21.01.2022.

Objektno-orijentisano programiranje 2

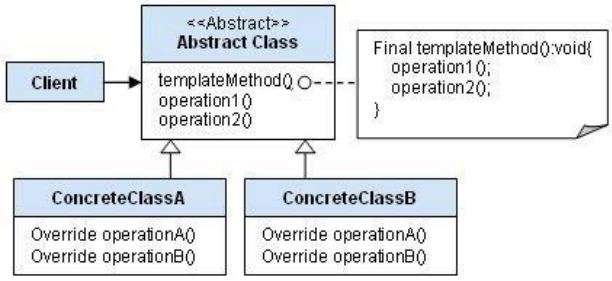
Šablonski metod

Motivacija i osnovna ideja:

Pretpostavimo da je potrebno implementirati neku metodu (algoritam) čije izvršenje možemo da podelimo na korake. Izvedene klase mogu imati drugačije ponašanje te metode, ali koraci koje treba primeniti su isti (doduše drugačije implementirani). Da bismo bili sigurni da će izvedene klase zaista primenjivati iste korake i to u dobrom redosledu, možemo iskoristiti šablonski metod. Ideja šablonskog metoda je da sam algoritam kao niz koraka ostane fiksan u svim izvedenim klasama, a da u izvedenim klasama samo menjamo ponašanja pojedinih (ili svih) koraka algoritma koje ćemo izdvojiti kao zasebne metode. Metodu koja implementira algoritam pozivajući metode koje su u izvedenim klasama (možda[1]) preklopljene nazivamo šablonskom metodom.

Tipični primeri primene su potezne igre kod kojih se svaki potez sastoji iz nekoliko koraka, aplikacije sa grafičkim interfejsom čiji izgled/ponašanje treba da zavise od uređaja na kome se pokreću, klase za eksportovanje izvrštaja u različitim formatima itd.

UML dijagram projektnog uzorka *Šablonski metod* dat je na slici 1.



Slika 1: Projektni uzorak Šablonski metod

Zadaci:

1. Implementirati opšti oblik projektnog uzorka šablonski metod u skladu sa UML dijagramom sa slike 1.
2. Napisati klasu GUIAplikacija i izvedene klase MobilnaAplikacija i DesktopAplikacija. Iscrtavanje aplikacije, bilo da je mobilna ili desktop, podrazumeva iscrtavanje zaglavlja, iscrtavanje menija i osnovnog prostora za rad.
3. Napisati apstraktnu klasu graf i metode za pretragu u dubinu i širinu (DFS i BFS). Metode u svojoj implementaciji treba da se oslanjaju na metodu *dajSusede(int v)* koja vraća listu (vector) suseda čvora v. Izvesti dve konkretne klase: GrafMatSusedstva i

[1] Nekada, u zavisnosti od primene, koraci algoritma, tj. šablonske metode, mogu se implementirati i u osnovnoj klasi, kao podrazumevano ponašanje algoritma. U izvedenim klasama opciono možemo da preklopimo određeni broj koraka/metoda.

GrafListaSusedstva koje predstavljaju različite implementacije grafa u računaru i koje preklapaju metodu *dajSusede(int v)*.

Posetilac

Motivacija i osnovna ideja:

