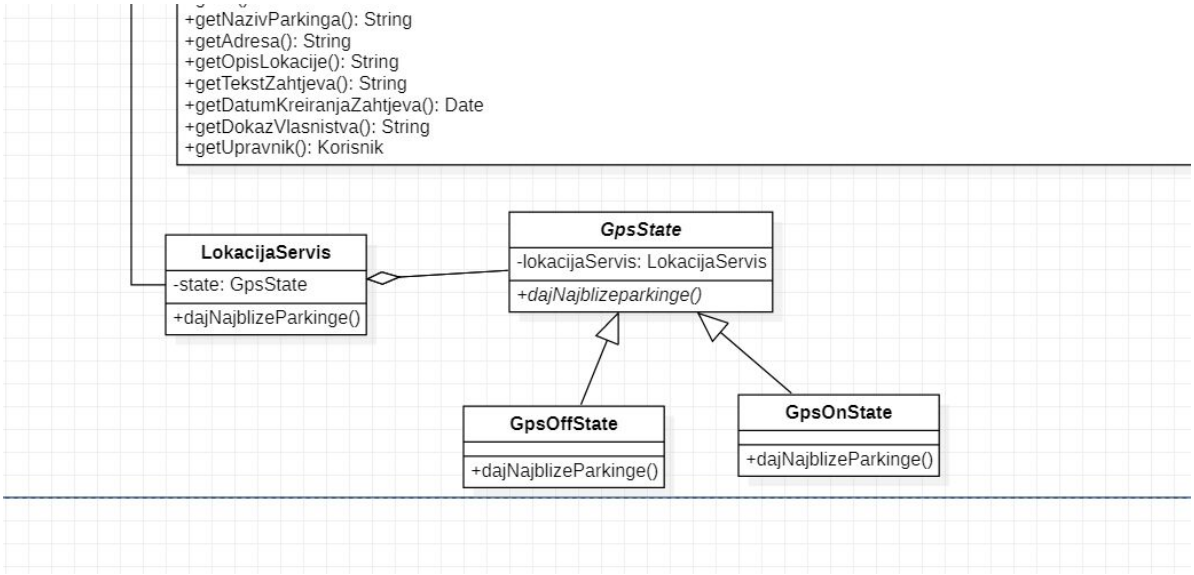
Paterni ponašanja

Strategy pattern

- ***Izdvaja algoritam iz matične klase i uključuje ga u posebne klase. Omogućava nam da definiramo porodicu algoritama, stavimo ih u posebnu klasu i učinimo njihove objekte izmjenjivim.***
  Kako u našem sistemu trenutno nemamo procesa koji se mogu vršiti na više različitih načina, tj. nemamo više algoritama koji rješavaju neki problem, u naš sistem trenutno ne možemo implementirati ovaj patern. Ukoliko bismo odlučili da omogućimo sortiranje parkinga pri čemu bi korisnik mogao birati koji metod sortiranja da koristi zasigurno bi iskoristili ovaj patern. Neki algoritmi sortiranja su kompleksniji, a neki jednostavniji i nekada je korisniku potrebno da sam bira kako da sortira. Da bi mogli implementirati Strategy patern potrebna je klasa ParkingContext i interfejs ISortAlgoritmi. Preko klase ParkingContext klasa Parking nam daje kontekstualne informacije za algoritme koji nasljeđuju interfejs ISortAlgoritmi. Interfejs ISortAlgoritmi definira metodu sortiraj() koja je zajednička za sve algoritme. Ovaj interfjes bi nasljeđivale sve klase koje vrše sortiranje na različite načine npr. BubbleSort, MergeSort, HeapSort (ove klase implementiraju konkretne algoritme). Ova funkcionalnost omogućava korisnika da sam bira koji metod sortiranja želi u zavisnosti od njegovih potreba.

