STRUKTURALNI PATERNI

Adapter pattern:

Da bi vlasnik mogao pratiti recepte na jednostavan način, trebao bi moći dobiti sortirane ili odabrane po ljubimcu koji ga zanima. Tu bi mogli koristiti adapter pattern da obavlja tu funkciju da ne bi mijenjali glavnu klasu Recept

Bridge pattern:

S obzirom da u našem slučaju kod prijava imamo više načina da se prijavimo u sistem, od skeniranja QR koda, ručne prijave, a i budućih mogućih načina na koje bi se prijavljivali, mogli bismo imati apstraktnu klasu PrijavaKorisnika koja će imati dalje mogućnosti kao što su QRLogin, RucniLogin, itd.

Facade pattern:

Korisnik koji je prijavljen kao veterinar/ka treba imati pristup i ljubimcima, pregledima, receptima. Svaku od tih klasa možemo posmatrati kao poseban podsistem onoga što veterinar/ka može da radi. Tako bismo u klasi Facade mogli imati svaki od tih 3 podsistema, a svaki od tih podsistema ima svoje vlastite operacije. Sve one operacije koje veterinar/ka može da izvodi možemo da stavimo u klasu Facade jer u njoj možemo držati operacije sastavljene od različitih dijelova podsistema. Na taj način sakrivamo kompleksnost sistema i pružamo korisniku (veterinaru/ki) interfejs kojim on može da pristupa svakom od tih podsistema.

Proxy patern:

Proxy patern bi mogao biti realizovan na klasi Administrator, zato što toj klasi ne bi trebalo moći direktno pristupiti, a ona sama ima poprilično velike efekte na čitav sistem.

Composite patern:

Decorator patern:

Pošto kod klase ljubimac dodajemo sliku, mogli bismo dodati interfejs ISlikaLjubimca koja bi imala metode kao što su postaviSliku(), izrežiSliku() i sl. Uglavnom da se slika može malo editovati prije nego što se postavi u klasi ljubimac.

Flyweight patern: