Druga laboratorijska vježba iz Oblikovnih obrazaca u programiranju

(načela oblikovanja, strategija, promatrač, dekorator)

1) Prevedi i isprobaj program zadan na stranici predavanja "Tehnike: C++". Promijeni redak:

```
Derived d;
```

u

```
Derived2 d;
```

Dodaj još i:

```
#include "derived2.hpp"
```

Pokreni program.

Implementiraj novu izvedenu klasu Derived3 koja nasljeđuje Base. Sučelje stavi u datoteku `derived3.hpp`, implementacijsku datoteku ne moraš imati (sve metode mogu biti "umetnute", `inline`). Sada u glavnom programu uključi `derived3.hpp`, te umjesto `Derived2` navedi `Derived3`. Pokreni program.

Stara implementacija klijenta radi s novom izvedbom osnovne klase. Klijenta nismo morali ni pipnuti (čak ni ponovo prevesti!). Koje načelo kaže da *stari kod* treba moći koristiti *novi kod*?

Za strpljive: napišite odgovarajuće programe u Pythonu i Javi.

2) Prevedi i isprobaj priloženi dopunjeni program s predavanja (str.~``Logička načela: NBP i proceduralni stil?'').

```
#include <iostream>
#include <assert.h>

struct Point{
  int x; int y;
};
struct Shape{
  enum EType {circle, square};
  EType type_;
};
struct Circle{
   Shape::EType type_;
   double radius_;
   Point center_;
};
struct Square{
   Shape::EType type_;
   double side_;
   Point center_;
};
void drawSquare(struct Square*){
  std::cerr <<"in drawSquare\n";
}
void drawCircle(struct Circle*){
  std::cerr <<"in drawCircle\n";
}
```

```
void drawShapes(Shape** shapes, int n){
  for (int i=0; i<n; ++i){
    struct Shape* s = shapes[i];
    switch (s->type_){
    case Shape::square:
      drawSquare((struct Square*)s);
      break;
    case Shape::circle:
      drawCircle((struct Circle*)s);
      break;
    default:
      assert(0);
      exit(0);
    }
  }
}
int main(){
  Shape* shapes[4];
  shapes[0]=(Shape*)new Circle;
  shapes[0]->type_=Shape::circle;
  shapes[1]=(Shape*)new Square;
  shapes[1]->type_=Shape::square;
  shapes[2]=(Shape*)new Square;
  shapes[2]->type_=Shape::square;
  shapes[3]=(Shape*)new Circle;
  shapes[3]->type_=Shape::circle;

  drawShapes(shapes, 4);
}
```

Dodaj metodu moveShapes koja pomiče oblike zadane prvim argumentom za translacijski pomak određen ostalim argumentima. Ispitaj dodanu funkcionalnost.

Dodaj razred Rhomb. Dodaj jedan objekt tipa Rhomb u listu objekata u main(). Sjeti se, sad moramo promijeniti i drawShapes().

Ovo je domino-efekt (krutost), kojeg ćemo kasnije pokušati zauzdati. Za probu, zaboravi adekvatno promijeniti moveShapes(). Isprobaj ponovo. Sad bi moveShapes trebao "puknuti". To je krhkost uzrokovana redundancijom. Ni to ne želimo imati u programu.

Konačno, implementiraj rješenje s predavanja, i komentiraj njegova svojstva.

3) Prevedi i pokreni program na stranici ``Logička načela: NIO, injekcija ovisnosti'' (u izvorni kod je potrebno dodati deklaracije i implementacije razreda AbstractDatabase i MockDatabase). Objasni načela inverzije i injekcije ovisnosti.

4. Razmatramo komponentu `DistributionTester` čiji zadatak je generirati prikladni niz cijelih brojeva te ispisati 10., 20., ..., i 90. percentil njihove distribucije. Generiranje cijelih brojeva trebalo bi biti podržano na svaki od sljedećih načina:

- slijedno, u ovisnosti o zadanim granicama intervala i koraku uvećanja;
- slučajno, u ovisnosti o zadanim parametrima normalne distribucije i željenom broju elemenata;
- kao Fibonaccijev niz u ovisnosti o zadanom ukupnom broju elemenata.

