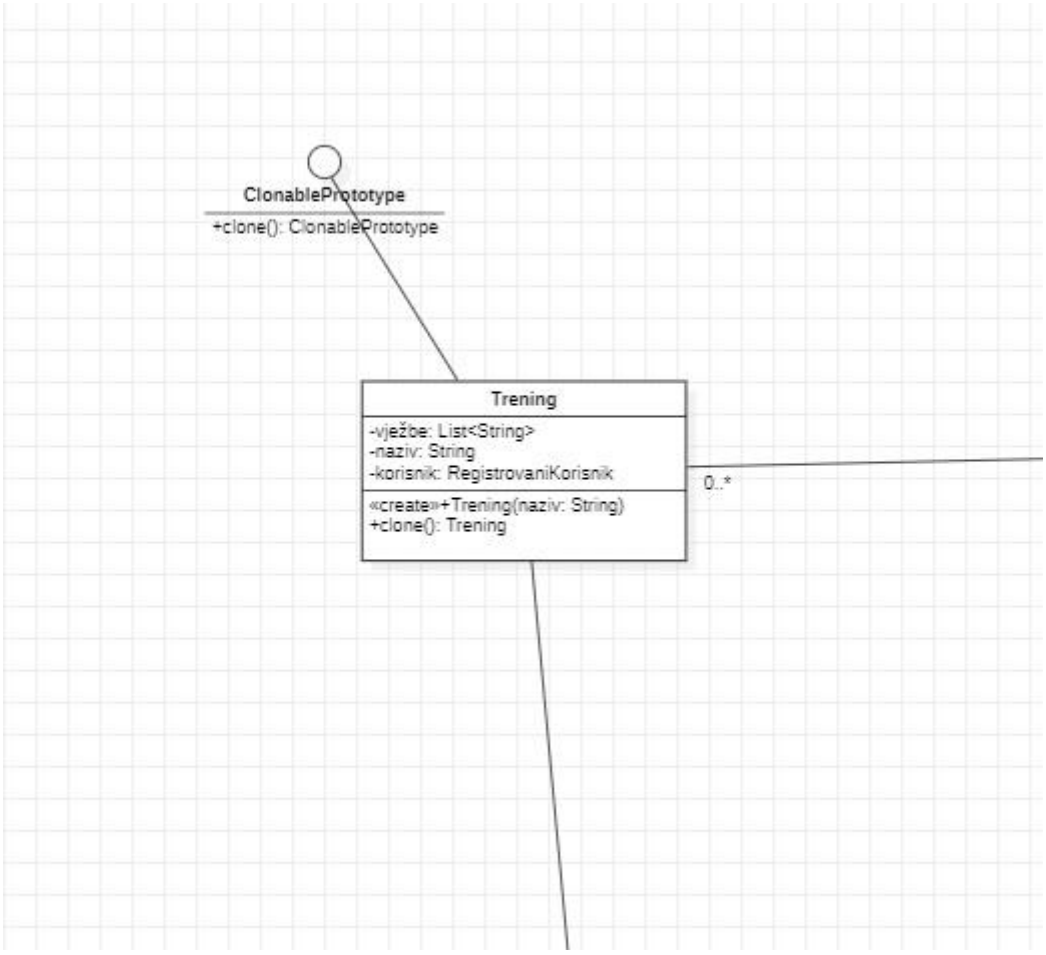
Singleton
Singleton kreacijski pattern nam sluzi da se, po potrebi, osiguramo da ce za odredjenu klasu postojati samo jedna instanca. To se postize tako sto se konstruktor proglasi privatnim (jer se samim cinom pozivanja konstruktora stvara novi objekat i mijenjanje toga je van nase kontrole), napravi se privatni staticki atribut tipa same te klase (u kontekstu C#-a to je referenca) i napravi se javna staticka *kreaciona* metoda. Ta metoda radi 2 stvari: ako je gore spomenuti atribut/referenca *null* kreira se novi objekat tipa same te klase i referenca se postavi da "pokazuje" na njega; u zadnjem koraku vraca objekat na koji atribut "pokazuje".

