Univerzitet u Sarajevu
Elektrotehnički fakultet u Sarajevu
Odsjek za računarstvo i informatiku

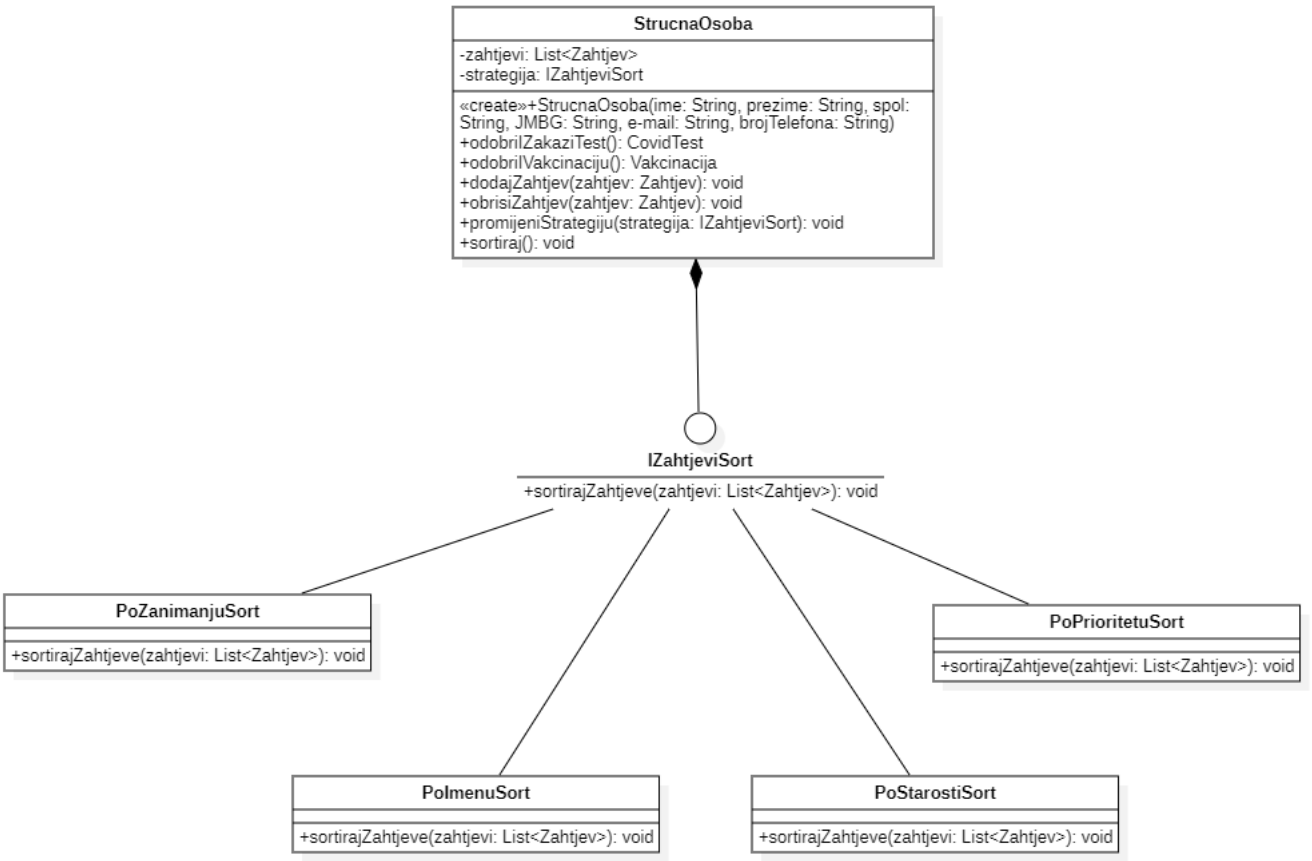
Paterni ponašanja
Imunizacija '21

Objektno orijentisana analiza i dizajn

Naziv grupe: Schpritzer
Članovi: Muhamed Borovac
Eldar Čivgin
Dženan Nuhić
Benjamin Pašić

