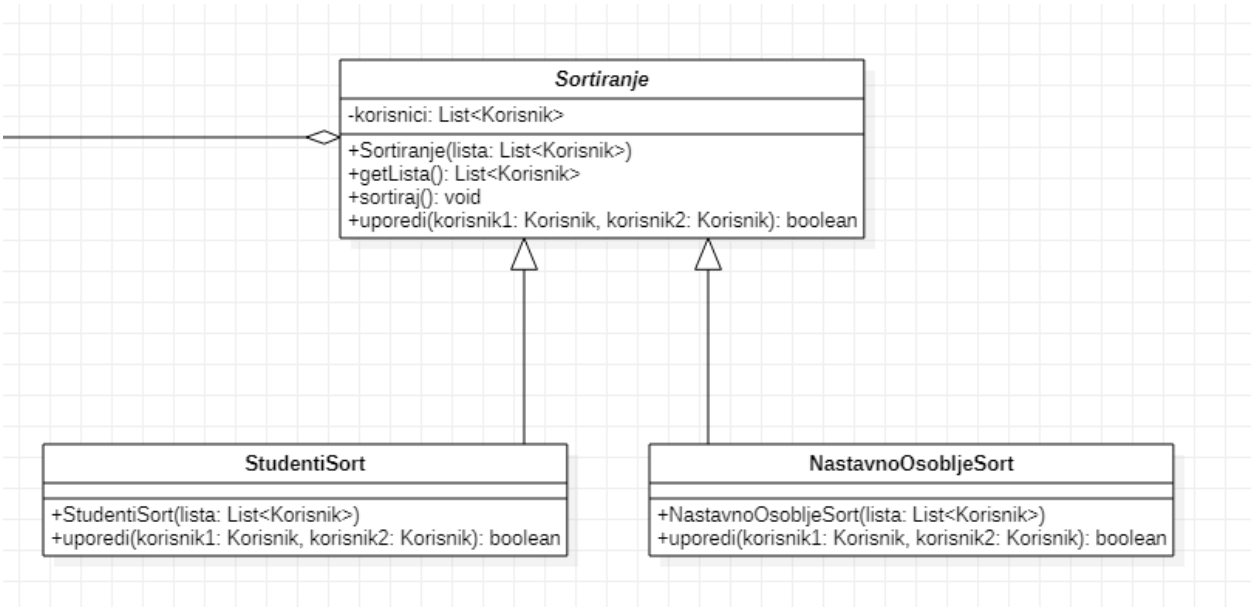
Tako da ćemo u našem sistemu imati sljedeće klase:

Sortiranje – Apstraktna klasa koja će definisati templateMethod (odnosno implementirat će sami algoritam sortiranja koji neće biti „preklopljen“ u izvedenim klasama). Pored templateMetoda definisat će i sporedne operacije koje će za razliku od templateMethod-a biti overrideovane u izvedenim klasama u zavisnosti, konkretno u ovom slučaju, kako želimo da sortiramo studente, nastavno osoblje odnosno po kom kriteriju. Naša templateMethod-a će se zvati sortiraj koja će primit listu u našem slučaju Korisnika(jer su oni ti koji se zapravo trebaju sortirati na neki način). Pored metode sortiraj imat ćemo i metodu uporedi koja će biti samo definisana u ovoj klasi, a implementirana u izvedenim klasama na različite načine. Ova klasa će kao svoj privatni atribut imati listu korisnika.

StudentiSort – klasa koja je nasljeđuje klasu **Sortiranje** i override-a metodu poređenja i to na način da će se studenti porediti po trenutnom broju položenih predmeta, pa onda po prosjeku.

NastavnoOsobljeSort – klasa koja također nasljeđuje klasu **Sortiranje** i override-a metodu poređenja, ali na način da će se sada nastavno osoblje drugačije porediti u odnosu na studente. Konkretno će se porediti po tome koji profesor ima više predmeta(odnoso veću normu).

