Univerzitet u Sarajevu

Elektrotehnički fakultet u Sarajevu

Kreacijski dizajn paterni

FitnessFusion

Predmet: Objektno orijentisana analiza i dizajn

Naziv tima: Time – out

Članovi tima: Ahmetović Zerina

Bajrović Taner

Bosno Hamza

Brčaninović Hasan

Peljto Emina

Sarajevo, maj 2023.

Kreacijski dizajn paterni:

Kreacijski dizajn paterni nam pomažu da dizajniramo sistem u kome su odvojeni procesi kreiranja objekta i njihove upotrebe u sistemu. Upotrebom ovih paterna se prekida uska povezanost između klasa, što je popraćeno enkapsuliranim pristupom prilikom instanciranja objekata. Za razliku od strukturalnih paterna, kreacijski dizajn paterni u prvi plan stavljaju objekte i daju odgovor na pitanja najboljeg načina na koji se objekat može instacirati, pri čemu će se ostvariti puni potencijal kreiranog objekta. Osim toga, oni nam na jednostavan način to omogućavaju, što je dodatni motiv za upotrebu ovih paterna.

U ovom dokumentu su razmotreni *Singleton, Prototype, Builder, Factory Method, Abstract Factory* paterni. U našem sistemu smo pronašli primjenu za *Prototype, Builder i Singleton* dizajn paterne, čija će se upotreba u našem sistemu detaljnije opisati.

Neki od motiva za upotrebu ovih dizajn paterna su sljedeći problemi:

- Potreba za kreiranjem jedinstvenog objekta koji je lahko dostupan svim klasama
- Problem instanciranja više sličnih objekata
- Problem instanciranja objekata iz kompleksnih klasa
- Problem instanciranja i inicijalizacije objekata sa mnogo opcionalnih parametara

U nastavku slijedi detaljniji opis prethodno navedenih paterna koji su se iskoristili, kao i potencijalna mjesta na kojima bi se mogli upotrijebiti i drugi paterni u našem sistemu.

Prototype dizajn patern:

Prototype dizajn patern se koristi kada želimo da kreiramo kopiju objekta neke instancirane klase, tj. da izvršimo takozvano kloniranje objekta. Tokom kloniranja kreira se potpuno nova instanca koja je referencirana na drugi memorijski objekat. Drugim riječima, ovaj patern nam pomaže da postignemo duboko kopiranje (kloniranje) objekta na jednostavan način. Ovaj patern pronalazi primjenu u onim situacijama u kojima je cilj da se kopiranjem postigne potpuno novi objekat koji je identičan orginalnom objektu pri čemu su njihove reference na memorijske objekte različite. U nastavku slijedi opis problema i riješenja za koji je ovaj patern pronašao primjenu u našem sistemu.

Opis problema:

Dati sistem u okviru svoje funkcionalnosti za raspored treba da iz klase termina kreira objekte koji će se koristiti za predstavljanje i rad sa modelom termina. Termini se dijele u dvije skupine:

- Treninzi – predefinisani termini iz programa aktivnosti
- Dodatni termini – korisnički definisani termini

