Grupa2 – Zamger

dizajn patterni

Studenti:

Hamzić Huso

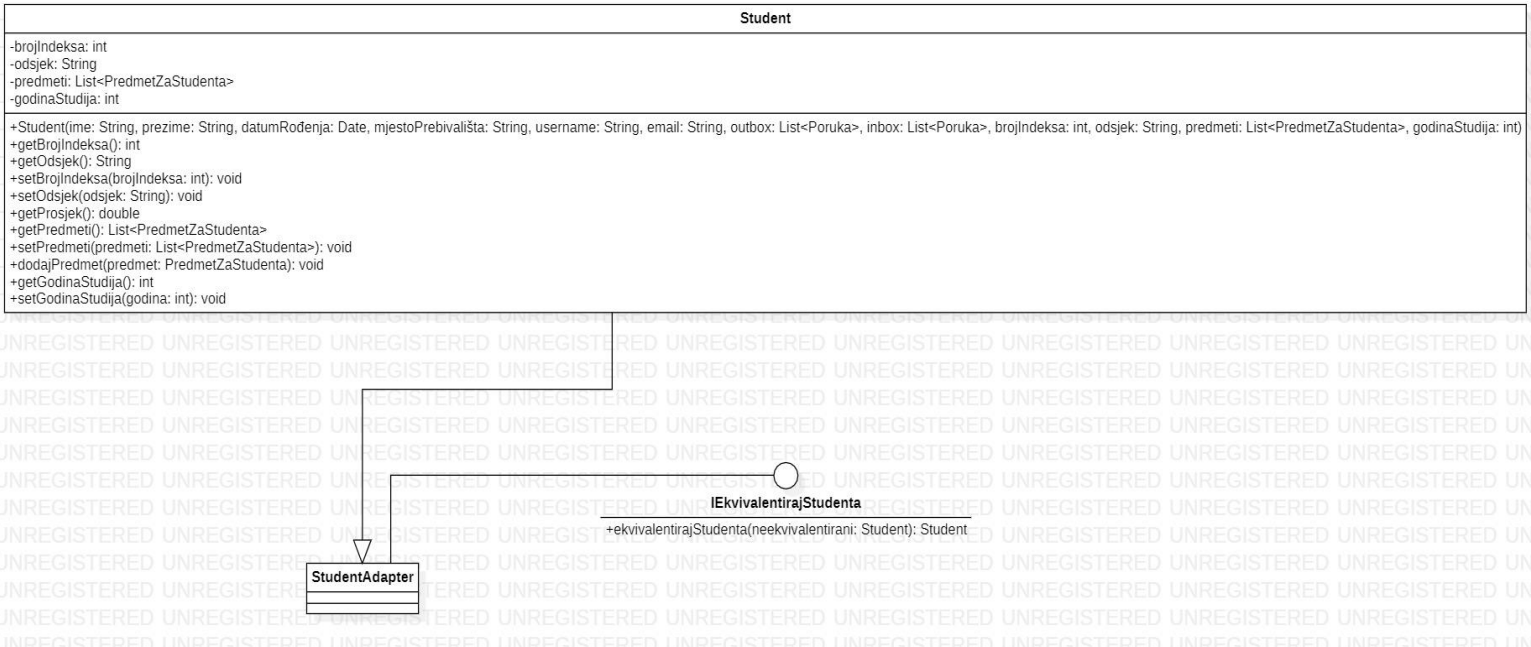
Handžić Rijad

Džumhur Paša

Strukturni patterni:

Adapter pattern: služi da se postojeći objekat prilagodi za korištenje na neki novi način u odnosu na postojeći rad, bez mijenjanja same definicije objekta

Što se tiče ovog patterna, nije ga bilo teško primjeniti u našem sistemu. Kako je sistem koncipiran tako da studentska služba može/mora unijeti korisnika u sistem kako bi isti mogao pristupiti svim funkcionalnostima, ono što smo morali omogućiti a i što je normalna pojava u realnom životu i konkretno kod pravog Zamgera je ta da student može upisati na primjer master studij čak i ako prethodno nije pohađao BSc studij na istom fakultetu. To dovodi do potencijalnih „problema“ koji bi se manifestovali kao nekonzistentni podaci u samoj bazi gdje se čuvaju rezultati svakog od studenata. Fakultet na kom je student završio raniji ciklus studija možda ima različit način bodovanja i vrednovanja različitih predmeta i na neki način to moramo ekvivalentirati sa načinom koji je zastupljen na fakultetu na kom je deployan ovaj naš sistem. To nas dovodi do toga da moramo nekako omogućiti da se u kontroleru zaduženom za upis studenta na fakultet ekvivalentiraju rezultati na predmetima koje je taj student prethodno slušao i naravno položio, bez izmjena u već postojećim klasama i ogromnih prepravki. Rješenje se nameće u vidu korištenja **adapter patterna** i to na sljedeći način:



