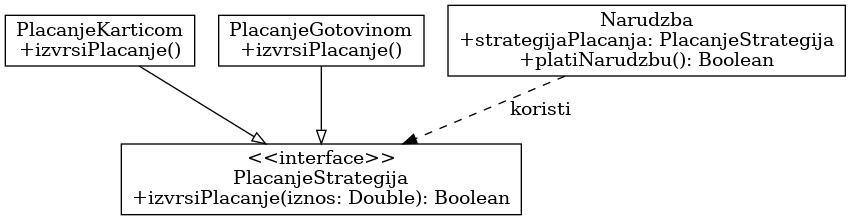
# **Paterni ponašanja**

## **Strategy pattern**

Potreba za **strategy patternom** javlja se kada želimo omogućiti različite načine ostvarivanja istog cilja, bez da mijenjamo osnovnu logiku sistema. U slučaju našeg online kafića, taj cilj je izvršavanje plaćanja narudžbe, a korisnicima je omogućeno da biraju između dva metoda plaćanja — plaćanje karticom i gotovinom pri preuzimanju u radnji.

Svaki od ovih načina plaćanja funkcioniše na drugačiji način, ima svoje specifične korake i tehničke zahtjeve. Na primjer, plaćanje karticom uključuje autorizaciju i verifikaciju putem treće strane, dok je gotovinsko plaćanje jednostavno evidentiranje da je iznos primljen. Korištenjem strategy patterna, svaki način plaćanja može se implementirati kao posebna strategija, bez potrebe da se osnovna struktura narudžbe mijenja.

Umjesto da se u glavnu logiku narudžbe ugrađuju svi mogući načini plaćanja, sistem koristi zajednički interfejs koji definiše metodu za izvršavanje plaćanja. Svaka konkretna strategija implementira taj interfejs na svoj način.



## **State pattern**

U našem online kafić sistemu potreba za state patternom javlja se kada narudžba može imati različita stanja, a ponašanje sistema zavisi od tog trenutnog stanja. Na primjer, narudžba može biti u stanju "Potvrđena” i “Gotova”.

Korištenjem state patterna, omogućavamo da se ponašanje narudžbe automatski mijenja u skladu sa stanjem u kojem se nalazi, bez potrebe za komplikovanim uslovnim strukturama. Umjesto da imamo veliki broj if ili switch izraza u klasi Narudzba, svako stanje se implementira kao zasebna klasa koja sadrži specifičnu logiku za to stanje.

State pattern omogućava da lakše upravljamo promjenama i dodavanjem novih stanja u budućnosti (npr. "Otkazana" ili "Vraćena"), čineći sistem fleksibilnijim, modularnijim i lakšim za održavanje.

## **Template method pattern**

Potreba za template method patternom javlja se kada imamo algoritme koji dijele zajednički tok izvršavanja, ali se razlikuju u specifičnim koracima. U našem online kafić sistemu, ova potreba se manifestuje u procesu kreiranja i obrade različitih tipova proizvoda.

Imamo hijerarhiju proizvoda: Proizvod kao baznu klasu, sa podklasama Pice i Hrana. Svaki tip proizvoda prolazi kroz sličan proces pripreme, ali sa specifičnim koracima:

**Zajednički algoritam pripreme:**

1. Validacija narudžbe
2. Provjera dostupnosti sastojaka
3. **Specifična priprema** (razlikuje se za piće i hranu)
4. Označavanje kao gotovo
5. Notifikacija

Template method pattern omogućava da definišemo ovaj zajednički algoritam u baznoj klasi Proizvod, ali da omogućimo podklasama da implementiraju specifične korake.

## **Observer pattern**

Potreba za observer patternom javlja se kada više objekata treba da bude obavješteno o promjenama stanja određenog objekta. U našem online kafiću, observer pattern možemo primijeniti kada se stanje narudžbe mijenja — različiti dijelovi sistema moraju biti obaviješteni o ovim promjenama.

