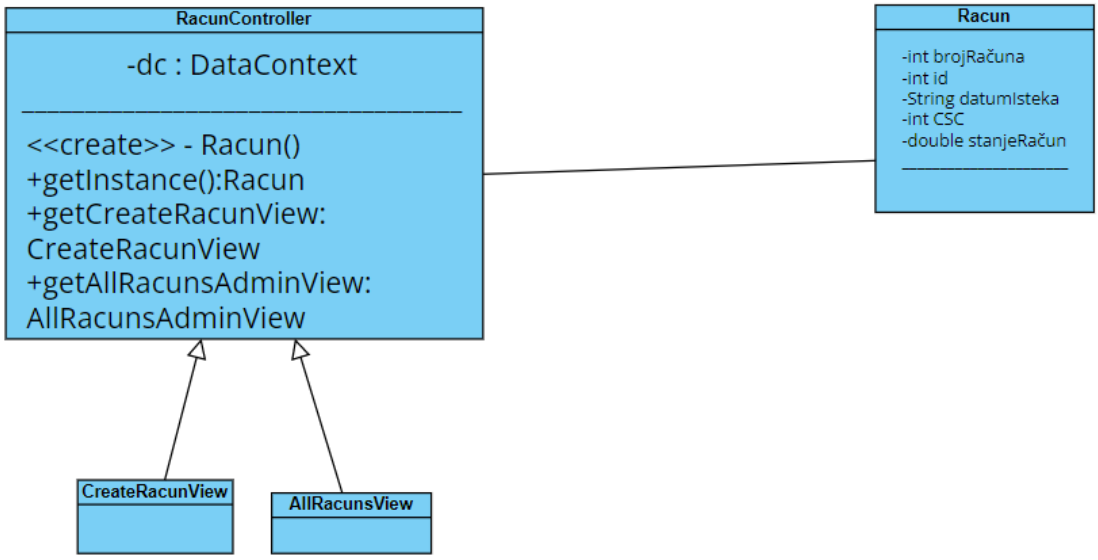
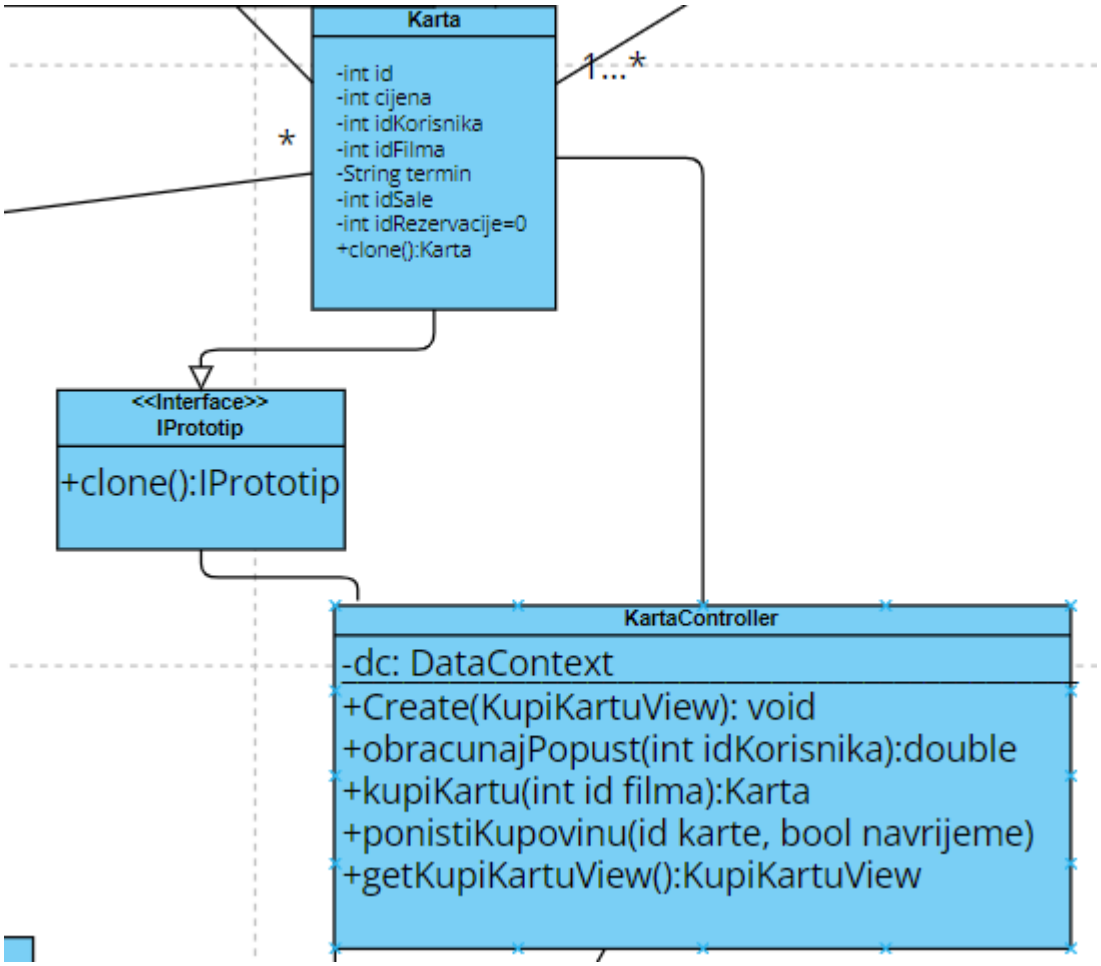
Singleton dessign pattern se koristi kada želimo da budemo sigurni da koristimo samo jednu instancu klase, to se postiže tako što konstruktor proglasimo za privatni, te implementiramo metodu getInstance(), koja proizvodi novu instancu klase samo ukoliko je trenutna instanca jednaka null, u suprotnom samo vraća već postojeću instancu. Singleton omogućava globalni pristup instanci klase, što znači da se može pristupiti iz bilo kojeg dijela programa. To je korisno kada želimo dijeliti jedan objekat ili resurs među različitim dijelovima aplikacije. Singleton pattern može biti implementiran na način da bude thread-safe, tj. bezbjedan za korištenje u višenitnim okruženjima. To se može postići upotrebom mehanizama sinhronizacije poput korištenja ključeva zaključavanja (locks) ili upotrebom dvostepene provjere (double-checked locking) prilikom kreiranja instance. Iako Singleton može biti koristan u određenim situacijama, treba biti oprezan prilikom njegove upotrebe. U nekim slučajevima može dovesti do povećane složenosti koda i težeg testiranja. Također, Singleton može biti problematičan ako se koristi za dijeljenje mutable objekata među više dijelova aplikacije, jer to može dovesti do nesigurnosti i nepredvidivog ponašanja. Singleton je koristan alat koji može biti primijenjen u određenim situacijama kada je potrebno osigurati jedinstvenu instancu klase i globalni pristup toj instanci. Važno je pažljivo razmotriti prednosti i nedostatke prije nego što se odlučite koristiti singleton pattern u vašem projektu. U našem projektu implementirali smo singleton tako što smo proglasili za privatni, da se ne bi moglo praviti više instanci pomoću konstruktora, te smo dodali metodu +getInstance(). Naravno to treba uraditi za sve klase. Imalo bi smisla u našem projektu da jedan korisnik u jednom trenutku može imati samo jedan aktivan račun.