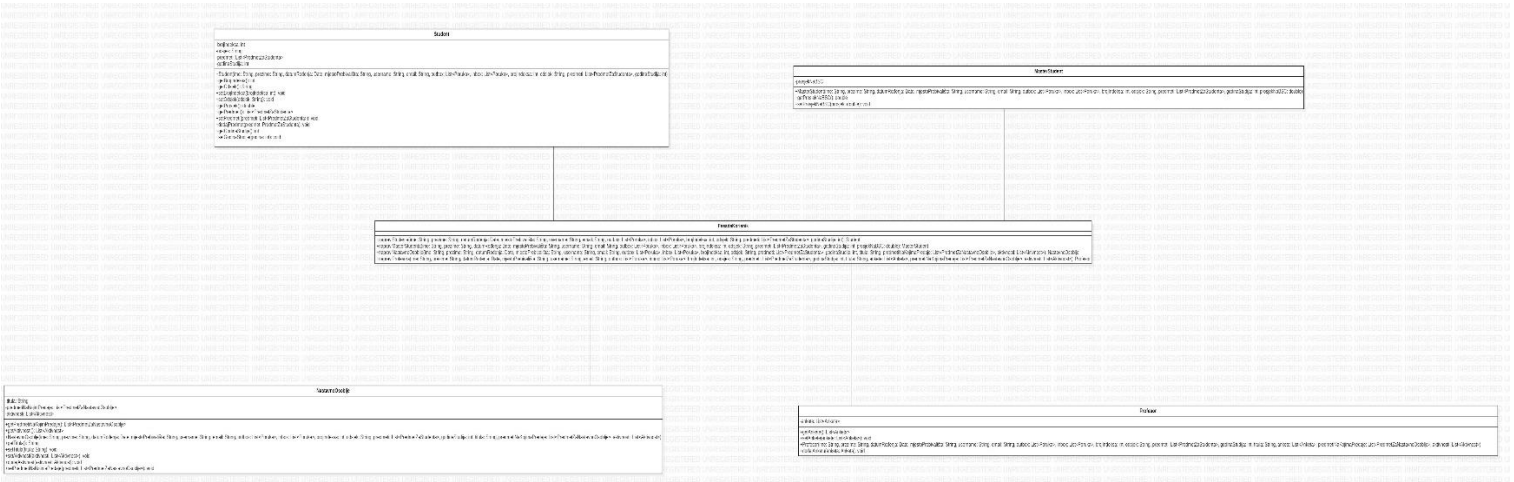
Napravljen je interface *IEkvivalentirajStudenta* koji definiše metodu ekvivalentirajStudenta koja će biti implementirana u klasi **StudentAdapter** koja nasljeđuje klasu Student(onako kako nalaže pattern). Ta implementacija će se sastojati od raznih skaliranja bodova svih predmeta koje je student slušao, kako bi se isti mogli porediti sa bodovima i ocjenama studenata koji se već ranije nalaze u bazi. Metoda će primati objekat tipa Student koji će predstavljati neekivalentiranog studenta, te će se on nakon toga ekvivalentirati te upisati u bazu. Odnosno poziv iz kontrolera bi išao nešto ovako:

```
StudentAdapter adapter = new StudentAdapter();

Student neekvivalentirani = new Student(...)

adapter.ekvivalentirajStudenta(neekvivalentirani).spremiUBazu();
```

Gdje bi se naravno ovo dešavalo samo ukoliko je na formi čekiran checkbox koji jasno naznačuje da student koji se trenutno unosi nije ranije studirao na ovom fakultetu već da dolazi sa neke druge institucije.