Na primjer, kada se narudžba prebaci iz stanja "Potvrđena" u stanje "Gotova", nekoliko komponenti sistema treba da reaguje:

* **Korisnik** treba da primi notifikaciju da je njegova narudžba spremna
* **Sistem za recenzije** treba da omogući korisniku ostavljanje recenzije
* **Sistem za statistike** treba da ažurira podatke o obrađenim narudžbama
* **Skladište** treba da ažurira zalihe proizvoda

Umjesto da klasa Narudzba direktno pozove sve ove komponente, observer pattern omogućava da se različiti "posmatrači" registruju za praćenje promjena stanja narudžbe. Kada se stanje promijeni, svi registrovani posmatrači automatski dobijaju obavještenje i mogu preduzeti odgovarajuće akcije.

## **Iterator pattern**

Potreba za iterator patternom javlja se kada želimo efikasno i jednostavno prolaziti kroz kolekciju objekata, bez da otkrivamo unutarnju strukturu kolekcije.

Zamislimo da imamo poseban dio u našem online kafić sistemu koji prikazuje sve recenzije jednog proizvoda ili sve stavke unutar narudžbe. Također, možemo imati prikaz svih proizvoda dostupnih na meniju, bilo da su to pića ili hrana.

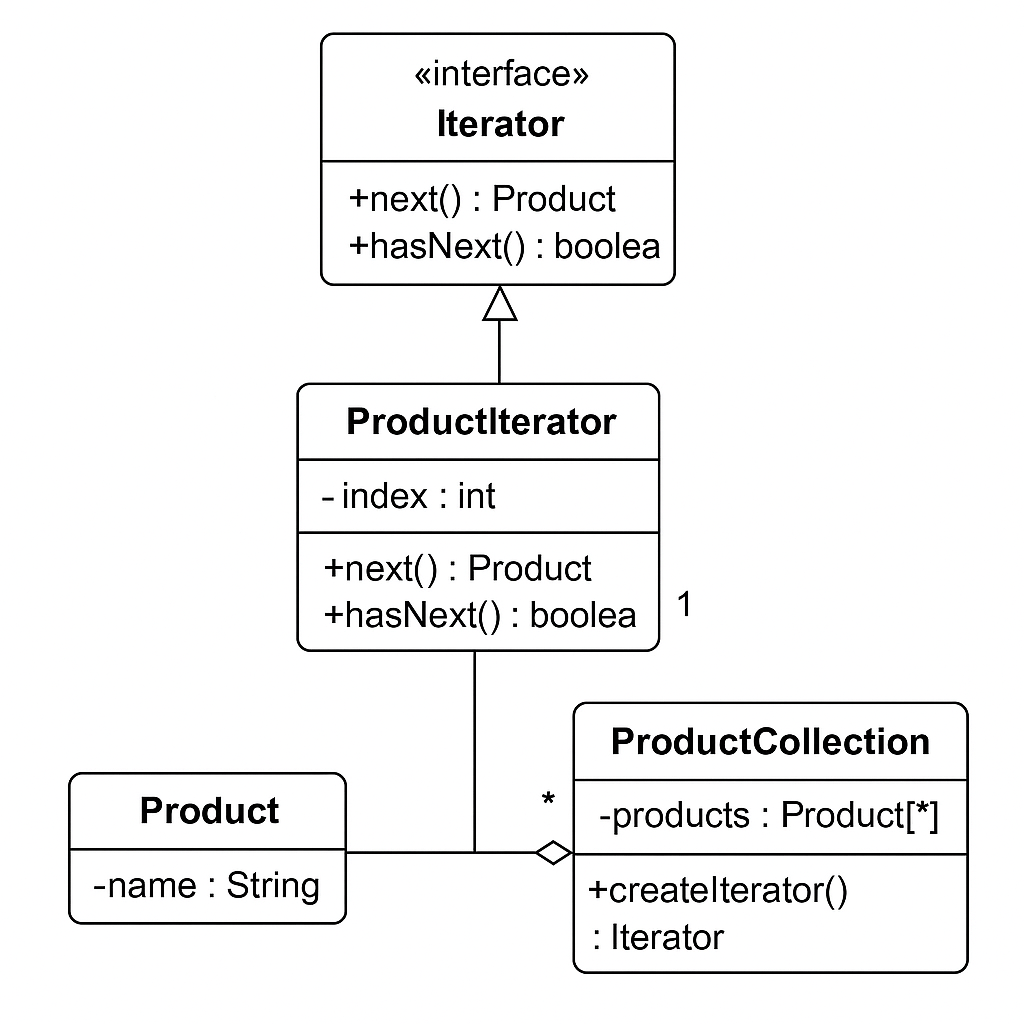
Koristeći iterator pattern, možemo definirati interfejs Iterator koji sadrži metode poput sljedeći (next) i imaSljedeći(hasNext), koje omogućavaju da iteriramo kroz proizvode, stavke narudžbe ili recenzije jedan po jedan.