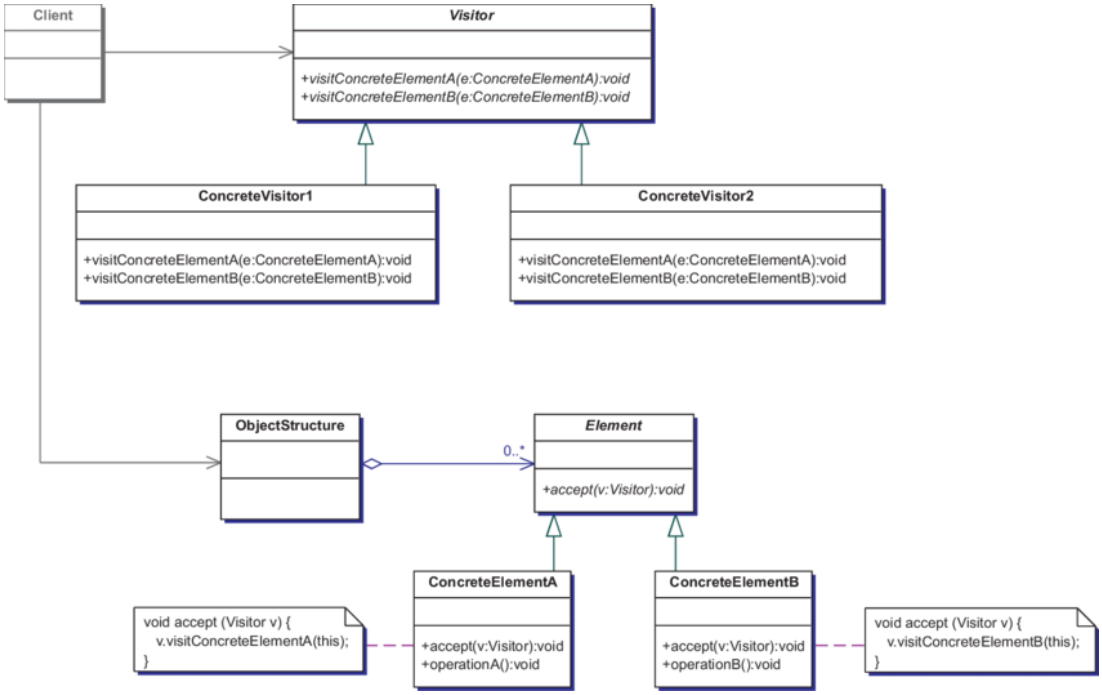
Pretpostavimo da imamo hijerarhiju klasa kojoj želimo da dodamo neku novu funkcionalnost (metodu). Standardni pristup je da željenu metodu deklarišemo u osnovnoj klasi i implementiramo u svakoj izvedenoj klasi. Ako je u pitanju samo jedna nova funkcionalnost, ovaj standardni pristup nudi zadovoljavajuće rešenje. Međutim, šta ako želimo da dodamo nekoliko novih funkcionalnosti? Da li treba dodati nekoliko novih metoda u svim klasama? To i dalje jeste jedna opcija, ali ne i poželjna. Naime, naša hijerarhija klasa onda postaje opterećena novim funkcionalnostima, kompleksna i podložna greškama. U objektno-orijentisanom programiranju treba što više težiti tzv. Single-responsibility principu, po kome svaka klasa treba da ima samo jednu odgovornost (funkcionalnost).

Projektni uzorak *Posetilac* (engl. *Visitor*) nudi alternativno (i bolje) rešenje za opisani problem. Svaka nova funkcionalnost biće implementirana u zasebnoj klasi. Naravno, ne možemo "pobeći" od toga da za različite klase iz naše originalne hijerarhije obezbedimo različito izvršenje logike te nove funkcionalnosti, ali sada ćemo tu logiku prebaciti u klasu predviđenu za datu funkcionalnost koju ćemo nadalje zvati *Posetilac*. Posetilac treba da "ume" da *poseti* različite objekte iz naše početne hijerarhije klasa, a *posećivanje* je ovde izraz koji koristimo za to da se operacija (nova funkcionalnost) primeni nad željenim objektom – pritom ta funkcionalnost nije implementirana u klasi tog objekta, već u klasi *Posetioca.*

Naravno, kako želimo da omogućimo dodavanje različitih funkcionalnosti, imaćemo zasebnu novu hijerarhiju klasa *Posetilaca.* Konačno, kako da zaista pozovemo novu funkcionalnost, tj. kako da objekti iz naše ciljne klase zaista izvrše svoje "nove metode"? Jedina izmena u prvobitnim klasama je što moraju imati samo jednu novu metodu (u svakoj izvedenoj klasi mora da se implementira), a to je metoda kojoj prosleđujemo posetioca i šaljemo mu zahtev da poseti taj objekat. Ovaj takozvani Double dispatch pristup omogućuje nam da zadržimo dinamički polimorfizam i da se nove funkcionalnosti iz hijerarhije klasa Posetioca pozivaju baš za one odgovarajuće klase iz naše početne hijerarhije.

Na primer, recimo da iz klase A imamo izvedene klase B i C i da želimo dodati funkcionalnosti f i g tim klasama. Kreiraćemo klase Posetilac i PosetilacF, PosetilacG (izvedene iz Posetilac) sa metodama poseti(B) i poseti(C) (pretpostavljamo da je A apsraktna, pa nam ne treba poseti(A)). Klase A, B, C imaće metode izvršiOperaciju(Posetilac) (obično se ta metoda zove *prihvati* (engl. *accept*)). U klasi A ta metoda može biti čisto virtualna, dok će se u klasama B i C svesti na prosti poziv p->poseti(*this), gde je p pokazivač na Posetioca.

Učesnici projektnog uzorka *Posetilac* i njihova komunikacija dati su UML dijagramom prikazanim na slici 2.



Slika 2: Projektni uzorak Posetilac

Zadaci:

4. Implementirati opšti oblik projektnog uzorka Posetilac u skladu sa UML dijagramom sa slike 2.
5. Napisati klase Nezaposleni i Zaposleni, izvedene iz klase Osoba i omogućiti lako dodavanje novih funkcionalnosti pomoću različitih posetilaca. Implementirati posetioce za rad, odmor i eksportovanje u fajl.
6. Napisati klasu BinarnoStablo i implementirati je kao kompozit. Dodati posetioce za određivanje veličine stabla, minimalnog elementa, maksimalnog, sume.