Fasadni pattern: služi kako bi se klijentima pojednostavilo korištenje kompleksnih sistema - klijenti vide samo fasadu, odnosno krajnji izgled objekta, dok je njegova unutrašnja struktura skrivena

Ovaj pattern je isto lahko primjenjiv na našem sistemu i to čak skoro pa na istom mjestu kao i **adapter pattern**. Sistem je osmišljen tako da će kontroler u jednoj metodi(neka event metoda) pokupiti informacije sa forme koji su prethodno uneseni, a koji se odnose na osobu koja se treba registrovati u sistem(upisati na fakultet u nekoj funkciji npr studenta ili nastavnog osoblja). Uz adekvatan odabir checkbox-ova na formi lahko se može zaključiti kojeg tipa taj korisnik mora biti(NastavnoOsoblje, Student, Profesor, MasterStudent), pa zašto ne bismo omogućili neku jednostavnost prilikom kreiranja istih. **Facade pattern** smo primjenili na sljedeći način:



(slika se ne vidi baš najbolje zbog rezolucije – poželjno je otvoriti istu na githubu)

Naime, omogućili smo da se može instancirati objekat tipa **FasadaKorisnik** koji će imati 4 metode i to metode za kreiranje instanci svih mogućih korisnika na sistemu, pozivima kao što su npr:

```
fasada.kreirajProfesora(...) //tri tačke predstavljaju sve
potrebne parametre za tu metodu.
```

Nakon čega metoda vraća objekte onih tipova koji su traženi, tako da korisnik uopšte ne mora znati šta se sve „pod haubom“ dešava. On samo pozove metode kreiranja koje mu vrate tražene objekte i to je to.

Nakon kreiranja npr. da se pozvala metoda `kreirajStudenta(...)` koja bi vratila objekat tipa **Student** koji bi mogao biti jedan od onih neekvivalentiranihi studenata o kojima smo pričali ranije, totalno validan bi bio sljedeći segment koda u nekoj od onAction metoda u kontroleru:

```
StudentAdapter adapter = new StudentAdapter();
FasadaKorisnik fa = new FasadaKorisnik();
adapter.ekvivalentirajStudenta(fa.kreirajStudenta(...)).spremiUBazu();
```

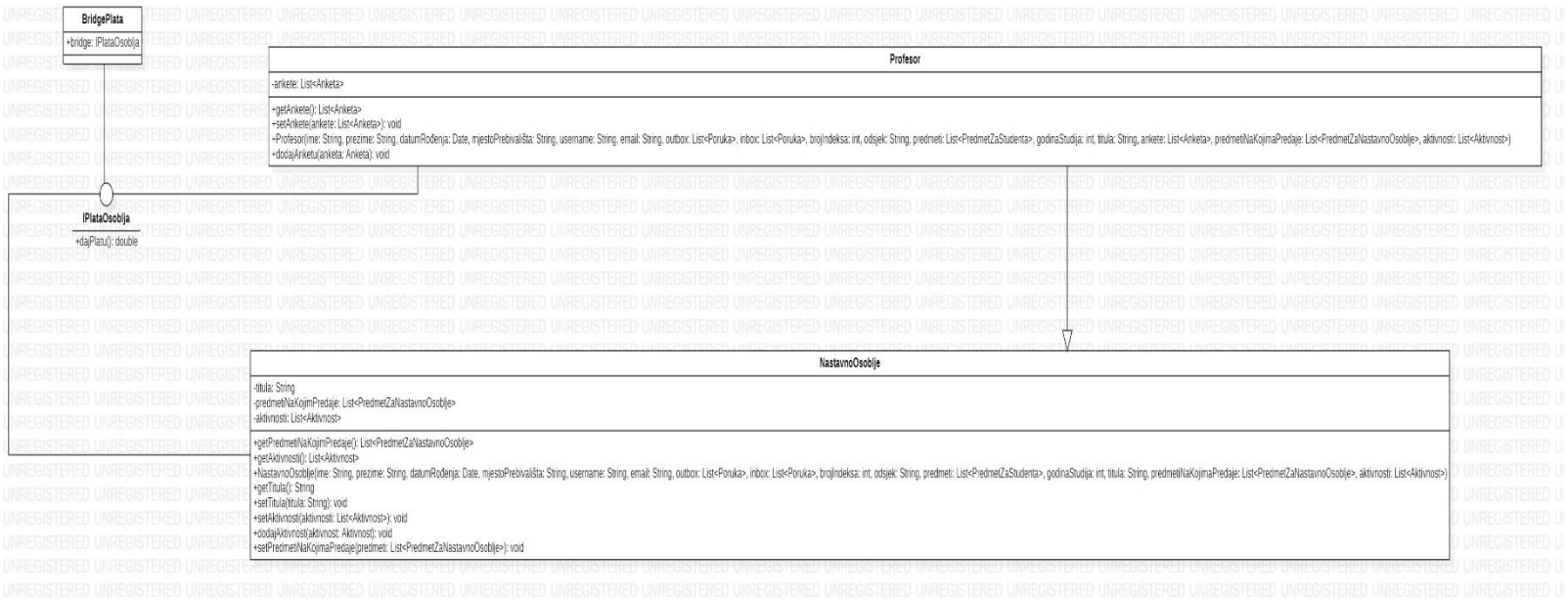
Dekorator pattern: <u>služi za omogućavanja različitih nadogradnji objektima koji svi u osnovi predstavljaju jednu vrstu objekta (odnosno, koji imaju istu osnovu) – umjesto da se definiše veliki broj izvedenih klasa, dovoljno je omogućiti različito dekoriranje objekata (tj. dodavanje različitih detalja)</u>