1. Strategy patern

Uloga **Strategy patern**-a je da izdvaja algoritam iz matične klase i uključuje ga u posebne klase. Koristi se kada postoje različite primjenljivne strategije za neki problem. On omogućava korisniku izbor jednog algoritma iz familije algoritama za korištenje.
U našem sistemu, možemo primjeniti ovaj paten u klasi *StrucnaOsoba*. Ova klasa sadrži privatni atribut zahtjevi koji je tipa *List¡Zahtjev¿*. Mi želimo da omogućimo korisniku da sortira ove zahtjeve na različite načine (po imenu, po starostnoj dobi, po zanimanju, itd.), kako bi mu se prvo prikazali oni zahjevi, koji su mu u tom momentu najvažniji. Ovo možemo uraditi na sljedeći način: Prvo ćemo kreirati interfejs *IZahtjeviSort*, koji će sadržavati metodu *sortirajZahtjeve()*, koja kao parametar prima listu zahtjeva. Nakon toga ćemo kreirati klase *PoImenuSort*, *PoZanimanjuSort*, *PoStarostiSort*, *PoPrioritetuSort*, itd. Svaka od ovih klasa će implementirati interfejs *IZahtjeviSort*, odnosno njegovu metodu *sortirajZahtjeve()*, na odgovarajući način. Sada ćemo u klasi *StrucnaOsoba* kreirati privatni atribut *strategija*, koji je tipa *IZahtjevSort*. Nakon toga u klasu *StrucnaOsoba* ćemo dodati i metodu *sortiraj()* koja će samo pozivati metodu *sortirajZahtjeve()* atributa strategija i na taj način sortirati listu zahtjeva. Trebamo još da implementiramo metodu *promijeniStrategiju()* koja prima *IZahtjeviSort* objekat i mijenja atribut strategija. Na ovaj način smo postigli da korisnik ima opciju da promijeni redosljed zahtjeva koji mu se prikazuju na način koji mu u tom momentu najviše odgovara. Također, ako nekada bude potrebno, veoma lako bi bilo dodati novu strategiju sortiranja čime postižemo dobar reusability koda.
Nakon primjenjivanja Strategy patern-a, naš dijagram klasa će izgledati ovako:

