**MINISTERUL EDUCAȚIEI ȘI CERCETĂRII AL REPUBLICII MOLDOVA**

**Universitatea Tehnică a Moldovei**

**RAPORT**

Lucrare de laborator nr. 3

la cursul ***„Programarea orientată pe obiecte”***

***Tema: Supraîncărcarea operatorilor***

**Varianta 21**

**A efectuat :**   **St. gr. CR-221FR Serba Cristina**

**A verificat: asist. univ. Mantaluța Marius**

**Chișinău 2024**

# Sarcina lucrării:

а) Să se creeze o ierarhie a claselor om şi colaborator, care ocupă un post anumit şi primeşte un

salariu anumit. Să se supraîncarce operatorii pentru ieşiri şi intrări de obiecte, constructorul de

copiere, operatorii de atribuire utilizînd funcţiile respective ale clasei de bază.

b) Să se creeze clasa carte de joc, care conţine grad şi culoare. Cartea poate fi întoarsă şi

deschisă. Creaţi clasa – butuc de cărţi, care conţine cărţi de joc. Creaţi două clase derivate de la

butuc de cărţi, în una cărţile pot fi scoase numai într-o ordine, iar în alta – aleator.

# Mersul programului:

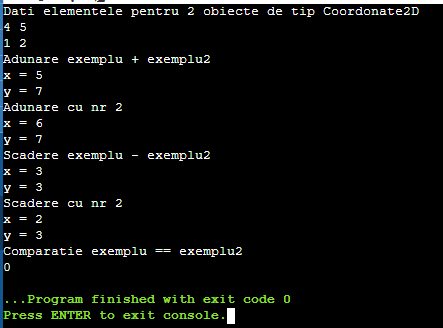
a)

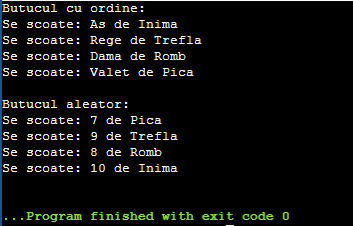
|  |
| --- |
| #include <iostream>  #include <string>  using namespace std;  class Om {  protected:  string nume;  string prenume;  public:  Om(const string& numeParam, const string& prenumeParam)  {  nume = numeParam;  prenume = prenumeParam;  }  // Supraincarcare operator pentru iesire  friend ostream& operator<<(ostream& out, const Om& om) {  out << "Nume: " << om.nume << ", Prenume: " << om.prenume;  return out;  }  // Supraincarcare operator pentru intrare  friend istream& operator>>(istream& in, Om& om) {  cout << "Introduceti numele: ";  in >> om.nume;  cout << "Introduceti prenumele: ";  in >> om.prenume;  return in;  }  // Constructor de copiere  Om(const Om& other)  {  nume = other.nume;  prenume = other.prenume;  }  // Operator de atribuire  Om& operator=(const Om& other) {  if (this != &other) {  nume = other.nume;  prenume = other.prenume;  }  return \*this;  }  virtual ~Om() {}  };  class Colaborator : public Om {  protected:  string post;  double salariu;  public:  Colaborator(const string& numeParam, const string& prenumeParam, const string& postParam, double salariuParam): Om(numeParam, prenumeParam)  {  nume = numeParam;  prenume = prenumeParam;  post = postParam;  salariu = salariuParam;  }  // Supraincarcare operator pentru iesire  friend ostream& operator<<(ostream& out, const Colaborator& colaborator) {  out << "Nume: " << colaborator.nume << ", Prenume: " << colaborator.prenume << ", Post: " << colaborator.post << ", Salariu: " << colaborator.salariu;  return out;  }  // Supraincarcare operator pentru intrare  friend istream& operator>>(istream& in, Colaborator& colaborator) {  cout << "Introduceti numele: ";  in >> colaborator.nume;  cout << "Introduceti prenumele: ";  in >> colaborator.prenume;  cout << "Introduceti postul: ";  in >> colaborator.post;  cout << "Introduceti salariul: ";  in >> colaborator.salariu;  return in;  }  // Constructor de copiere  Colaborator(const Colaborator& other) : Om(other.nume, other.prenume)  {  nume = other.nume;  prenume = other.prenume;  post = other.post;  salariu = other.salariu;  }  // Operator de atribuire  Colaborator& operator=(const Colaborator& other) {  if (this != &other) {  nume = other.nume;  prenume = other.prenume;  post = other.post;  salariu = other.salariu;  }  return \*this;  }  ~Colaborator() {}  };  int main() {  Colaborator colaborator("Moraru", "Mihai", "Manager", 5000);  cout << "Constructorul cu parametri: " << endl;  cout << colaborator << endl;  Colaborator colaborator2 = colaborator;  cout << "Constructorul de copiere: " << endl;  cout << colaborator2 << endl;  Colaborator colaborator3("Ciobanu", "Petru", "Director", 7000);  cout << "Obiectul initial: " << endl;  cout << colaborator3 << endl;  cout << "Obiectul dupa atribuire: " << endl;  colaborator3 = colaborator;  cout << colaborator3 << endl;  return 0;  } |

b)