Na način kako je ovaj pattern objašnjen na tutorijalu uz više pregledavanja videozapisa sa tutorijala, autori nisu bili u mogućnosti naći njegovu primjenu u njihovom sistemu(što ne znači da je nema). Naime primjećeno je da se ovaj pattern koristi kada se treba više funkcionalnosti nad nekim istim objektom(istog tipa) dinamički dodavati/odnosno koristiti „u hodu“ a što i samo ime kaže. Na tutorijalu je prezentovan primjer sa slikom gdje je korisniku omogućeno da sa slikom radi svašta(razne funkcionalnosti, odnosno u slučaju slika, dekoracija istih). Kako je naš sistem zamišljen da se drugačije ponaša od vrste korisnika koji je trenutno prijavljen na njega kao neki *currentUser* uočavamo da dosta vrsta korisnika neće moći haman nikako mijenjati većinu objekata(za razliku od **Studentske** i **NastavnogOsoblja** npr). Javila se ideja da se možda tim korisnicima omogući pregršt nekih stvari koje one mogu raditi(kao što na tutorijalu korisnik edituje sliku, tako npr studentska služba može editovati studenta, njegove lične podatke itd...), ali ipak zbog jednostavnosti sistema autori su se odlučili da takvim stvarima ostave minimalan prostor odnosno da te izmjene ne budu toliko kompleksne i da se mogu završiti sa par *setter-a* za određenog studenta i pozivanjem samih metoda koje klasa **Student** ima a koje to omogućuju. Autori su mišljenja da nema potrebe za daljnim izdvajanjem ovako jednostavnih metoda u razne interfejse(kao na tutorijalu) jer su ti *edit-i* minorni kao što je već rečeno pa samim time i da nema potrebe za primjenom ovog patterna.

Bridge pattern: služi kako bi se apstrakcija nekog objekta odvojila od njegove implementacije - omogućava se nadogradnja modela klasa u budućnosti te osigurava da se neće morati vršiti određene promjene u postojećim klasama

Ovaj pattern je također našao primjenu u našem sistemu. Naime nije na odmet imati pri ruci funkcionalnost koja može računati platu za korisnike na sistemu koji zarađuju istu(zbog statistike i neke analitike) međutim primjetimo da na sistemu imamo više različitih korisnika koji u realnom svijetu nemaju isti iznos plate odnosno da se isti drugačije računaju. Tu nam pomaže **Bridge pattern** koji omogućuje da se apstrakcija(u ovom slučaju način računanja plate) odvoji od

implementacije te metode. Kako na sistemu postoje dva tipa korisnika koji primaju plaću(**NastavnoOsoblje** i **Profesor**) zaključujemo da će ove metode morati implementirati neki interfejs koji će računati njihovu plaću ali na drugačiji način jer u realnom svijetu to funkcioniše na sljedeći način. Akademski radnici imaju neki fiksni dio plate koje je za sve isti no međutim na taj dio se dodaje dodatak koji se naknadno računa po određenim koeficijentima. Taj fiksni dio se zna koliki je no vidimo da ovaj ostatak plate varira od tipa korisnika(profesori će imati drugačiji koeficijent od preostalog nastavnog osoblja). Prateći sve ovo što je rečeno, ovaj pattern je implementiran ovako na našem sistemu:



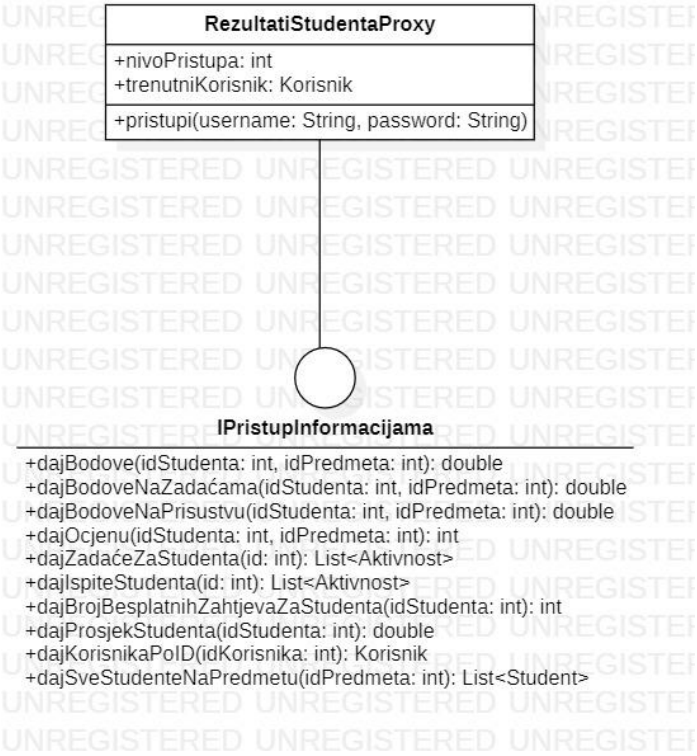
Imamo interface *IPlataOsoblja* kojeg implementiraju klase **Profesor** i **NastavnoOsoblje** naravno svaka na svoj način, ranije je navedeno zašto(računaju preostali dio svoje plate u zavisnosti od svojih različitih koeficijenata). To sve spajamo u klasi **BridgePlata** koji ima atribut tipa *IPlataOsoblja* koja također implementira istoimeni interfejs te na dodatni dio plate koji je izračunat u zavisnosti u metodi `dajPlatu()` ove klase se računa konačna plata koju određeni član akademskog osoblja prima, i to na način da se na njegov dodatak(koji zavisi od koeficijenta te osobe) dodaje fiksni dio plate koji je regulisan na toj instuticiji/u toj državi i tako.

Kompozitni pattern: služi za kreiranje hijerarhije objekata - koristi se kada svi objekti imaju različite implementacije nekih metoda, no potrebno im je svima pristupati na isti način

Što se ovog patterna tiče, mogao bi se iskoristiti na istom mjestu kao i **Bridge pattern** da su autori sistema odlučili da plate računaju ne po nekim koeficijentima za profesore i za nastavno osoblje već da plata svakog od članova nastavnog osoblja zavisi npr od broja studenata kojima isti predaju, ili od broja predmeta koje isti drže. Tada bi naravno koristili onaj „jaki" polimorfizam prilikom skupljanja podataka o plaći čitavog ansambla klasično **foreach** petljom koristeći interfejs kojeg su sve klase osoblja naslijedile. Kako je **Bridge pattern** iskorišten, a i prepoznato je i hipotetičko mjesto uporabe ovog patterna sa razmatranjem istog možemo završiti(u suštini naš sistem uopće nema puno interface-a koji su implementirani od strane različitih klasa da bi ovaj pattern mogao naći upotrebu negdje drugo).

Proxy pattern: služi za dodatno osiguravanje objekata od pogrešne ili zlonamjerne upotrebe - omogućava se kontrola pristupa objektima, te se onemogućava manipulacija objektima ukoliko korisnik nema prava pristupa traženom objektu

Ovaj pattern je idealan za uporabu na ovom sistemu, koji je i zamišljen da se koristi i da se različito ponaša u zavisnosti koju ulogu trenutni korisnik ima na sistemu. Naravno očita upotreba ovog patterna je u dijelu pristupa nekim „senzitivnijim" podacima konkretno nekog studenta. Nema smisla da profesor ili nastavno osoblje koje nije zaduženo na određenom predmetu mogu vidjeti/imaju uvid u rezultate nekog studenta/studenata na tom istom predmetu. Primjetno je da će se morati uspostaviti neki nivoi pristupa koji će tačno definisati kojim podacima ko na sistemu može pristupiti odnosno koje podatke ko može dobiti. To rješavamo korištenjem **proxy patterna** i to konkretno na sljedeći način:



Imamo interface *IPristupInformacijama* koji ima dosta nekih funkcionalnosti ali kojima ne bi svaki korisnik na sistemu trebao imati pristup. Klasa **RezultatiStudentaProxy** implementira ovaj interface te u sebi sadrži atribut nivoPristupa i trenutniKorisnik. Kada se sa login forme pokupe podaci o korisniku tada će početi akcija login-a odnosno metoda `pristupi(...)` unutar klase **RezultatiStudentaProxy** će provjeriti da li postoji neko sa istim pristupnim podacima u bazi, te u slučaju ako postoji očitat će kojeg je tipa taj korisnik iz tabele. U zavisnosti kojeg je tipa, tako će se i instancirati objekat trenutniKorisnik (polimorfizam je ovdje iskorišten). Nakon toga u istoj metodi će se tome korisniku dodijeliti određeni nivo pristupa pomoću kojeg će isti moći pozivati odnosno koristiti funkcionalnosti koje će implementirati ova klasa(većina funkcionalnosti radi sa bazom). Pa neki blok koda bi glasio ovako:

```
RezultatiStudentaProxy proxy = new RezultatiStudentaProxy();

proxy.pristupi(„hhamzic1“, „ILoveOOAD<3“); /*hhamzic1 je Student u sistemu, tako da će trenutniKorisnik biti tipa Student dok će nivo pristupa biti 1(Student će moći pristupiti samo svojim rezultatima na određenim predmetima) pa se može pozvati metoda npr: */

proxy.dajOcjenu(1,1);
```

```
/* gdje će se sad, kako hhamzic1 ima nivo pristupa 1 provjeriti da li
je idStudenta koji je poslao kao parametar, jednak njegovom id-u(jer on
može vidjeti samo svoje rezultate), ako jest nakon toga se provjerava
da li se idPredmeta koji je poslao kao drugi parametar nalazi u kolekciji
predmeta kojih je on slušao(jer nema smisla da traži ocjenu sa predmeta
kog nije slušao). Ako je ovo sve ispunjeno, iz baze će se vratiti ocjena
koju je hhamzic1 postigao na predmetu sa id-om 1(OOAD npr). */
```

Analogno bi npr. studentska služba imala najveći pristup i mogla bi dobiti podatke svih studenata na svim predmetima. Uočavamo da je upotreba ovog patterna imperativ te je fakat korisno(centralizuje na neki način čitav sistem te su „role-ovi“ pregledni).

Flyweight pattern: koristi se kako bi se onemogućilo bespotrebno stvaranje velikog broja instanci objekata koji svi u suštini predstavljaju jedan objekat - samo ukoliko postoji potreba za kreiranjem specifičnog objekta sa jedinstvenim karakteristikama (tzv. specifično stanje), vrši se njegova instantacija, dok se u svim ostalim slučajevima koristi postojeća opća instanca objekta (tzv. bezlično stanje).

Kako je navedeno u tutorijalu i primjer sa igricom i bespotrebnim pravljenjem istih karaktera, primjetimo da ovaj pattern ne možemo iskoristiti u našem sistemu iz jednog jednostavnog razloga. Svaki korisnik na sistemu je unikatan/jedinstven. Svaki student ima jedinstven broj indeksa, predmete koje sluša i tako dalje, tako da ne možemo nikako upotrijebiti ono kopiranje objekata ako su oni isti (npr karikature u igrici kao što je navedeno u tutorijalu, da ne pravimo svaki put novu figuru). Pored korisnika imamo npr predmete, ankete i ostale stvari, ali opet je to sve unikatno. Svaki predmet je unikatan(različit broj zadaća, različite zadaće/ispiti, studenti na njemu itd...), svaka anketa je unikatna(različita pitanja itd...). Nažalost poboljšanja o kojima govori ovaj pattern ne možemo iskoristiti u našem sistemu, no međutim to ne pravi neku veliku razliku zato što se nikad neće za vrijeme trajanja jedne sesije instancirati više od 1 korisnika i određenog broja predmeta.

Kreacijski patterni: