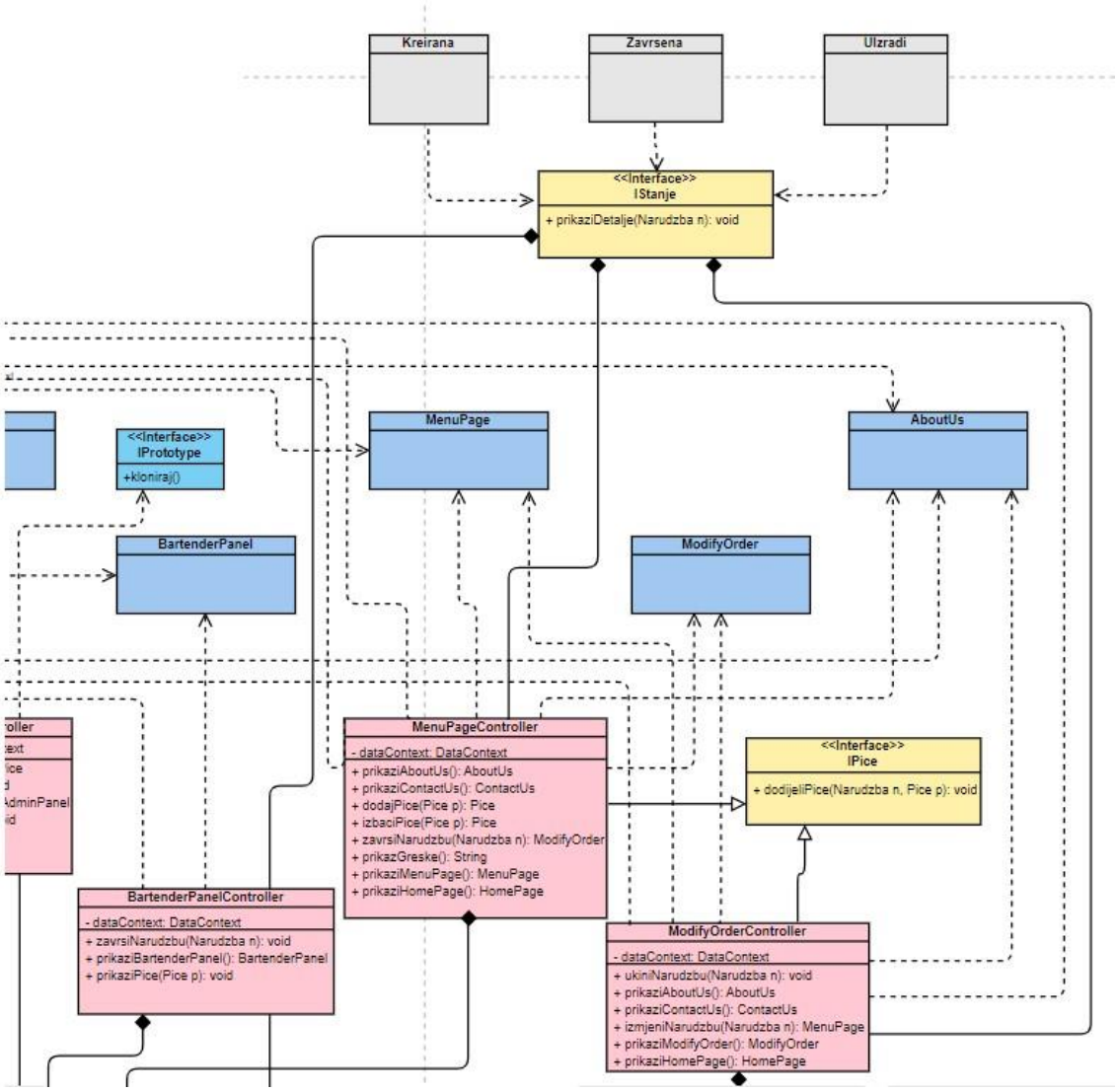
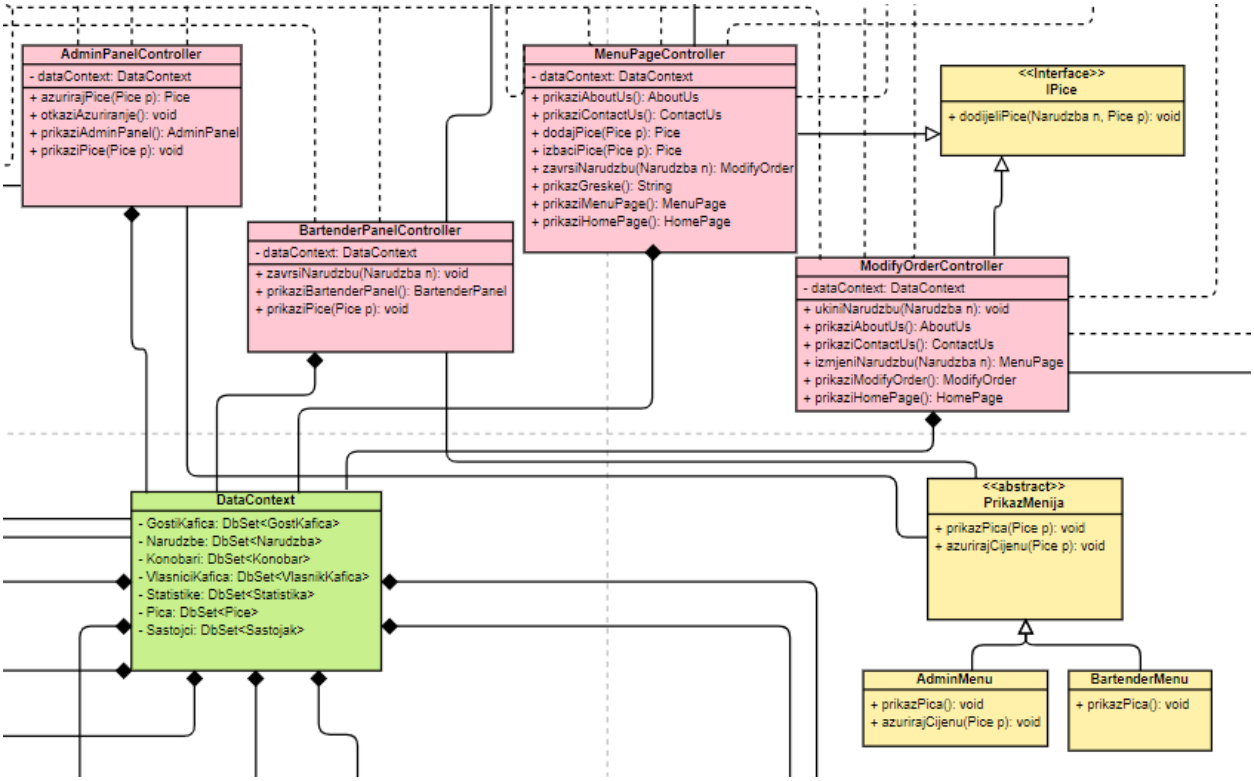
Patterni ponašanja