Singleton pattern bi se u nasem projektu mogao iskoristiti ako bi se odlucili implementovati neku vrstu globalnog tajmera koji bi se ponasao kao *logger*. Biljezio bi kada se svaki Task, Vjezba... zapocnu, kada se zavrse, koliko je dugo trebalo da se izvrse, koliko je korisnik bio aktivan na aplikaciji u toku dana, koliko je taskova/vjezbi... odradjeno od registracije, koliko je preskoceno... Posto se vrijeme racuna od registracije korisnika, nema smisla da ovih tajmera bude vise, vec samo jedan.

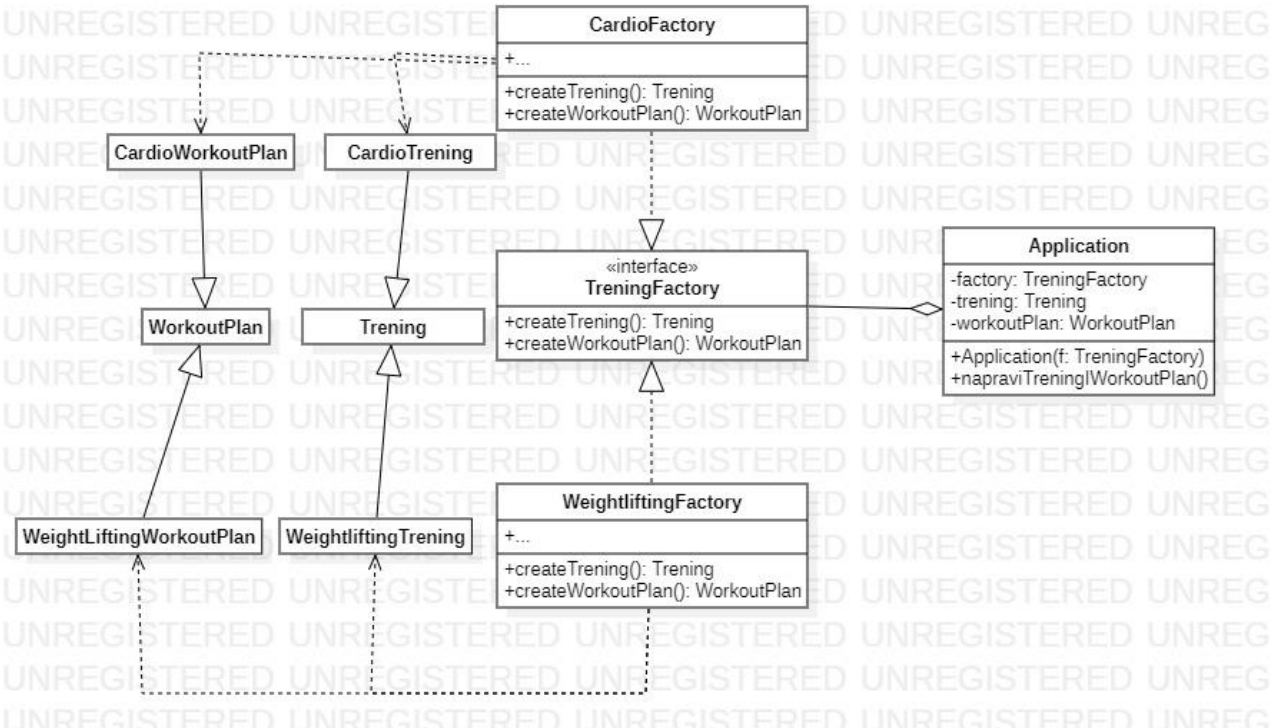
Prototype
Prototype kreacijski pattern nam sluzi kada zelimo da kopiramo objekat bez da nam kod ovisi od njegove klase. Ovo kopiranje bi se inace radilo na sljedeci nacin: instanciramo objekat iste klase i sve atribute prvog rucno "prekopiramo" u drugi pazeci na plitkocu kopija (govorimo o dubokoj kopiji). Zavisnosti od vidljivosti i ugnjezdenosti pojedinih atributa ovaj proces moze biti jako komplikovan (lista objekata, svaki objekat ima 3 dodatne liste drugih objekata, pa svaki takav objekat opet ima neki stack, pa elementi stacka su liste, pa te liste su privatne bez gettera...). Svrha ovog patterna jeste da se kreiranje instance klase delegira samom objektu te klase.
Prototype pattern bi se jako jednostavno mogao iskoristiti u nasoj aplikaciji. Primjer navodimo samo za treninge, a slican je za jela i taskove. Recimo da, nakon sto udjemo u odredjeni trening,



postoji dugme *Dupliciraj trening.* To dugme bi kreiralo identican trening kao trenutni i spasio ga u bazu. Naknadno korisnik moze da mijenja taj trening.