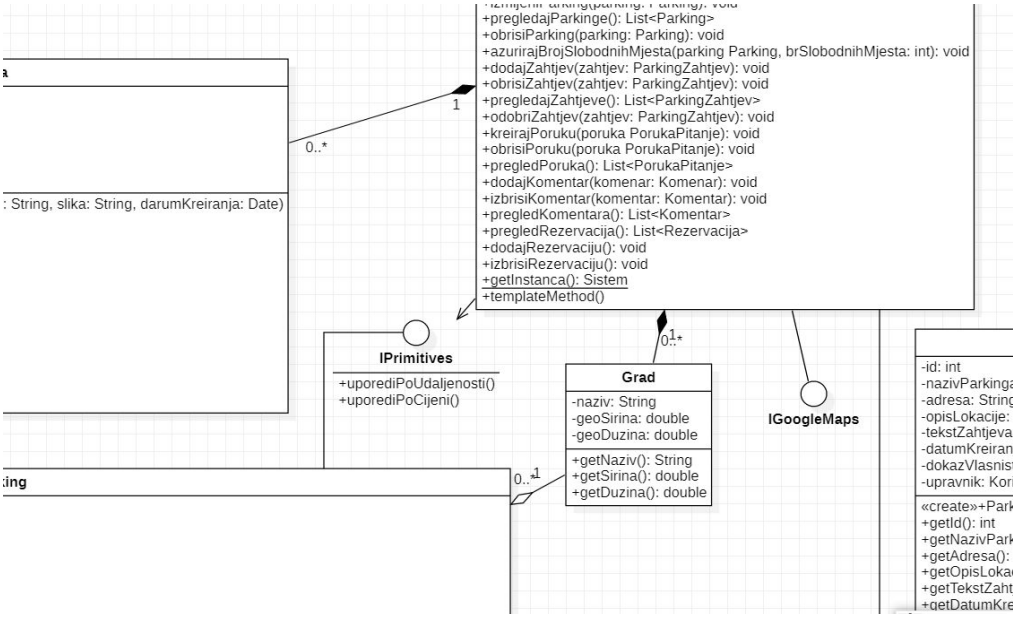
State pattern

- ***Mijenja način ponašanja objekata na osnovu trenutnog stanja.***
  Ovaj pattern je jako koristan i iskoristit cemo ga u vise slucajeva u nasem sistemu gdje zelimo da se sistem ponasa u zavisnosti od stanja nekog objekta. Ovdje ćemo navesti jedan jednostavan primjer upotrebe. Naime, želimo da kada korisnik zatraži da mu se prikažu najbliži parkinzi njegovoj trenutnoj lokaciji izvrši akcija u odnosu na to da li je njegov GPS upaljen tj. da li korisnik želi da podijeli njegovu trenutnu lokaciju. U slučaju da nije, onda se prikazuje prompt gdje se traži od korisnika da uključi dijeljenje lokacije, u suprotnom sistem prikazuje lokacije.
  Neka imamo klasu **LokacijaServis** (context) koja između ostalog sadrži atribut/field **state** tipa **GpsState** te metodu **dajNajblizeParkinge()** koja se poziva iz trenutnog state-a. Apstraktna klasa **GpsState** sadrži privatni atribut **lokacijaServis** koja je referenca na istoimenu klasu koja se postavlja kroz konstruktor. Takođe ova klasa sadrži apstraktnu metodu **dajNajblizeParkinge()** koju će naslijediti i implementirati **GpsState** subklase. Te subklase su **GpsOffState** i **GpsOnState**. Defaultno stanje tj privatni atribut klase **LokacijaServis** će se postaviti na **GpsOffState**. Nakon što klijent zatraži najbliže parkinge, pozvat će se metoda **dajNajblizeParkinge()** trenutnog **GpsState** objekta. U slučaju da je to **GpsOffState**, zatražit će se dijeljenje lokacije. Ako korisnik prihvati, state se prebacuje u **GpsOnState** pomoću metode **changeState()** unutar klase **LokacijaServis** te se ponovo poziva metoda i ovaj put vraća najbliže lokacije te klasa **LokacijaServis** ostaje u stanju **GpsOnState** dok korisnik ne ugasi dijeljenje lokacije.