Patterni ponašanja koje smo odlučile implementirati u našem projektu su: state i template method.

State pattern – ovaj pattern se koristi kada kada imamo različita stanja u koje neka klasa, tj. njen objekat može doći. Stanja se automatki mijenjaju, odnosno mi ih ne možemo svojevoljno promijeniti. Ovaj pattern smo odlučile implementirati u našem programu na sljedeći način. Uvest ćemo različita stanja narudžbe. Prvo stanje u kojem se narudžba može nalaziti jeste da je tek kreirana i to je onaj period kada gost pritisne 'Finish order' dugme. Sljedeće stanje u kojem se može naći jeste u izradi, to je stanje kada prođu 3 minute od pravljenja narudžbe i ona se pošalje u sistem koji će je dalje proslijediti konobaru naobradu. I posljednje stanje u kojem se narudžba može naći jeste završetak, to je dio kada konobar ili primi ili odbije narudžbu, ona će biti završena. Ova različita stanje ćemo staviti u sistem na sljedeći način. Imat ćemo interfejs *IStanje* koji će biti povezan sakontrolerima koji rade sa narudžbom, a to su *BartenderPanelController*, *MenuPageController* i *ModifyOrderController*. Iz ovog interfejsa ćemo naslijediti klase koje će predstavljati različita stanja narudžbe, a to su ona stanja koja smo maloprije spomenuli: *Kreirana*, *Uizradi* i *Zavrsena*.



Template method pattern – je pattern koji ima jednu glavnu klasu koja ima razne metode. Ta klasa se grana na druge klase koje nasljeđuju sve metode iz bazne klase, ali neke metode može promijeniti, te će različite klase na različite načine obavljati neku metodu. Ovaj pattern smo odlučile implementirati na sljedeći način. Dodat ćemo jednu apstraktnu klasu koja će se zvati PrikazMenija. Kontrolerske klase AdminPanelController i BartenderPanelController će biti povezane s ovom klasom. Iz nje su naslijeđene dvije klase, a to su AdminMenu i BartenderMenu. Abstraktna klasa PrikazMenija ima dvije metode: prikazPica() i azurirajCijenu(Pice p). Metoda koja je ista za obje naslijeđene klase je prikazMenija koja i za Admina i za Bartendera prikazuje isto meni. Metodu azurirajCijenu(Pice p) uzima samo AdminMenu, jer on ima pravo na mijenjanje cijene, dok Bartender nema, te tu metodu neće ni implementirati.



