12.11.2021.

Objektno-orijentisano programiranje 2

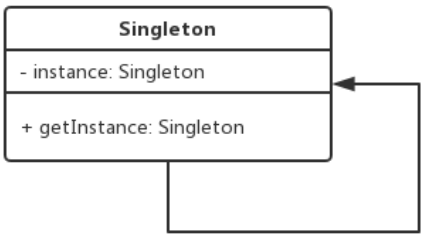
Unikat

Motivacija i osnovna ideja:

Nekada je potrebno imati klasu za koju želimo da imamo samo jednu instancu. Na primer, često u aplikacijama imamo klase za konfigurisanje nekog ponašanja na nivou celog programa ili za upravljanje resursima celog programa. Nekada je prosto logika klase takva da nema potrebe za postojanjem više od jednog objekta date klase – primer za to su fabrike iz projektnog uzorka apstraktna fabrika (treba nam po jedna od svake konkretne fabrike). U ovakvim situacijama možemo koristiti projektni uzorak *Unikat* (engl. Singleton).

Da osiguramo postojanje samo jednog objekta date klase, tipičan pristup je da uvek pristupamo statičkom članu date klase preko statičke metode koju često nazivamo getInstance() čiji je zadatak da samo pri prvom pozivu kreira objekat željenog tipa i da prilikom svakog poziva, uključujući i prvi, vrati taj objekat. Da bismo sprečili kreiranje objekta na druge načine potrebno je onemogućiti pozive konstruktora van klase kao i operatora dodele vrednosti (postavljajući im nivo pristupa na private ili protected).

UML dijagram projektnog uzorka Unikat dat je na slici 1.



Slika 1 Projektni uzorak Unikat

Zadaci:

1. Implementirati opšti oblik projektnog uzorka Unikat tako da getInstance() vraća pokazivač na Singleton.
2. Implementirati opšti oblik projektnog uzorka Unikat tako da getInstance() vraća referencu na Singleton.
3. Kreirati klase BaseSingleton i DerivedSingleton1 i DerivedSignleton2 koje nasleđuju klasu BaseSingleton. Omogućiti da može postojati samo po jedan objekat obe izvedene klase.

4. Modifikovati kod zadataka 6-7 od prošle nedelje tako da konkretne fabrike budu unikati. Ukloniti atribut fabrika iz konkretnih graditelja, a fabrike koristiti pozivom TipFabrike::getInstance().

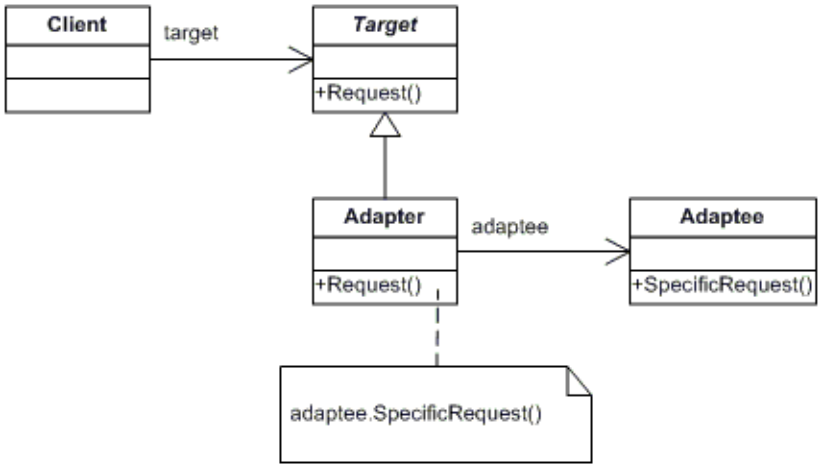
Adapter

Motivacija i osnovna ideja:

Pretpostavimo da imamo hijerarhiju klasa koju smo implementirali u skladu sa nekim željenim interfejsom i da želimo da istoj hijerarhiji pridodamo klasu koja poseduje drugačiji interfejs. Pretpostavimo i da spornu klasu ne želimo ili prosto ne možemo da modifikujemo (možda koristimo nečiju biblioteku i nemamo mogućnost modifikacije koda). Takođe, ne želimo da zbog toga menjamo svoje postojeće klase. Potrebno je da interfejs sporne klase nekako prilagodimo, *adaptiramo*, ciljnom interfejsu. Strukturni projektni uzorak adapter namenjen je razrešavanju ovakvih situacija.