|  |
| --- |
| #include <iostream>  #include <vector>  #include <string>  #include <cstdlib>  #include <ctime>  using namespace std;  // Clasa pentru o carte de joc  class CarteDeJoc  {  protected:  string grad;  string culoare;  bool esteDeschisa;  public:  CarteDeJoc()  {  grad = "";  culoare = "";  esteDeschisa = false;  }  CarteDeJoc(const string& gradParam, const string& culoareParam)  {  grad = gradParam;  culoare = culoareParam;  esteDeschisa = false;  }  // Metoda pentru a deschide o carte  void deschide()  {  esteDeschisa = true;  }  // Metoda pentru a intoarce o carte  void intoarce()  {  esteDeschisa = false;  }  // Verifica daca cartea este deschisa  bool esteCarteaDeschisa() const  {  return esteDeschisa;  }  // Supraincarcarea operator afisare  friend ostream& operator <<(ostream& out, const CarteDeJoc& carte)  {  out << carte.grad << " de " << carte.culoare;  return out;  }    // Supraincarcarea operator atribuire  void operator =(const CarteDeJoc &carte)  {  grad = carte.grad;  culoare = carte.culoare;  esteDeschisa = carte.esteDeschisa;  }  };  // Clasa pentru butuc de cărți  class ButucDeCarti {  protected:  vector<CarteDeJoc> carti;  public:  // Adauga o carte in butuc  void adaugaCarte(const CarteDeJoc& carte)  {  carti.push\_back(carte);  }  // Metoda pentru a scoate ultima carte din butuc  virtual CarteDeJoc scoateCarte()  {  CarteDeJoc ultimaCarte;  if(esteGol() == false)  {  ultimaCarte = carti.back();  carti.pop\_back();  }  return ultimaCarte;  }  // Verifica daca butucul este gol  bool esteGol()  {  return carti.empty();  }  };  // Clasa derivata pentru butucul de carti in care cartile trebuie scoase intr-o ordine specifica  class ButucOrdine : public ButucDeCarti  {  private:  int index;  public:  ButucOrdine()  {  index = 0;  }  // Supraincarcarea metodei de scoatere a cartii pentru a fi in ordine  CarteDeJoc scoateCarte() override  {  CarteDeJoc carteOrdine;  if (true == carti.empty())  {  cout << "Butucul de carti este gol!" << endl;  }  else  {  carteOrdine = carti[index];  carti.erase(carti.begin() + index);  if (index > 0)  {  // decrementeaza pana la 0  index--;  }  }  return carteOrdine;  }  };  // Clasa derivata pentru butucul de carti aleatorii  class ButucAleator : public ButucDeCarti  {  public:  // Supraincarcarea metodei de scoatere a cartii pentru a fi aleatorie  CarteDeJoc scoateCarte() override  {  CarteDeJoc carteAleatorie;  if (carti.empty())  {  cout << "Butucul de carti este gol!" << endl;  }  else  {  srand(time(nullptr)); // Initializeaza generatorul de numere aleatoare  int index = rand() % carti.size(); // Genereaza un index aleator  carteAleatorie = carti[index];  carti.erase(carti.begin() + index); // Sterge cartea scoasa din butuc  }  return carteAleatorie;  }  };  int main()  {  // Exemplu de utilizare a claselor pentru butucurile de carti  ButucOrdine butucOrdine;  butucOrdine.adaugaCarte(CarteDeJoc("As", "Inima"));  butucOrdine.adaugaCarte(CarteDeJoc("Rege", "Trefla"));  butucOrdine.adaugaCarte(CarteDeJoc("Dama", "Romb"));  butucOrdine.adaugaCarte(CarteDeJoc("Valet", "Pica"));  cout << "Butucul cu ordine:\n";  while (false == butucOrdine.esteGol())  {  cout << "Se scoate: " << butucOrdine.scoateCarte() << endl;  }  ButucAleator butucAleator;  butucAleator.adaugaCarte(CarteDeJoc("10", "Inima"));  butucAleator.adaugaCarte(CarteDeJoc("9", "Trefla"));  butucAleator.adaugaCarte(CarteDeJoc("8", "Romb"));  butucAleator.adaugaCarte(CarteDeJoc("7", "Pica"));  cout << "\nButucul aleator:\n";  while (false == butucAleator.esteGol())  {  cout << "Se scoate: " << butucAleator.scoateCarte() << endl;  }  return 0;  } |

# Rezultatele obținute:

a) 

b)

# Concluzii:

Moștenirea și compoziția sunt două concepte fundamentale în programarea orientată pe obiecte, care permit dezvoltatorilor să creeze ierarhii de clase și să reutilizeze codul într-un mod eficient. Scopul acestei lucrări a fost de a studia în profunzime aceste două concepte, evidențiind avantajele și dezavantajele fiecăruia, precum și regulile și principiile care le guvernează.

În cadrul lucrării, am analizat moștenirea ca un mecanism prin care o clasă poate să-și extindă funcționalitatea prin preluarea caracteristicilor și comportamentului unei alte clase de bază. Avantajele moștenirii includ reutilizarea codului, extensibilitatea și organizarea logică a codului. Cu toate acestea, trebuie să fim conștienți de dezavantajele moștenirii, cum ar fi complexitatea excesivă, rigiditatea și dependența de structurile de clasă.

În ceea ce privește compoziția, am explorat ideea de a construi obiecte complexe prin combinarea altor obiecte mai simple. Compunerea oferă o flexibilitate mai mare decât moștenirea, deoarece permite modificarea comportamentului la runtime și evită problema ambiguității de la care poate suferi moștenirea.

În concluzie, moștenirea și compoziția sunt două tehnici esențiale în programarea orientată pe obiecte, fiecare având avantaje și dezavantaje specifice. Este important ca dezvoltatorii să înțeleagă în mod clar aceste concepte și să le folosească în mod corespunzător pentru a construi aplicații robuste și ușor de întreținut.