Ovime je omogućeno da klijent može instancirati neku od ove dvije klase šaljući joj kolekciju koju mora sortirati te nad njihovom instancom pozvati metodu sortiraj koja će sortirati tu kolekciju u zavisnosti koji objekti se nalaze u njoj. Također bitno je napomenuti da će biti onemogućeno instancirati objekat tipa StudentiSort sa listom objekata NastavnoOsoblje i slično, jer onda ovo sve nema smisla. Konkretna primjena ovog patterna u našem sistemu izgleda ovako, s kojom ćemo završiti razmatranje ovog patterna



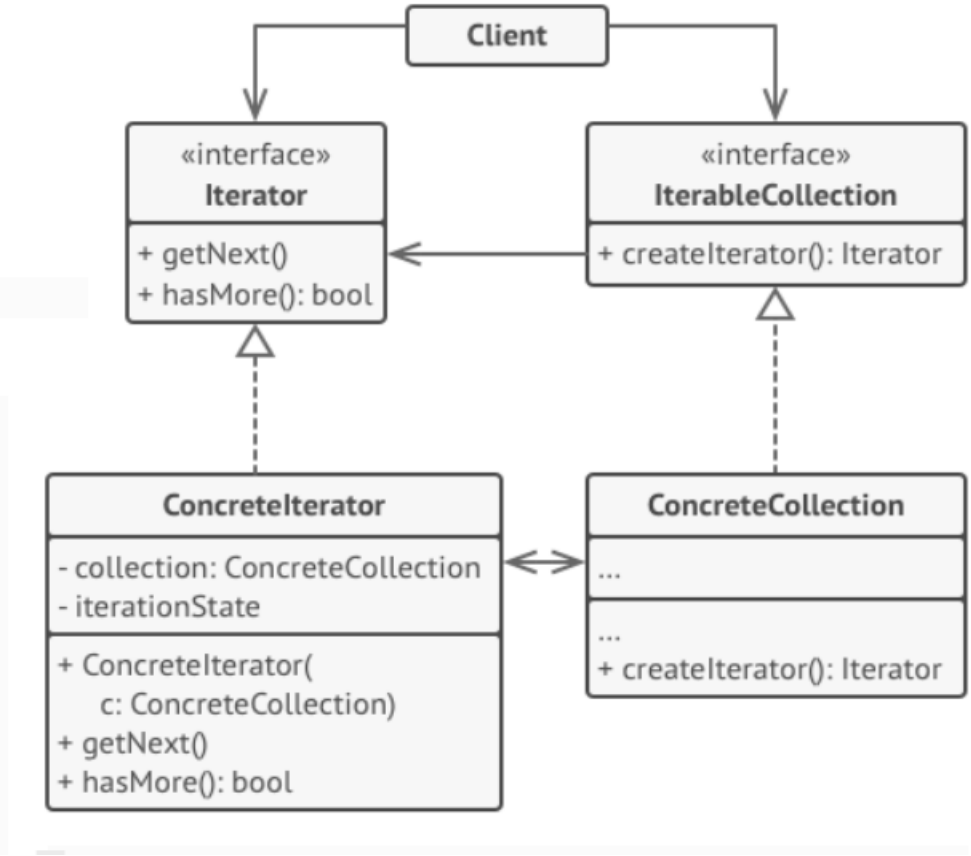
Primjer koda za ovaj pattern bi u nekoj od kontrolerskih procedura glasio ovako:

```
StudentiSort srt = new StudentiSort(this.getStudenti());
srt.sortiraj(); /*lista studenata je sortirana po kriteriju po kojem
se studenti uspoređuju, i sad možemo npr ispisati tu sortiranu listu*/

foreach(Korisnik k in srt.getLista()) k.ispisi(); /*ovdje
pretpostavljamo da će korisnik imati implementiranu metodu ispisa,
odnoso format ispisa*/
```

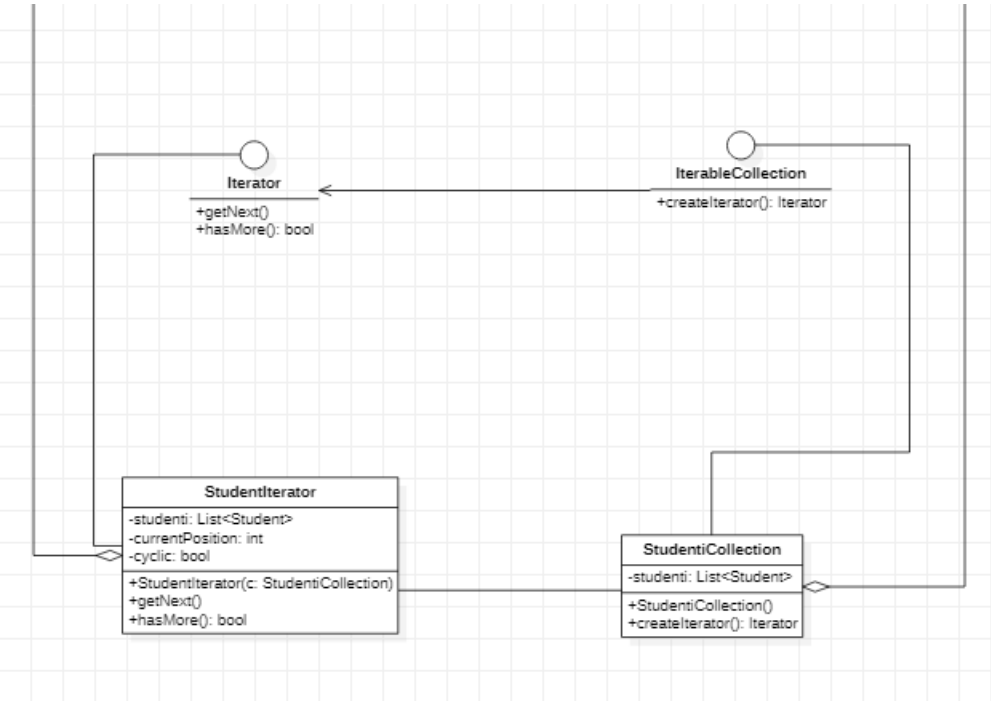
Iterator pattern – Pattern ponašanja koji se koristi za prolaženje kroz elemente kolekcije bez izlaganja strukture iste. Dakle, pruža nam pristup elementima, bez obzira na to kako je kolekcija strukturirana, da bi se oni mogli koristiti u ostalim dijelovima koda.