Prototype dessign pattern je koristan kada želimo kreirati nove objekte na osnovu postojećeg objekta, umjesto da ih instancirate iznova. Ovo može biti korisno u situacijama kada je kreiranje objekta skupo ili složeno. Važno je napomenuti da se prototype pattern može koristiti zajedno sa drugim patternima, kao što su factory pattern ili builder pattern, kako bi se postigla dodatna fleksibilnost i prilagodljivost u kreiranju objekata. Ključna funkcionalnost prototype patterna je kloniranje objekta. Umjesto da se objekti instanciraju putem konstruktora, postojeći objekat se

klonira kako bi se kreirala nova kopija. Kloniranje može biti plitko ili duboko, zavisno od složenosti objekta i potreba aplikacije. Prototype pattern omogućava dinamičko kloniranje objekata tokom izvršavanja. To znači da se nove instance mogu kreirati i mijenjati u zavisnosti od zahtjeva aplikacije, bez potrebe za statičkim kodom. Upotreba prototype patterna može biti efikasna jer se izbjegava skup proces kreiranja objekata koji su složeni za inicijalizaciju. Umjesto toga, koristi se kloniranje objekta koji već ima inicijalizirane vrijednosti. U našem projektu imamo IPrototip koji je interfejs kojeg implementira klasa Karta, odnosno ona ima metodu clone(). Taj interfejs sada koristi klasa KartaController u kojoj možemo pozvati metodu clone(), koja će nam klonirati objekat tipa Karta. Naprimjer možemo reći IPrototip x=(IPrototip) karta.clone(), s obzirom da je Karta izvedena iz interfejsa objekat će moći biti tipa Karta. Ukratko, prototype pattern omogućava kreiranje novih objekata na osnovu postojećeg objekta, čime se izbjegava skup proces instanciranja. Kloniranje objekata pruža efikasno rješenje za kreiranje više sličnih objekata i omogućava dinamičko prilagođavanje novim zahtjevima aplikacije, a mi ćemo u klasi KartaController po potrebi kreirati odnosno klonirati objekte tipa Karta koristeći ovaj metod, odnosno nećemo to raditi direktno, nego preko interfejsa namjenjenog samo za to.