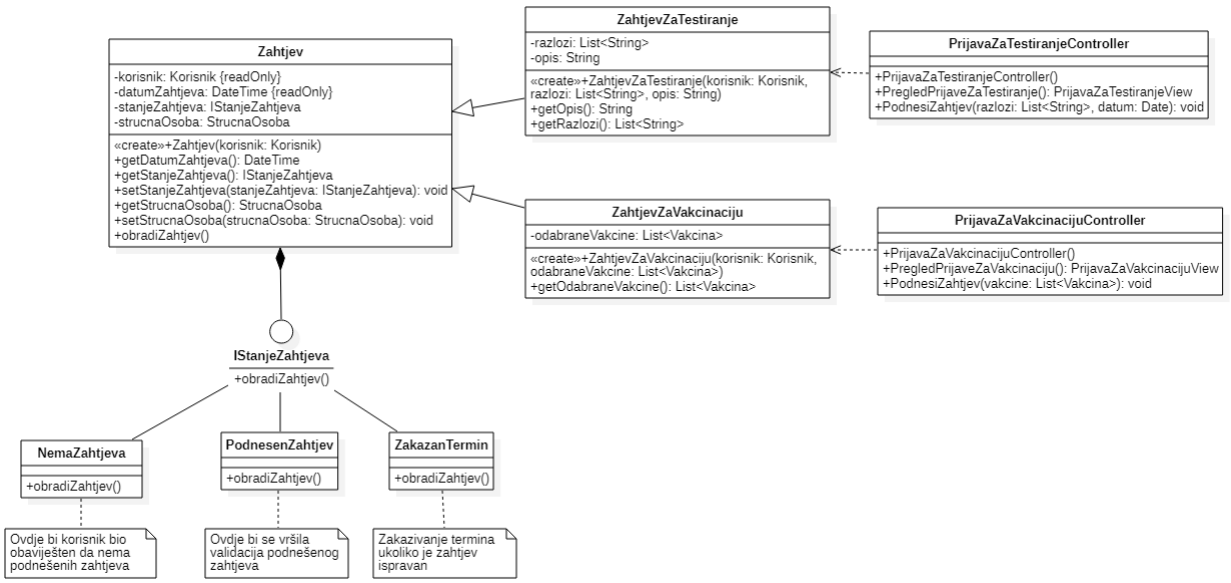


Slika 1: Strategy patern

2. State patern

State patern omogućava objektu da promijeni svoje ponašanje, od kojih zavisi njegovo ponašanje. Nakon promjene stanja, objekat se počinje ponašati kao da je promijenio klasu.
Objekat mijenja način ponašanja na osnovu trenutnog stanja.
State patern predstavlja dinamičku verziju Strategy patern-a.
Postiže se promjenom podklase unutar hijerarhije klasa.
Podržava open-closed princip.
Ovaj patern u našem sistemu možemo primijeniti na sljedeći način:
Kada korisnik podnese zahtjev, nalazit će se tri moguća stanja: *Zahtjev podnesen*, *Zahtjev validiran* i *Zakazan termin*. U našem sistemu ćemo dodati interfejs *IStanjeZahtjeva* te klase *NemaZahtjeva*, *PodnesenZahtjev* i *ZakazanTermin* koje implementiraju ovaj interfejs. U klasi *Zahtjev* ćemo zamijeniti atribut *odobren* (tipa Boolean) sa atributom *stanjeZahtjeva* (tipa *IStanjeZahtjeva*).

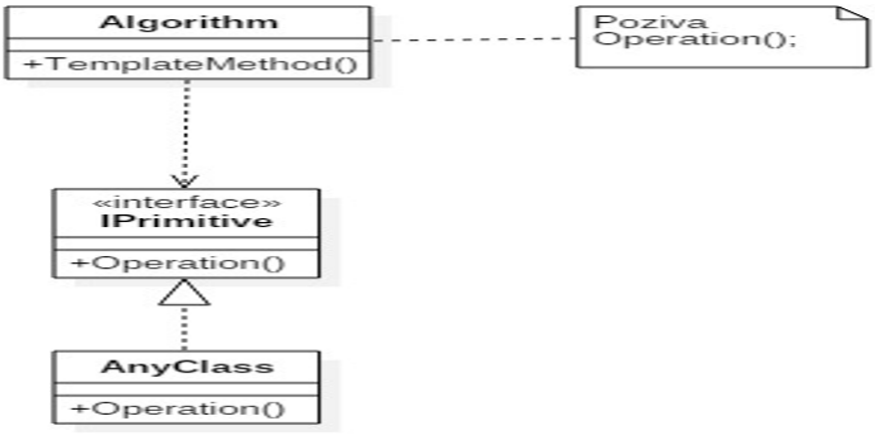
Nakon primjenjivanja State patern-a, naš dijagram klasa će izgledati ovako:



Slika 2: State patern

3. Template method patern

Svrha **Template method patern**-a je da omogući izdvajanje određenih koraka algoritma u odvojene podklase. Samim tim, struktura algoritma se ne mijenja, mali dijelovi operacija se izdvajaju i ti se dijelovi mogu implementirati različito.
Ovaj patern se sastoji od klase *Algorithm* koja uključuje metodu koja izdvaja dijelove svojih operacija u druge klase. Tu se nalazi interfejs *IPrimitive* koji definira operacije koja pomenuta metoda izdvaja u druge klase te *AnyClass* koja implementira interfejs *IPrimitive*.

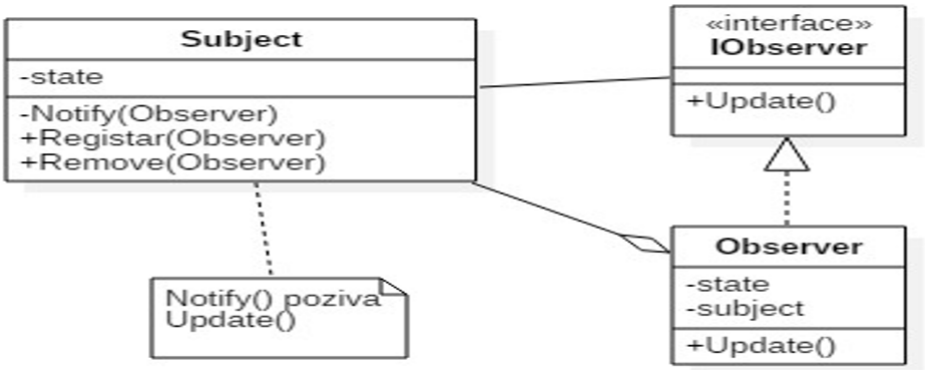


Slika 3: Template method patern

U našem sistemu, ovaj patern možemo primjeniti u sklopu klase *CovidTest* koja bi predstavljala našu *Algorithm* klasu, a njene metode *getInfo* i *getOpisTesta* bi predstavljale metode koje želimo odvojiti u podklase. Za svaki tip *CovidTest*-a mi bi imali određenu podklasu koja bi implementirala pomenute izdvojene metode.