Abstract Factory

Da bi na primjeru pokazali kako implementirati ovaj kreacijski pattern u našu aplikaciju, dodati ćemo određene klase i funkcionalnosti. Prvo ćemo zamisliti da je klasa Trening apstraktna klasa koja se dijeli na sljedeće vrste (familije): CardioTrening, WeightliftingTrening. Osim klase Trening, dodati ćemo klasu WorkoutPlan koja će opisivati strukturu treninga u zavisnosti od vrste kojoj pripada tj. imati ćemo moguće klase: CardioWorkoutPlan, WeightliftingWorkoutPlan. Trening i WorkoutPlan će biti naše Product klase, te je sada potrebno dodati Factory klase. Pošto za svaku familiju moramo imati konkretnu Factory klasu, dodati ćemo dvije nove klase koje implementiraju TreningFactory interface: CardioFactory, WeightLiftingFactory. Svaka od navedenih factory klasa će na svoj način implementirati factory metodu (u ovom slučaju dvije factory metode) uz pomoć koje će kreirati trening I workout plan na osnovu izabrane vrste treninga od strane korisnika. Na sljedećoj slici je prikazan opisani primjer, gdje su u klasama WorkoutPlan I Trening te izvedenim klasama sakriveni atributi i operacije.



Factory method

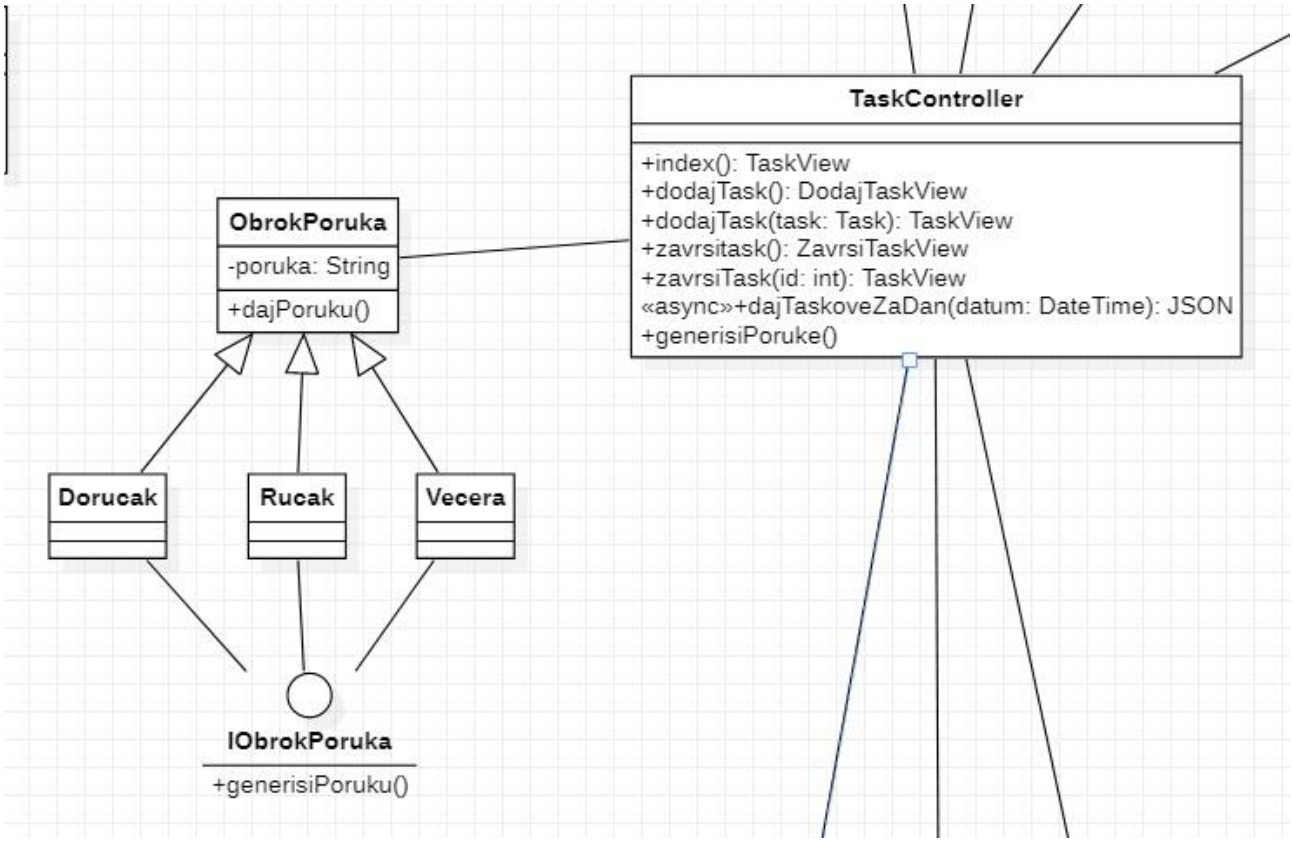
Factory Method Pattern je pattern koji pripada grupi kreacijskih patterna i koji nam onogućava da kreiramo objekte na način da podklase same imaju mogućnost da odluče koja će klasa biti instancirana. Bitno je napomenuti da sve subklase koriste isti interfejs, i da korisnik nema potrebe da zna za razliku između rezultata koje vraćaju različite subklase. Za implementaciju ovog patterna potrebna nam je klasa koje će kreirati interfejs, subklase koje će vraćati različite implementacije tog interfejsa, kreatorska klasa za koju je važno napomenuti da njen povratni tip mora odgovarati interfejsu i konkretne kreatorske klase koje vraćaju različite tipove objekta.