Strategy pattern - design pattern koji se koristi kada želimo omogućiti promjenu ponašanja objekta tokom izvođenja programa. Ovaj pattern omogućuje definiranje različitih strategija (algoritama) koje objekt može koristiti, a odabir strategije se može mijenjati dinamički. U našem projektu, Strategy pattern bi mogao biti primijenjen kod plaćanja narudžbe. U različitim dijelovima vašeg koda gdje se vrši kreiranje i manipulacija narudžbama, možete koristiti Strategy pattern tako da omogućite odabir različitih strategija plaćanja. Na primjer, prilikom kreiranja narudžbe, klijent može odabrati plaćanje putem PayPal-a, Credit Card-a ili gotovinom. Kreiranjem interfacea PaymentStrategy definiramo metode poput makePayment i refund za različite načine plaćanja. Na primjer, možete imati implementacije PayPalStrategy, CreditCardStrategy, CashStrategy itd. Order klasa bi sadržavala metode poput setPaymentStrategy koja postavlja željenu strategiju plaćanja i makePayment koja će pozvati metodu makePayment na odabranoj strategiji. Implementacija Strategy patterna omogućava fleksibilnost u odabiru strategije plaćanja, omogućujući jednostavno dodavanje novih načina plaćanja u budućnosti. Također, odvaja logiku plaćanja od same klase Order, što olakšava održavanje i poboljšava čitljivost koda.

Observer pattern - često se koristi za implementaciju komunikacije između objekata u kojem jedan objekt obavještava druge objekte o promjeni svog stanja, što bi nama moglo biti korisno u sistemu, ali smo se ipak odlučile da ga ne implementiramo. U kontekstu našeg sistema, observer pattern bi se mogao primjeniti na više načina, te ćemo sada opisati neke od potencijalnih slučajeva kada bi nam koristio. Kada konobar obradi narudžbu, možemo implementirati observer pattern da obavijesti druge relevantne objekte o tome. Na primjer, ako bi imali objekt "Šank" koji će se registrovati kao observer i biti obaviješten svaki put kada konobar obradi narudžbu. Šank može zatim pokrenuti pripremu narudžbe. Ako vlasnik želi pratiti stanje kafića, možemo koristiti observer pattern kako bi obavijestili vlasnika o važnim događajima. Na primjer, ako imamo objekt "Vlasnik" koji će biti observer on će biti obaviješten kada se zalihe nekog važnog sastojka približe minimumu ili kada broj gostiju pređe određeni prag.

Iterator pattern - ovaj pattern se koristi za pristupanje elemenata kolekcije bez otkrivanja unutrašnje strukture te kolekcije. Primjer primjene patterna Iterator u našem projektu može biti prolaženje kroz narudžbe i prikazivanje njihovih detalja na korisničkom interfejsu. Umjesto direktnog pristupa elementima kolekcije naloga, koristili bismo iterator da bismo dobili svaku narudžbu jednu po jednu i prikazali njene detalje na interfejsu. Implementacija Iterator patterna pruža jednostavnost, fleksibilnost i intuitivan pristup elementima kolekcije. Također olakšava dodavanje novih metoda i operacija koje se mogu izvoditi na narudžbama u budućnosti, bez promjene interne strukture kolekcije. U projektu mogao bi biti primijenjen na sljedeći način: Ako bismo klasu "OrderCollection" implementirali kao kolekciju naloga, unutar te klase bismo definirali i implementirali "Iterator" interfejs, koji bi sadržavao metode poput "getNext", "hasMore" i "reset" za pristup elementima kolekcije. Takođe, unutar klase "OrderCollection" implementirali bismo "OrderIterator" koji bi bio odgovoran za iteraciju kroz elemente kolekcije naloga. Klijenti mogu dobiti instancu objekta "OrderIterator" pozivanjem metode "getIterator" unutar klase "OrderCollection" i tako mogu koristiti iterator za petlju kroz narudžbe.

Chain of responsibility pattern – koristi se kada želimo da se obavljaju neke radnje u lancu, gdje ukoliko imamo neke tri radnje ne možemo odmah preći na drugu ili sa prve na treću, već se sve odvija u lancu, tj. dok ne završimo prvu radnju ne možemo preći na drugu itd. Ovaj pattern nismo odlučile implementirati, ali sada ćemo navesti jedan primjer gdje bi se mogao upotrijebiti. Ukoliko bi u našoj aplikaciji, osim prodaje pića, postojala još prodaja slanih i slatkih jela, tada bismo osim klase *Pice* imali i klase *SlanoJelo* i *SlatkoJelo*. Kada gost dođe i želi da naruči hranu mi možemo primijeniti ovaj pattern gdje bi se hrana naručivala u lancu. Na primjer, želimo da gost prvo odabere neku slanu hranu, nakon što odabere onda prelazi na slatku hranu i nakon što i to odabere dolazi do pića koje isto tako treba odabrati. Ovdje bi se naručivanje odvijalo u lancu, tj. ne bi mogli prvo odabrati piće ili sa slanog jela da pređemo na odabir pića.

Mediator pattern – možemo koristiti za olakšavanje komunikacije između različitih objekata putem posrednika, umjesto da direktno komuniciraju između sebe. Primjer gdje bi mogli iskoristiti pattern je naveden u nastavku. Kada bi imali više konobara u kafiću, možemo koristiti mediator pattern kako bi omogućili komunikaciju između njih. Mediator objekt može poslužiti kao centralna tačka za razmjenu informacija između konobara. Na primjer, kada jedan konobar završi s obradom narudžbe, može poslati obavijest mediatoru, a zatim mediator može proslijediti tu informaciju drugim konobarima koji možda imaju veze s tom narudžbom (na primjer, ako je potrebna pomoć pri dostavi).