4. Observer patern

Uloga **Observer patern**-a je da uspostavi relaciju između objekata tako kada jedan objekat promijeni stanje drugi zainteresirani objekti se obavještavaju.



Slika 4: Observer patern

S obzirom da doze nekih vakcina nisu beskonačne ili su bolje vakcine namijenjene ugroženom dijelu stanovništva, osoba nekada nije u mogućnosti da se vakciniše željenom, već najboljom mogućom opcijom od dostupnih vakcina. Tako da na neki način želimo da obavještavamo osobe koje nestrpljivo čekaju

na svoju prvu dozu da li su obnovljene zalihe vakcina, odnosno da li ima dovoljno doza vakcine kojom bi se osobe željeli vakcinisati. Da bismo postigli ovo, upravo možemo iskoristiti Observer patern.
U naš sistem ovaj patern bi se mogao implementirati na način tako da osobe koje su prijavljene i čekaju na svoju prvu dozu željene vakcine dobivaju obavijesti ukoliko neka vakcina postaje dostupna ili ukoliko je eventualno na tržište stigla neka najnovija i naučno dokazano efikasnija vakcina protiv Covid-19. Pri svakoj promjeni u ovom dijelu sistema, zainteresirane osobe koje se trebaju vakcinisati se obavještavaju.

5. Iterator patern

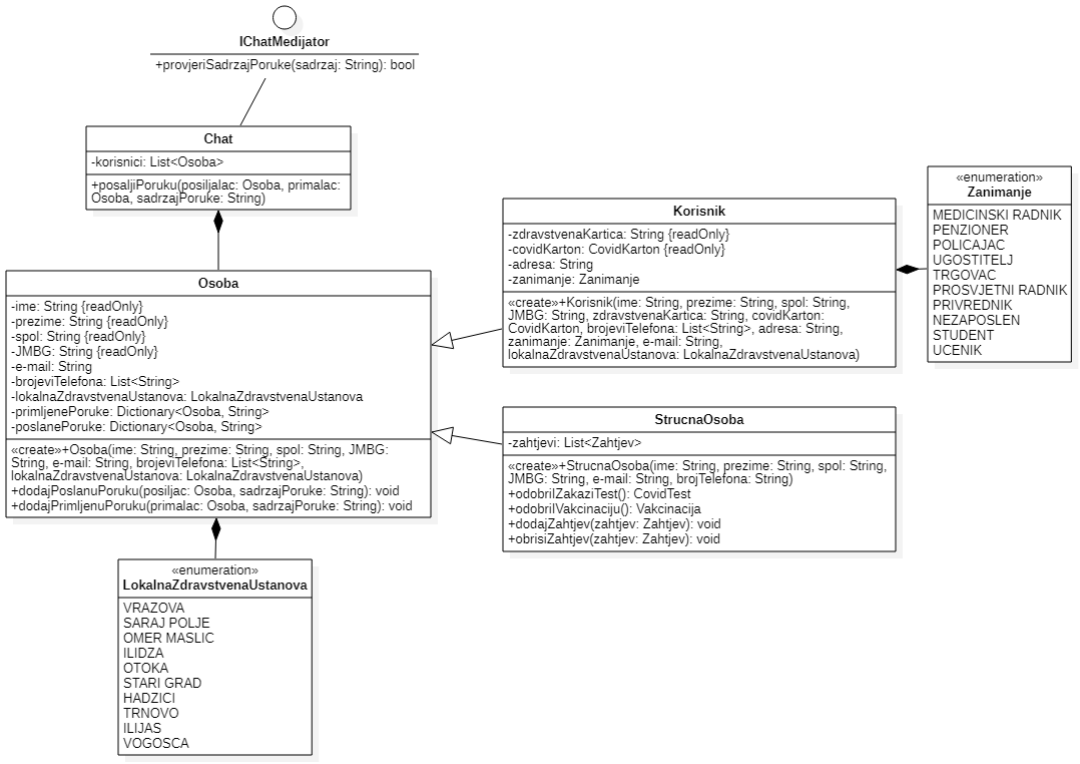
Iterator patern omogućava sekvencijalni pristup elementima kolekcije bez poznavanja kako je ta kolekcija struktuirana.
U našem sistemu, ovaj patern možemo iskoristiti u klasi *CovidKarton* za kretanje kroz listu bolesti koju u sebi sadrži atribut *bolesti*. Prvo ćemo kreirati interfejs *IKreatorBolestiIteratora* koji će sadržavati metodu *kreirajIterator()*, koja će primati vrstu iteratora (*int*) i odgovarajuću listu (u ovom slučaju listu bolesti). Nakon toga ćemo kreirati interfejs *IBolestiIterator* sa metodom *dajSljedecuBolest()* koja vraća *Bolest* objekat. Iz *IBolestiIterator* interfejsa ćemo naslijediti razne klase, koje će predstavljati načine na koje korisnik može da se kreće kroz listu. Možemo npr. imati *AlergijeFirstIterator*, *BolestiFirstIterator*, *HronicneBolestiFirstIterator*, *RecentBolestiFirstIterator*, *AbecedniIterator*, *RandomIterator*, itd. Svaka od ovih klasa će implementirati metodu *dajSljedecuBolest()*, na svoj odgovarajući način. Ove klase u sebi mogu sadržavati listu bolesti, trenutnu bolest i sve ostale atribute koji bi im mogli biti korisni. Na kraju, u klasu *CovidKarton* ćemo dodati atribut *iteratorBolesti* tipa *IBolestiIterator*, te ćemo naslijediti *IKreatorBolestiIterator*-a, što znači da ćemo implementirati metodu *kreirajIterator()*.
Ovim smo omogućili korisniku da se on kreće kroz listu bolesti, na koji god on način želi. On samo pozove metodu *kreirajIterator()* i proslijedi joj vrstu iteratora koju želi. Također, on ne mora da zna kako je ova kolekcija struktuirana, odnosno kako je sortirana, koji se tip podataka koristi interno itd.