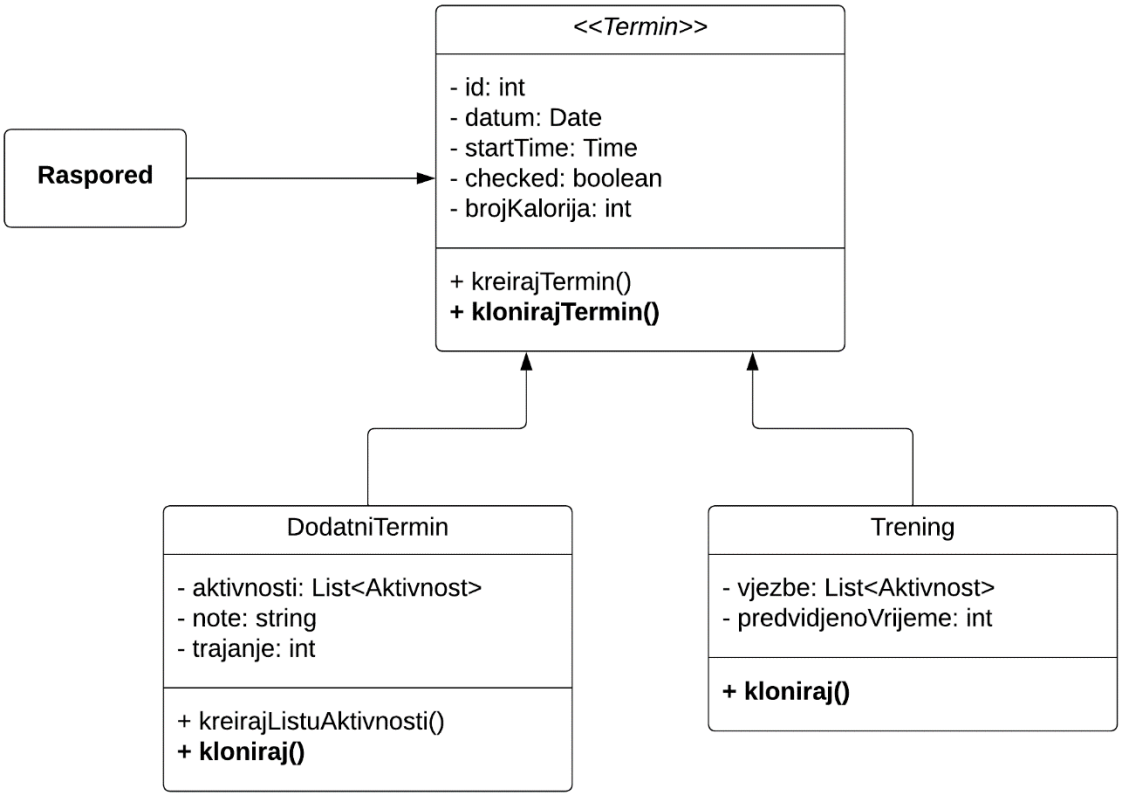
Korisnik tokom popunjavanja rasporeda, može da kreira dva ili više termina koja su identična. Problem u ovoj situaciji je kreiranje objekata koji su veoma slični na brz i jednostavan način, kao i ponovna upotreba postojećih termina.

Opis rješenja:

Za dati problem, kreacijski patern koji daje najbolje rješenje je Prototype dizajn patern. On nam omogućava da na jednostavan način kreiramo kopije već postojećih objekata koji modeliraju termine. Osim toga, ono zbog čega je ovaj patern dobro rješenje je njegova mogućnost da vrši dodatne izmjene nad klonom

objekta, što je korisno u situacijama kada korisnik želi da kreira termine koji pripadaju istoj kategoriji, ali se na primjer razlikuju u listi vježbi koje se izvode.

Na narednoj slici je prikazana struktura ovog paterna koja je primjenjena na naš sistem.



Elementi ovog paterna su:

- *Raspored* – je klijent klasa koja zahtjeva objekat
- *DodatniTermin*, *Trening* – su prototip klase koje implementiraju kloniranje
- *Termin* – je apstraktna klasa koja omogućava koniranje

Builder dizajn patern:

