# Kreacijski paterni

## 1. Singleton pattern

Singleton pattern se koristi za ograničavanje instanciranja klase na samo jedan objekt. To znači da klasa ima samo jednu instancu kojoj se omogućuje globalni pristup.

Singleton design pattern bi se npr. mogao koristiti prilikom konekcije na bazu. Umjesto da pravimo novu instancu za svaki poziv, napravimo samo jednu instancu koju pozivamo tokom života aplikacije. U C# zajedno sa EF Core je to automatski riješeno tako da nemamo problema s tim.

## 2. Prototype pattern

Prototype pattern se koristi za kreiranje novih objekata na temelju već postojećih objekata. Umjesto da se koristi klasično instanciranje objekata (new operator), Prototype pattern koristi kopiranje (cloning) postojećeg objekta kako bi se stvorila nova instanca.Glavna ideja iza Prototype patterna je da se stvori objekt koji će služiti kao prototip (uzorak) i zatim se koristi taj prototip za kreiranje novih objekata kroz kopiranje. Na taj način, izbjegava se direktno instanciranje objekata.

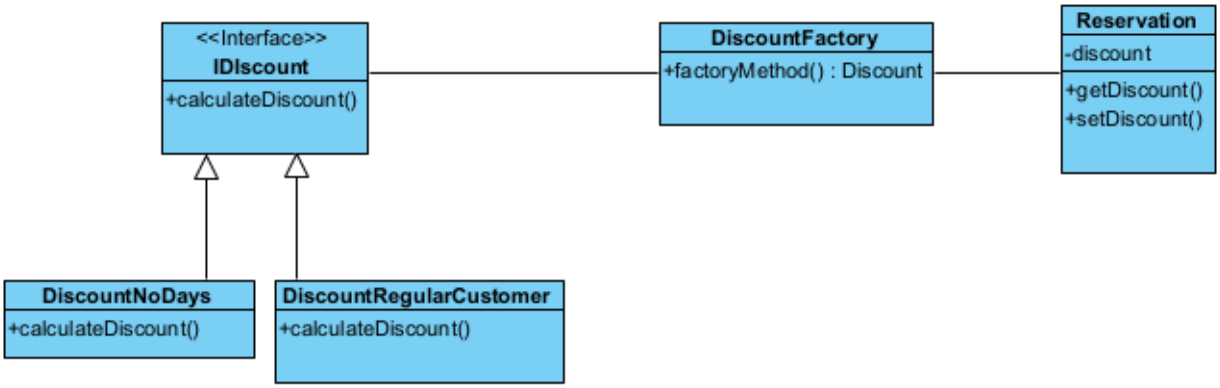
Ovaj pattern možemo koristiti kada imamo neko vozilo i želimo napraviti vozilo slično njemu možda sa drugom bojom. Da ne bi sva polja kopirali redom napravit ćemo prototip i klonirati to vozilo i samo izmjeniti boju.

## 3. Factory Method pattern

Factory Method pattern (Patern fabričke metode) je dizajn obrazac (design pattern) koji se koristi za kreiranje objekata, ali prepušta konkretne detalje kreiranja podklasama. Factory Method pattern definiše apstraktnu klasu ili interfejs koji sadrži apstraktnu metodu koju konkretne podklase implementiraju kako bi kreirale objekte.Glavna ideja iza Factory Method patterna je da se odvoji proces kreiranja objekata od koda koji koristi te objekte. Umjesto da direktno instancirate objekte, koristite Factory Method koji abstrahuje detalje kreiranja i pruža fleksibilnost da se odaberu različite implementacije objekata.

Ovaj pattern mozemo koristiti za računanje popusta za određenu rezervaciju. Napravit ćemo postojale različite Discount klase za računanje popusta na različit način u odnosu na neki kriterij (popust u odnosu na broj dana rezervacije ili popust ukoliko je stalni klijent), tada bi pozivali Factory Method kako bi dohvatili instancu discounta. Tako možemo koristiti tu instancu za izračunavanje konačne cijene rezervacijenekog vozila.
**Ovaj design pattern ćemo implementirati u našu aplikaciju.**

Implementiramo ga tako što napravimo  interfejs klasu Idiscount, zatim imamo različite discount klase koje impementiraju taj interfejs i imat ćemo Creator klasu koja će imati Factory metodu koja će biti apstraktna za dvije klase koje će vršiti stvarni proračun popusta.


<<Interface>>
IDIscount
+calculateDiscount()
DiscountFactory
+factoryMethod() : Discount
Reservation
-discount
+getDiscount()
+setDiscount()
DiscountNoDays
+calculateDiscount()
DiscountRegularCustomer
+calculateDiscount()

## 4. Abstract factory pattern

Abstract Factory pattern pruža interface za stvaranje porodice srodno povezanih objekata bez eksplicitnog specificiranja njihovih konkretnih klasa. Ovaj obrazac omogućava kreiranje objekata koji su međusobno kompatibilni i rade zajedno, bez da se detaljno zna o njihovim konkretnim implementacijama.Glavna ideja iza Abstract Factory patterna je da se korisniku pruži intefrace koji definiše niz metoda za kreiranje objekata. Svaki konkretsn factory implementira taj interface i pruža implementacije za kreiranje objekata koji su međusobno kompatibilni. Na taj način, korisnik može koristiti apstraktni factory da kreira objekte bez da zna o konkretnu implementaciju.