Ovaj pattern se može primijeniti u našem sistemu za iteriranje kroz listu svih studenata. To iteriranje će se vršiti po različitim kriterijima. Struktura patterna u općem slučaju ima sljedeći izgled:



U našem slučaju, ConcreteCollection će biti klasa koja sadrži listu svih studenata i nazvat ćemo je StudentiCollection. Ona mora naslijediti interfejs IterableCollection u kojem je definisana metoda createIterator, koju naravno i implementira. ConcreteIterator klasa je u našem slučaju klasa StudentiIterator. Upravo će metoda createIterator() (implementirana u klasi StudentiCollection) postavljati sve atribute klase StudentiIterator. Klasa StudentiIterator mora sadržavati sve neophodne podatke za iteriranje kroz kolekciju (indeks trenutnog elementa, sljedećeg, broj elemenata i sl.), tako da je omogućeno da se kroz istu iterira sa više nezavisnih iteratora. Ova klasa sadrži i instancu kolekcije studenata koja je stvorena dubokim kopiranjem. Klasa StudentIterator nasljeđuje i interfejs Iterator u kojem su definisane metode getNext() i hasMore() (koje daju različite rezultate za različite vrste listi). U metodu createIterator(), kao što je je već spomenuto, će se kao parametri slati sve vrijednosti atributa klase StudentIterator, te ako za atribut cyclic stavimo da je true, onda će metoda getNext() vraćati prvi element liste

ukoliko smo došli do kraja, a hasNext() će uvijek vraćati true, naravno ukoliko lista ima barem jedan element.



Observer pattern - uloga observer pattern-a je da uspostavi relaciju između objekata tako kada jedan objekat promijeni stanje drugi zainteresirani objekti se obavještavaju.

Ovaj pattern nećemo koristit u našem sistemu, pa ćemo razmotriti samo hipotetsko mjesto upotrebe ovog patterna, pa zatim i objasnit zažto ga nećemo koristiti.

Posmatrajmo klasu PredmetZaStudenta koji ima atribut ocjena koji će uvijek inicijalno biti postavljen na 5. Tako da bismo mogli dodati akciju da se nakon promjene vrijednosti atributa ocjena tj. nakon što profesor unese studentu ocjenu, pošalje automatizovana Poruka od profesora ka studentu sa nekim tekstom tipa “Upisana ocjena iz predmeta Objektno orijentisana analiza i dizajn 6”.

Upravo za ovu situaciju bi mogli koristit observer pattern.

Da bismo to izveli prvo nam je potreban IObserver interface sa metodom update koja će formirati interfejs između klasa PredmetZaStudenta i Profesor,

gdje PredmetZaStudenta ima ulogu Subject klase, a klasa Profesor Observer klase, što znači da će klasa Profesor implementirati IObserver intefejs.Još jedino što nam treba je event mehanizam koji će pozivati update operaciju za observera(profesora na tom predmetu).

U toj update operaciji će se i slati već spomenuta poruka of profesora ka studentu. Metode registerObserver i removeObserver nam ne trebaju jer ce svaki predmet imati samo po jednog observera, tj zaduženog profesora na predmetu.

Trenutno smo mišljenja da bi ova opcija bila “višak” na našem sistemu jer smatramo da ovo nije najpogodniji način za obavještavanje studenta o upisanoj ocjeni. Zaključujemo da pattern nije adekvatan za naš sistem u ovome trenutku.