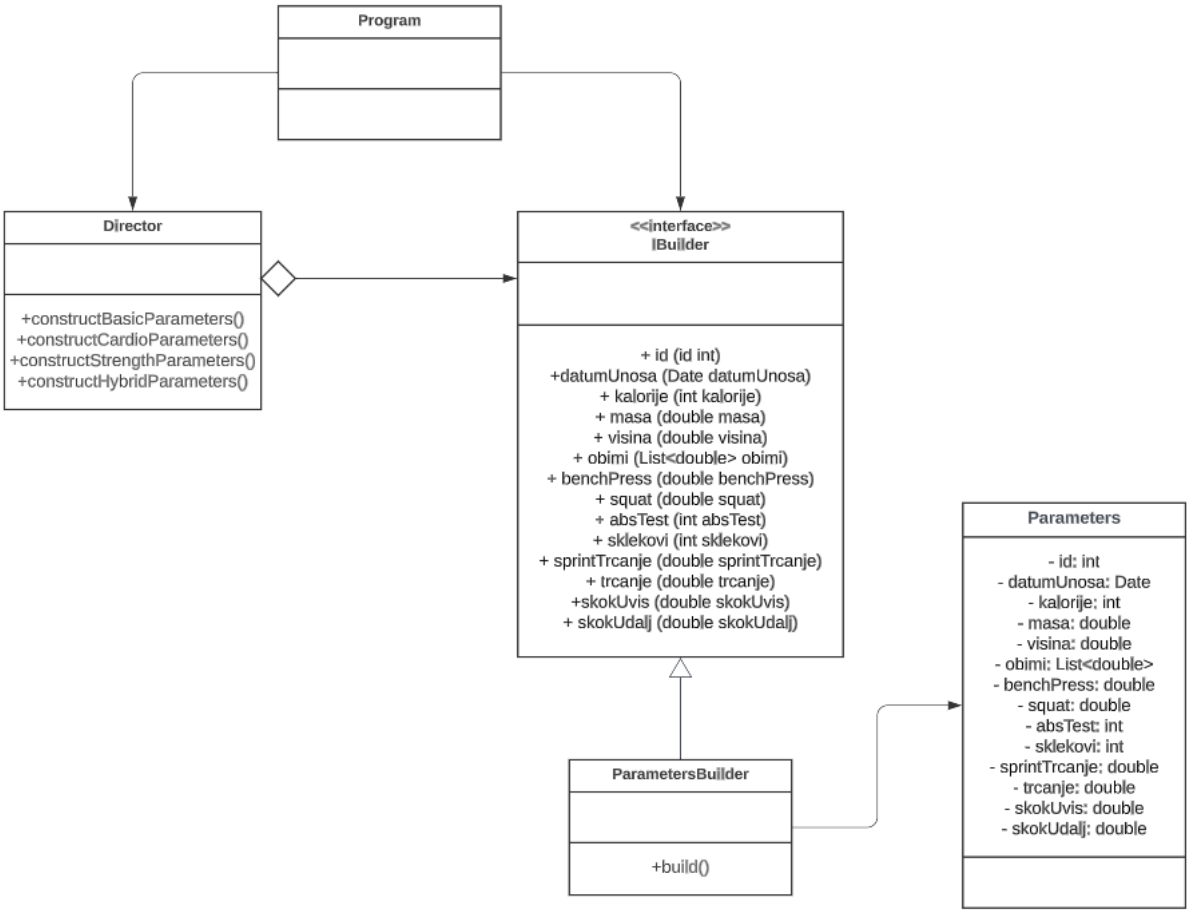
Ovaj dizajn patern nam daje mogućnost da na jednostavan način kreiramo instancu kompleksne klase koja tipično ima mnogo atributa koji su opcionalni prilikom instanciranja. Prednost ovog dizajn paterna je u tome što on ne zahtjeva posebno struktuiranje klasa u neke apstraktne strukture kao što su stablo. On zapravo omogućava da se iz takozvanih *fat klasa* (u smislu velike brojnosti atributa) kreira proizvoljan objekat, pri čemu se ne moraju inicijalizirati svi parametri. U narednom dijelu se opisuje problem, kao i riješenje u kojem se primjenio ovaj patern.

Opis problema:

Dati sistem omogućava praćenje parametara iz kojih se generišu rezultati kojeje postigao korisnik. Kako korisnik u toku upotrebe ovog sistema može da prati različite programe aktivnosti, to slijedi da je potrebno omogućiti unos samo onih parametara koji su relevantni za tekući program aktivnosti. Obzirom da postoji velika potreba za kreiranjem objekata koji će imati varijabilan broj parametara prilikom praćenja rezultata, slijedi da bi klasičan način kreiranja objekta sa *new* operatorom bio nezgrapan.