6. Chain of responsibility patern

Chain of responsibility patern predstavlja listu objekata, ukoliko objekat ne može da odgovori prosljeđuje zahtjev narednom u nizu.
U našem sistemu, ovaj patern bi mogli iskoristiti kada korisnik krene napraviti zahtjev za vakcinaciju (kroz klasu *ZahtjevZaVakcinaciju*). Tu bi imali jednu klasu *KreiranjeVakcinacije* sa metodom *kreiraj()* koja bi prvo pozvala klasu *ZahjtevZaVakcinaciju* koja bi kreirala jedan zahtjev a onda bi se po odobravanju istog zahtjeva, napravila i instanca klase *Vakcinacija*, te bi se konačno ta vakcinacija ažurirala u sklopu *CovidKarton*-a korisnika.

7. Medijator patern

Medijator patern omogućava reduciranje ovisnosti između objekata. Ovaj patern ograničava direktnu komunikaciju između objekata i forsira ih da kolaboriraju samo preko medijator objekta.
Umjesto za direktno povezujemo veliki broj objekata, koristimo *medijator*, koji je zadužen za njihovu komunikaciju. Kada neki objekat želi poslati poruku drugom objektu, on šalje poruku medijatoru, a medijator zatim prosljeđuje tu poruku drugom objektu.
U našem sistemu možemo primijeniti ovaj patern za realizaciju chat-a. Dodali bismo interfejs *IChatMedijator*. Ovaj interfejs bi sadržavao metodu *provjeriSadrzajPoruke* koja bi provjeravala da li poslana poruka sadrži neprimjereni sadržaj. U slučaju da poruka sadrži neprimjereni sadržaj, odbacuje se, a u suprotnom se šalje.
Klasa *Chat* bi implementirala ovaj interfejs te bi u sebi imala listu svih korisnika koji koriste chat, te metodu *posaljiPoruku* za slanje poruke. Unutar klase *Osoba* bismo čuvali sve poslane i primljene poruke za jednog korisnika.
Nakon primjenjivanja Medijator patern-a, naš dijagram klasa će izgledati ovako:



Slika 5: Medijator patern