Na koji način bi se ovaj pattern mogao iskoristiti u našoj *LifePlanner* aplikaciji?

Zamislimo da postoji funkcionalnost koja omogućava korisniku da, ako želi izabere opciju koja će mu prikazati koji je obrok prigodan za trenutno doba dana. Ovo bi moglo biti prikazano u vidu neke poruke, koja je u suštini objekat tipa string. Na primjer ukoliko bi korisnik odabrao tu opciju u 9h u jutro, poruka bi mogla glasiti *„Sada je vrijeme za doručak.“*. U slučaju da se opcija odabere u 15h, to jeste u 21h, poruke bi mogle glasiti *„Sada je vrijeme za ručak“,* i *„Sada je vrijeme za večeru.“* respektivno. Sama implementacija ovog patterna bi bila izvedena otprilike na sljedeći način.

U interfejsu bi se mogla naći metoda koja vraća string, recimo *string poruka().* Nakon toga potrebno bi bilo napraviti tri konkretne implementacije ovog interfejsa, to jeste napraviti tri podklase. Zatim slijedi kreatorska klasa koja implementira Factory metodu. Ta metoda u zavisnosti od trenutnog vremena (posmatra koliko je trenutno sati) instancira odgovarajuću podklasu.

Kao što vidimo u ovom primjeru korisnik zna da sve podklase imaju *poruka()* metodu, ali nema potrebe da zna na koji način je ona implementirana u svakoj od njih. Ovo je također dobro jer možemo uvesti na primjer i „dodatne obroke“ bez da se mijenja korisnikov kod, čime je očuvan Open/Closed princip.



Builder

Uloga ovog patterna je odvajanje specifikacije kompleksnih objekata od njihove stvarne konstrukcije. Koristi se isti set koraka za kreiranje razlicitih krajnjih produkata.

U nasem primjeru mozemo posmatrati kalendar taskova koji korisnik kreira. (naravno pored taskova za svaki dan, mogli bi da dodamo jos neke mogucnosti na kalendar kao sto su - rodjendani, napomene, sedmicni ciljevi etc) - ove opcije se posmatraju kao gradivni elementi za kalendar.

IGraditeljKalendara-interfejs koji definira pojedinačne dijelove koji se koriste za izgradnju produkta - u nasem slucaju bi to bili gradivni elementi za kalendar

Director klasa - sadrži logiku i neophodnu sekvencu operacija za izgradnju produkta

GraditeljKalendara klasa koja se poziva od strane Direktora da se izgradi produkt

Product klasa na osnovu koje se kreira objekat koji se gradi preko dijelova - u nasem slucaju je to kalendar.