Opis rješenja:

Kao dobro rješenje za ovu situaciju je Builder dizajn patern. Ovim paternom postižemo da iz naše kompleksne klase *Parametri* na jednostavan način kreiramo objekte sa varijabilnim brojem parametara. Svaki objekat se može graditi tj. instancirati na različit način, pri čemu se željena struktura postiže sljedećom strukturom ovog paterna koja je primjenjena na naš sistem, što je predstavljeno na narednoj slici.



Struktura predstavljena na prethodnoj slici čine sljedeći elementi:

- Director – klasa koja sadrži neophodnu sekvencu koraka za kreiranje CardioParameters, StrengthParameters, HybridParameters te BasicParameters
- *Ibuilder* – interface koji u sebi sadrži korake potrebne za kreiranje produkta
- *ParametersBuilder* – obezbjeđuje različite implementacije za konstrukcijske korake
- *Parameters* – rezultirajući objekti
- *Program/User* – klasa koja pristupa kreiranju objekata pomoću Buildera

Singleton dizajn patern:

Sa ovim paternom se postiže kreiranje jednistvenog objekta koji je globalnog opsega. Njime omogućavamo da svako novo kreiranje instance se odnosi na jednu centralnu instancu koja je uvijek ista. Svoju primjenu u ovom sistemu, ovaj patern je našao indirektnim putem, što je opisano u nastavku.

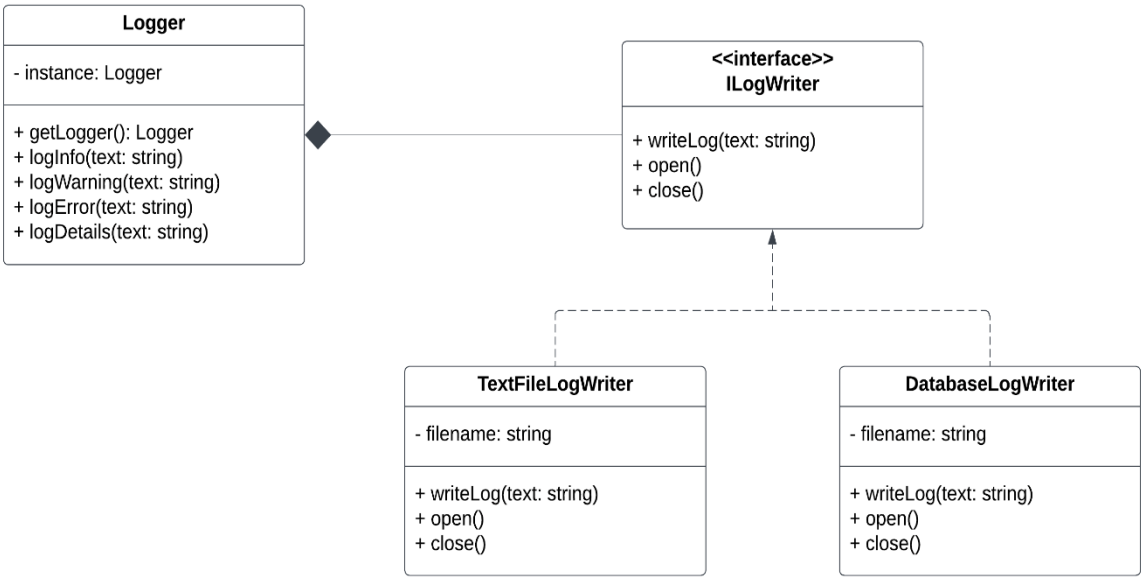
Opis problema:

Za rad sa različitim korisnicima, ovaj sistem pruža opciju prijave i registracije. Svaki korisnik u sistemu može da se prijavi više puta, pri čemu se kreiraju različiti log izvještaji. Prilikom svake prijave za korisnika se kreira nova instanca koju je potrebno ukloniti onda kada korisnik nije u sistemu. Iako su sve ove instance iste, i dalje slijedi potreba da se prilikom odjave korisnika uklone sve instance koje su kreirane što predstavlja problem pri radu sa velikim brojem takvih instanci.