Komponenta `DistributionTester` također treba podržavati određivanje p-tog percentila distribucije zadanog niza cijelih brojeva na svaki od sljedećih načina:

- kao element čiji je položaj u sortiranom nizu (počevši od 1) najbliži položaju percentila `n_p` definiranog s `n_p = p*N/100 + 0.5`, gdje N odgovara broju elemenata; primjerice,

80. percentil niza (1,10,50) bi u tom slučaju bio 50 (detaljnije).
- kao interpoliranu vrijednost između elemenata `v[i]` i `v[i+1]` za čije percentilne položaje vrijedi `p(v[i]) < p < p(v[i+1])`; percentilni položaj elementa `v_i` na rednom broju `i` računamo kao `p(v[i]) = 100*(i-0.5)/N`, gdje N odgovara broju elemenata, a redni broj `i` počinje od jedan; traženu interpoliranu vrijednost `v(p)` za zadani percentil `p` određujemo izrazom `v(p) = v[i] + N * (p-p(v[i]))*(v[i+1]-v[i])/100`; za percentile koji su manji od p(v[1]) odnosno veći od p(v[N]) vraćamo `v[1]` odnosno `v[N]`; primjerice, 80. percentil niza (1,10,50) bi u tom slučaju bio 46 (detaljnije).

Komponenta `DistributionTester` mora biti oblikovana na način da omogućava uključivanje drugih načina stvaranja cijelih brojeva i računanja percentila, i to bez potrebe za mijenjanjem same komponente.

Oblikujte rješenje problema u skladu s oblikovnim obrascem Strategija, i demonstrirajte funkcionalnost rješenja prikladnim ispitnim programom. Ispitni program treba stvoriti primjerak razreda `DistributionTester`, prikladno ga konfigurirati, te pokrenuti obradu koja rezultira ispisom percentila distribucije.

5. Potrebno je ostvariti programsko rješenje sa sljedećim komponentama.

`SlijedBrojeva` je komponenta koja interno pohranjuje kolekciju cijelih brojeva. Pri stvaranju te komponente, kolekcija je prazna. Komponentu treba oblikovati na način da elemente dobiva od nekog izvora brojeva. U sustavu trebaju postojati različite implementacije izvora brojeva: `TipkovnickiIzvor` koji od korisnika učitava broj po broj s tipkovnice te `DatotecniIzvor` koji brojeve čita iz datoteke. Neka izvori svoje iscrpljivanje signaliziraju vraćanjem vrijednosti -1 (ili na neki drugi prikladan način). U svim ostalim slučajevima očekuje se da izvori uvijek generiraju nenegativne brojeve. Komponenta `SlijedBrojeva` treba biti oblikovana na način da je prilikom njezinog stvaranja moguće umetnuti odgovarajuć izvor brojeva. Rješenje također treba biti takvo da omogućava transparentno dodavanje novih izvora bez promjene koda komponente `SlijedBrojeva`. Razmislite o kojem se tu oblikovnom obrascu radi i implementirajte rješenje u skladu s njime. Komponenta `SlijedBrojeva` treba započeti preuzimanje brojeva od podešenog izvora kada se pozove metoda `kreni` koja potom svake sekunde od izvora pokuša preuzeti po jedan broj. Ako izvoru treba više vremena za generiranje broja, preuzimanje sljedećeg broja potrebno je obaviti jednu sekundu nakon završetka prethodnog čitanja, ma koliko ono trajalo. Programsko rješenje treba napisati na način da je prilikom svake promjene interne kolekcije komponente `SlijedBrojeva` moguće obaviti **jednu ili više** akcija. Akcije koje treba podržati su sljedeće:

1. u tekstovnu datoteku zapisati sve elemente koji se trenutno nalaze u kolekciji te datum i vrijeme zapisa;
2. temeljem elemenata koji se trenutno nalaze u kolekciji potrebno je na zaslon ispisati sumu svih elemenata;
3. temeljem elemenata koji se trenutno nalaze u kolekciji potrebno je na zaslon ispisati prosjek svih elemenata;
4. temeljem elemenata koji se trenutno nalaze u kolekciji potrebno je na zaslon ispisati medijan svih elemenata.

Rješenje treba biti takvo da omogućava konfiguriranje akcija koje treba poduzeti te transparentno dodavanje novih akcija bez potrebe za mijenjanjem komponente `SlijedBrojeva` (primjerice, stupčasti grafički prikaz i slično). Razmislite koji je oblikovni obrazac prikladan za rješavanje ovog problema i implementirajte rješenje u skladu s tim oblikovnim obrascem.

6. Neka razred za enkapsulaciju baze podataka ima sljedeće sučelje:

```
class MyDBase{
public:
  struct Param{
    std::string table;
    std::string column;
    std::string key;
  };
  virtual int query(const Param& p)=0;
};
```

Metoda `query` vraća nulu u slučaju uspjeha, a inače negativan kod greške. Implementiraj dekoratore kojima se i) svaki upit zapisuje u datoteku `/var/tmp/mybase.log`, te ii) kojim se u slučaju da upit ne uspije baca iznimka. Razvij ispitno okruženje, pokaži mogućnost dinamičkog dodavanja i oduzimanja dekoratora i provjeri ispravnost koda.

Izrađeno vi-jem i geditom. Posljednja promjena: Monday, 13-Apr-2015 20:16:40 CEST

Svi komentari su dobrodošli: sinisa.segvic@fer.hr Povratak