Factory method pattern je koristan za instanciranje raličitih vrsta podklasa uz pomoć factory metode. Instanciranje se izvršava kroz definisanje factory metode umjesto direktnog pozivanja konstruktora, pri čemu sama metoda odlučuje koja će se podklasa instancirati i na koji način će biti izvršena programska logika. Factory method pattern ispunjava Open-Closed Prinicple SOLID principa. Dakle on podržava otvorenost za proširivanje, a istovremeno je zatvoren za promjene. Npr. ukoliko želimo dodati neku novu vrstu projekcije, tada moramo dodati i novu podklasu koja će u sebi implementirati factory metodu samo za tu vrstu projekcije. Ovaj pattern nećemo implementirati, ali ukoliko bismo ga implementirali, mogao bi se iskoristiti za razlike u projekcijama različitih žanrova. Definisali bi interfejs ProjekcijaFactory u kojoj bi bila definisana apstraktna metoda krerirajProjekciju(). Dalje bismo imali konkretne klase koje bi bile za različite žanrove, npr. akcioniFIlmFactory(), hororFilmFactory()... Sve ove konkretne klase bi implementirale metodu kreirajProjekciju(). koja bi vraćala odgovarajuće objekte za odgovarajući žanr. Pored toga ove klase bi mogle imati dodatne metode ili atribute specifične za određeni žanr.

Abstract factory pattern omogućava kreiranje srodnih povezanih objekata bez da se njihva konkretna pripadnost treba specifirati. Abstract factory pattern ustvari definiše apstraktnu fabriku u kojoj su definisane metode za kreiranje objekata. Dalje svaka konkretna fabrika će implementirati apstraktnu fabriku te implementirati metode za kreiranje datih objekata. Korištenjem Abstract factory patterna možemo izbjeći upotrebu velikog broja if-else uslova ukoliko želimo kreirati različite hijerahije objekata, jer samom upotrebom abstract factory patterna gdje određeni tip fabrike sadrži određene tipove objekata i tačno zna koju će podklasu instancirati. Pored navedenog, prednost Abstract factory patterna je i to što nam on omogućava laku zamjenu cijelih porodica objekata u slučaju da cijeli sistem trebamo prilagoditi nekim drugim okruženjima ili zahtjevima. Abstract factory pattern nećemo implementirati, ali on bi se za naš projekat mogao iskoristiti npr. za upravljanje tipovima podataka kao što su projektori. Ukoliko bismo imali različite projekcije filmova, 2D, 3D i 7D projekcije. Dakle projekcija bi bila jedna porodica i svi objekti unutra bi imali neke određene zajedničke metode, jedna od takvih metoda bila bi metoda za pokretanje projekcije. Apstraktna fabrika bi imala metodu koja bi kreirala projektor. Zatim bismo imali konkretnu fabriku koja bi imlementirala datu metodu. Tada bismo imali 2Dfactory, 3Dfactory gdje bismo implementirali metodu iz apstraktne klase na načine svojstvene konkretnim klasama u ovom slučaju projektorima.

Builder pattern se koristi za konstrukciju kompleksnih objetaka korak po korak, odnosno korištenjem builder patterna možemo izbjeći kompleksnost korištenja konstruktora određene klase koji može zahtijevati navođenje mnogo atributa. Kao što smo rekli prilikom

korištenja builder patterna objekat se konstruiše korak po korak, tako što se svaki korak izvršava korištenjem posebnog buildera (graditelja). Tako svaki korisnik može konfigurisati objekat na način da će ga prilagoditi svojim potrebama i na taj način se dovoljno olakšava konfiguracija objekta sa različitim parametrima ili opcijama. Ukoliko bismo implementirali Builder pattern za naš kino projekat, mogli bismo to uraditi za konstrukciju objekta tipa rezervacija. Imali bismo klasu Rezervacija koja bi služila za rezervaciju sjedišta i sadržavala neke osnovne podatke kao što su broj sjedišta, ime gledatelja, datum rezervacije. Zatim bismo kreirali RezervacijaBuilder klasu koja bi nam sve ove date podatke postavila i kreirala atribute rezervacije sjedišta. Onda bismo kreirali neke interfejse za različite rezervacije, npr. KlasicnaRezervacijaBuilder ili VIPRezevacijaBuilder, te bi nam ti interfejsi omogućili dodatne konfiguracije ili opcije po čemu bi se ova dva tipa rezervacija razlikovala.