Opis rješenja:

Kako bi se riješio ovaj problem, kao riješenje se može iskoristiti Singleton dizajn patern. Ovaj patern nam kao što je prethodno navedeno, pruža mogućnost da prilikom svakog kreiranja novog objekta uvijek dobijemo jedan objekat koji je jedinstven. Upotrebom Singleton paterna se reducira broj instanci koje su iste i koje se kreiraju, što olakšava posao prilikom uklanjanaja svih instanci koje su iste tj. uvijek se koristi jedna ista instanca. Osim toga, ovim paternom se postiže i dodatna ušteda memorijskih resursa kao i vremenskog odziva sistema, što su dodatni razlozi za upotrebu ovog paterna. U nastavku slijedi prikaz strukture ovog paterna koja je primjenjena za navedeni dio sistema.

Napomena: Kako se prilikom naše implementacije koristi eksterni server za rad sa bazom podataka koji nam pruža jedinstven rad sa korisnicima, to slijedi da je ovaj dizajn patern indirektno implementiran u ovaj sistem.



Struktura predstavljena na prethodnoj slici čine sljedeći elementi:

- *Logger* – je klasa za ispisivanje log-ova o promjenama u sistemu. Na primjer o kreiranju novih korisnika, loginu konkretnog korisnika. Ovi logovi za spašavaju ili u bazi podataka ili u text fajlu
- *ILogWriter* – je interfejs koji definiše metode za konkretne klase
- *TextFileLogWriter*, *DatabaseLogWriter* – su klase koje koriste singleton instancu

U nastavku su razmotrena dva kreacijska dizajn paterna koja nisu pronašla svoju konkretnu primjenu u našem sistemu, ali su razmotrene neke situacije u kojima bi se mogla pojaviti potreba za istim.

Factory Method dizajn patern:

Factory Method se koristi kada se proces kreiranja objekta želi ostaviti podklasama, ali je omogućeno da apstraktna klasa definira interfejs za stvaranje objekta. Podklase odlučuju kako će se ti objekti stvarati. Dobro ga je koristiti u sljedećim situacijama:

- kada želimo da kod bude neovisan o konkretnim klasama objekata koji se stvaraju i da koristi samo apstraktnu klasu ili interfejs
- kada želimo jednostavnost u dodavanju novih podklasa koje mogu stvarati objekte
- kada želimo da svaka podklasa ima kontrolu nad procesom instanciraja objekta
- kada želimo da naš kod upravlja s instancama preko zajedničkog interfejsa

U našem sistemu možemo ga primjeniti u sljedećim slučajevima:

- Osoba : Trener i Korisnik
- Parametri : Strength, Cardio i Hybrid
- Termin : Dodatni i Trening

Ukoliko je logika kreiranja objekta poprilično jednostavna onda je bolje koristi konstruktor.

Abstract Factory patern:

Ovaj dizajn pattern se primjenjuje u situacijama kada se vrši kreiranje (instanciranje) familije povezanih objekata bez specificiranja konkretnih klasa. Koristan je u situacijama kada se želi instancirati objekat iz strukture međusobno povezanih objekata, pri čemu se zavisnost o konkretnim klasama ne postiže. Ovaj dizajn patern nije pronašao svoju primjenu u našem sistemu jer u njemu ne postoji struktura klasa za koje bi se zahtijevala upotrebu više klasa koje bi predstavljale

fabrike za kreiranje različitih objekata iz strukture. Kako naš sistem nema potrebu za kreiranjem različitih varijacija objekata iz familije sličnih objekata, to slijedi da Abstract Factory pattern nije dobro rješenje.

Potencijalna situacija u kojoj bi se ovaj pattern mogao primjeniti u našem sistem bi bila kada bi naš sistem dodatno podržavao i klasu za različite zdravstvene i nutricionističke parametre.