TemplateMethod pattern

- ***Omogućava algoritmima da izdvoje pojedine korake u podklase.***
  Ovaj patern ćemo iskoristiti za naš sistem s obzirom da postoje situacije u kojima nam može biti od koristi. U predavanju je dat primjer upotrebe ovoga patterna u svrhu sortiranja osoba. Mi smo odlučili ovaj pattern iskorsitit u svrhu sortiranja parkinga. Naime, korisniku možda nije nužno potreban najbliži parking, već najjeftiniji parking u njegovoj blizini. S toga treba omogućiti i sortiranje po cijenama. To ćemo uraditi tako što ćemo prvo dodati interfejs IPrimitives koji ima metode uporediPoUdaljenosti() i uporediPoCijeni(). Ovaj interfejs će implementirati klasa Parking. Dalje, klasa Sistem predstavlja klasu Algorithm, tako da ćemo joj dodati metodu templateMethod(), unutar koje će se vršiti sortiranje nad parkinzima shodno odabranom kriteriju.



Iterator pattern

- ***Iterator patern omogućava sekvencijalni pristup elementima u kolekciji a da se pri tome ne mora znati njena struktura. Pored toga, patern omogućava filtriranje elemenata na različite načine.***

  U našem sistemu nismo iskoristili ovaj patern. Ovaj patern bi mogli iskoristiti za kretanje kroz liste u klasi Sistem. Ukoliko bismo željeli da omogućimo iteriranje foreach petljom kroz listu parkinga na način da nam prvi element bude onaj parking koji je najposjećeniji iskoristili bi ovaj patern. Iterator klasa bi trebala sadržavati listu<double> posjecenost i int trenutniIndex kao atribute. Posjećenost bi bio neki postotak koji se računa u odnosu na broj korisnika parkinga i koliko parking ima mjesta. U klasu Sistem bi dodali instancu klase Iterator. Osnovne metode koje treba implementirati su getNext() i hasMore(). Metoda getNext u klasi Iterator bi nam vraćala prvi element u listi koji ima veću posjecenost od trenutnog elementa. Klasa Sistem bi implementirala interfejs IterableCollection koji ima metodu createIterator(). Ova metoda će nam omogućiti da klasi iterator proslijedimo listu prihoda i indeks od kojeg započinjemo iteraciju.

Observer pattern

- ***Uspostavlja relaciju između objekata takvu da kad se stanje jednog objekta promijeni svi vezani objekti dobiju informaciju***

  Ovaj pattern cemo iskoristiti tako sto cemo napraviti jednu centralnu dispecersku klasu **EventManager** (Publisher) koja ce imati listu svih listenera (**hashMap<eventType, EventListener>**) te metode **subscribe(eventType, listener), unsubscribe(eventType, listener)** i **notify(eventType, data)**. Metode subscribe() i unsubscribe() ce da dodaju odnosno brisu listenere koji slusaju event/dogadjaj sa imenom eventType. Metoda notify() ce da obavijesti sve listenere koji slusaju tip **eventType** uz slanje podataka '**data'**. Dakle bilo koja klasa u kojoj trebamo da dispecujemo neke evente nece naslijediti **EventManager** nego uraditi kompoziciju tj imat ce ga kao atribut **eventManager**. Sa druge strane imamo interfejs **EventListener** koji posjeduje samo jednu **update()** metodu koja ce se pozivati iz **notify()** u **EventManageru**.

  Ukratko receno, svaka klasa koja zeli da uradi dispatch eventa, kao privatni atribut ima referencu na **EventManager,** dok svaka klasa koja zeli da slusa evente mora da implementira **EventListener** interfejs. Jedan od primjera upotrebe u nasem sistemu smo pronasli u metodi koja se nalazi u **Parking** kontroleru **azurirajBrojSlobodnihMjesta()** koja nakon sto azurira br. slob. mjesta, mora da obavijesti sve korisnike sistema o novom stanju u **realnom vremenu**. Dakle, zelimo da kada upravnik parkinga promijeni broj slob. mjesta, korisnici odmah mogu vidjeti promjene u samom **ParkingView** bez ponovnog slanja zahtjeva na server za najnovijim podacima.