Implementacija iteratora zavisi o unutarnjoj strukturi kolekcije (npr. niz, lista, set), ali izvana se koristi isto sučelje.

Iterator pattern omogućava promjenu načina pohrane kolekcija bez promjene koda koji ih koristi. Na primjer, možemo iz ArrayList preći na HashSet, a kod koji koristi iterator ostaje isti. Omogućava nam da jednostavno dodajemo nove vrste kolekcija i mijenjamo način iteracije bez uticaja na ostatak koda. Također nam omogućava da kod koji prolazi kroz kolekcije bude neovisan o konkretnoj implementaciji kolekcije, što olakšava održavanje i fleksibilnost sistema.

**Primjeri u sistemu:**

* **Narudzba** sadrži listu **stavkiNarudzbe** → možemo koristiti iterator za prolazak kroz sve stavke i izračun ukupne cijene.
* **Proizvod** ima više **recenzija** → pomoću iteratora možemo redom prikazati sve recenzije uz mogućnost filtriranja (npr. samo ocjene veće od 3).
* **Osoba** može imati više **narudžbi** → možemo koristiti iterator za prikaz historije narudžbi korisnika.



## **Mediator pattern**

Potreba za **mediator patternom** javlja se kada imamo veliki broj objekata koji međusobno komuniciraju, što može dovesti do zapetljanog i teško održivog koda zbog mnogobrojnih veza između objekata. Mediator pattern rješava ovaj problem tako što centralizira komunikaciju – svi objekti komuniciraju preko zajedničkog "posrednika" (mediatora), a ne direktno jedni s drugima.

U kontekstu našeg online kafića, zamislimo da imamo **komunikaciju između različitih komponenti sistema** — korisnika, narudžbi, kartica za plaćanje i recenzija. Bez mediatora, klase bi morale direktno pozivati metode jedne druge, što povećava ovisnost i smanjuje fleksibilnost.

Na primjer:

* Kada korisnik napravi narudžbu, treba se ažurirati status narudžbe, izvršiti plaćanje, poslati obavijest, i eventualno omogućiti recenziju.
* Umjesto da Osoba, Narudzba, KarticnoPlacanje i Obavijest komuniciraju direktno, možemo uvesti klasu NarudzbaMediator koja koordinira ovaj proces.

## **Chain of Responsibility Pattern**

**Chain of Responsibility pattern** se koristi kada želimo obraditi zahtjev kroz lanac različitih objekata, gdje svaki objekt ima priliku da obradi zahtjev ili ga proslijedi dalje. Na taj način, eliminiramo potrebu za čvrsto vezanim if-else blokovima i omogućavamo fleksibilno rukovanje zahtjevima.

U našem sistemu online kafića, ovaj pattern možemo primijeniti kod:

* **Validacije narudžbi** (npr. provjera kartice, dostupnosti proizvoda, validnost korisnika).
* **Obrade obavijesti** — sistem bi mogao imati lanac handlera koji provjeravaju tip obavijesti, važnost, status narudžbe i odlučuju kako je obraditi.