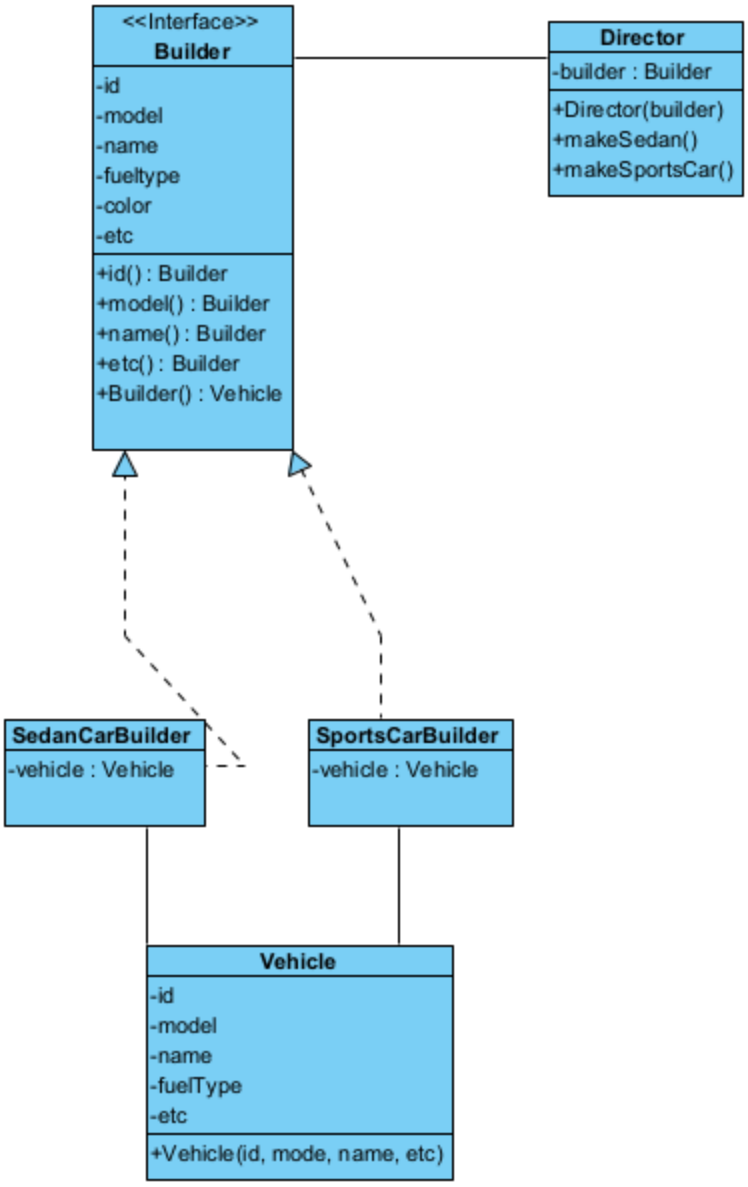
I ovaj pattern mozemo iskoristiti za implementaciju discounta za određenu rezerevaciju. U ovom slučaju kada koristimo instancu apstraktnog factory-a da bi dobili objekat discounta. Zatim tu instancu koristimo za izračunavanje konačne cijene rezervacije. Za našu aplikaciju ovaj pattern je bespotrebno komplikovanje tako da ga nećemo koristiti.

## 5. Builder pattern

Builder pattern se koristi za konstrukciju složenih objekata korak po korak. Omogućava kreiranje različitih reprezentacija istog objekta koristeći isti konstrukcijski kod.Glavna ideja iza Builder patterna je da se odvoji proces konstrukcije objekta od njegove reprezentacije. Koristi se kada želimo konstruirati objekte koji su složeni i imaju različite varijacije, ali želimo da se konstrukcija tih objekata izvršava na sistematičan način.

Builder pattern se može koristiti za kreiranje i konfiguriranje rezervacije vozila. Glavni dio Builder patterna je Builder klasa koja sadrži metode za postavljanje različitih atributa rezervacije, kao što su model vozila, godište, mjenjač, period rezervacije, cijena itd. Builder klasa omogućuje korisniku da postavlja samo one atribute koji su potrebni za rezervaciju, a ostatak se može postaviti na neku defaultnu vrijednost. Nakon što svi atributi budu proslijeđeni poziva se metoda build() na Builder objektu, koja lreira konačan objekt rezervacije vozila.
**Ovaj design pattern ćemo koristiti u našoj aplikaciji.**

Implementirat ćemo ga tako što ćemo imati interfejs Builder i CarBuilder koji implementira taj interfejs. Zatim imat ćemo Director klasu koja upravlja koracima izgradnje objekta. I na kraju imamo konkretan produkt odnosno Vehicle koji je vezan za CarBuilde


<<Interface>>
Builder
-id
-model
-name
-fueltype
-color
-etc
+id() : Builder
+model() : Builder
+name() : Builder
+etc() : Builder
+Builder() : Vehicle
Director
-builder : Builder
+Director(builder)
+makeSedan()
+makeSportsCar()
SedanCarBuilder
-vehicle : Vehicle
SportsCarBuilder
-vehicle : Vehicle
Vehicle
-id
-model
-name
-fuelType
-etc
+Vehicle(id, mode, name, etc)


Elektrotehnički fakultet
Univerziteta u Sarajevu