Postoje dve verzije ovog projektnog uzorka – objektni i klasni adapter. U oba slučaja potrebno je definisati klasu Adapter koja nasleđuje klasu u kojoj je deklarisan ciljni interfejs (zvaćemo je Target klasa). Kod objektnog adaptera, klasa Adapter agregira objekat klase koju treba adaptirati i pozive ciljnog interfejsa preusmerava agregiranom objektu. Klasni adapter implementira se višestrukim nasleđivanjem. Osim što Adapter treba da nasledi klasu Target, on privatno nasleđuje i klasu objekta koji treba adaptirati, a kod poziva ciljnih metoda vrši se delegiranje poziva osnovnoj klasi adaptiranog objekta.

Učesnici projektnog uzorka Adapter (objektni) i njihova komunikacija dati su UML dijagramom prikazanim na slici 2.



Slika 2 Projektni uzorak Adapter (objektni)

Zadaci:

5. Implementirati opšti oblik projektnog uzorka Adapter na osnovu UML dijagrama sa slike 2. Dodati i klasu Adapter2 koja je implementirana kao klasni adapter.
6. Kreirati klasu ShapeDrawerV1 i dve metode drawRectangle i drawCircle.
   Pretpostavimo da hoćemo da izmenimo naš softver novim klasama koje bi se drugačije koristile, ali da zbog postojećih korisnika želimo da zadržimo i staru implementaciju. Kreirati nezavisnu apstraktnu klasu ShapeDrawerV2 koja ima jednu čistu virtuelnu funkciju draw(ShapeType). Pritom, stara implementacija korektno radi i nećemo da pravimo novu implementaciju "od nule", već da korisitmo kod koji smo već implementirali. Problem ćemo rešiti kreirajući adapter kojim se omogućuje korišćenje stare klase ShapeDrawerV1, ali komuniciranjem preko novog interfejsa iz klase ShapeDrawerV2.
7. Adapter se (u neznatno izmenjenom obliku) može koristiti i da omogući korišćenje potpuno nezavisnih klasa zajedničkim interfejsom.
   Kreirati apstraktnu klasu Target sa čisto virtuelnom metodom print(std::string) i tri nezavisne klase A, B i C sa metodama printA(std::string), printB(std::string), printC(std::string) (svaka metoda u svojoj klasi). Kreirati šablonsku klasu PrintAdapter koja kao parameter šablona ima klasu koju treba da adaptira. PrintAdapter implementirati kao objektni adapter, a prilikom kreiranja proslediti kao argument pokazivač na metodu koju treba da izvrši pozivom na print. Testirati različite tipove adaptera u main funkciji.
   Podsetnik:

   Ako imamo funkciju returnType f(Arg1Type a, Arg2Type b), pokazivač naziva ptr na funkciju koja vraća rezultat tipa returnType i ima dva argumenta tipa Arg1Type i Arg2Type deklarisali bismo kao:
   returnType (*ptr)(Arg1Type, Arg2Type), a mogli bismo ga inicijalizovati sa:
   ptr = &f;
   Funkciju f preko pokazivača ptr pozvali bismo sa ptr(a, b);

   Ako je u pitanju metoda klase T (koja vraća returnType i ima dva argumenta kao ranije), odgovarajući pokazivač imena ptr na metodu deklariše se kao:
   returnType (T:: *ptr)(Arg1Type, Arg2Type);
   Adresu metode m klase T dobijamo kao &T::m, a poziv metode ukoliko imamo dostupan i pokazivač *obj na objekat klase T vršimo sa:
   (obj->*ptr)(a, b);