Explicație pentru clasa Asistent

Această clasă reprezintă un tip specific de angajat al magazinului, având funcția de asistent. Este derivată din clasa de bază Employee și implementează comportamente specifice acestui rol.

---

1. Scopul Clasei:

Clasa Asistent definește un angajat cu rol de asistent, incluzând:

- Calculul salariului specific (coeficient redus la 0.75 față de baza salarială).
- Identificarea rolului în cadrul organizației.

---

2. Membrii și metodele clasei:

- **Constructori:**
    - Asistent() - Constructor implicit care initializează un obiect gol. Este util pentru crearea temporară a unui obiect fără a furniza date concrete.
    - Asistent(int id, string firstName, string lastName, string CNP, string hireDate) - Constructor parametrizat care inițializează un asistent cu toate datele sale relevante. Apelarea constructorului de bază Employee asigură validarea datelor la crearea obiectului.
- **Metode principale:**
    - float calculateSalary(int years) const override:
        - Suprascrie metoda virtuală din Employee.
        - Calculează salariul în funcție de vechime (years), aplicând coeficientul specific de 0.75 pentru rolul de asistent.
        - Dacă luna curentă coincide cu luna nașterii angajatului (isBirthdayThisMonth()), adaugă un bonus de 100 RON.
        - **Legătură cu cerințe:** Respectă calculul salarial descris în cerințe, inclusiv bonusul pentru ziua de naștere.
    - string getRole() const:

- Returnează rolul specific acestui angajat, în acest caz "Asistent".
- Este util pentru afișarea datelor despre angajat sau pentru raportare.

---

3. Unde îndeplinește cerințele proiectului:

- **Cerințe privind angajații:**
  - Asistenții primesc un salariu de bază ajustat cu un coeficient de 0.75.
  - Bonusul de 100 RON pentru luna de naștere este implementat prin isBirthdayThisMonth().
- **Gestionare:**
  - Asistent este o specializare a clasei generale Employee, ceea ce asigură o arhitectură extensibilă și coerentă pentru gestiunea angajaților.

---

4. Posibile întrebări din partea profesoarei:

1. **De ce ai folosit o clasă derivată?**
   - Pentru a separa logica specifică fiecărui tip de angajat (Manager, Operator, Asistent) într-o manieră modulară, simplificând gestionarea.
2. **Unde este implementat coeficientul de salariu?**
   - În metoda calculateSalary, unde coeficientul 0.75 este aplicat după calcularea salariului de bază.
3. **Cum tratezi ziua de naștere?**
   - Prin apelarea funcției isBirthdayThisMonth(), moștenită din Employee, pentru a verifica dacă luna curentă coincide cu luna nașterii angajatului.

Explicație pentru clasa Clothing

Această clasă reprezintă articolele vestimentare vândute în magazin și derivă din clasa de bază Product. Este utilizată pentru a defini și gestiona produsele de tip îmbrăcăminte, având atribute și metode specifice.

---

1. Scopul Clasei:

Clasa Clothing definește caracteristicile și comportamentul articolelor vestimentare:

- Atribute suplimentare: culoare și marcă.
- Calcularea prețului final cu livrare.
- Afișarea detaliilor produsului.

---

2. Membrii și metodele clasei:

- **Atribute private:**
    - string color: Culoarea articolului vestimentar.
    - string brand: Marca articolului.
- **Constructori:**
    - Clothing() - Constructor implicit care inițializează un articol fără valori concrete. Este util pentru crearea temporară a obiectelor.
    - Clothing(string name, int stock, float basePrice, string uniqueCode, string color, string brand) - Constructor parametrizat care inițializează toate atributele clasei, inclusiv cele moștenite din Product.
- **Metode principale:**
    - float calculateDeliveryPrice() const override:
        - Suprascrie metoda virtuală din Product.
        - Calculează prețul total al produsului adăugând 20 RON (cost de împachetare și livrare) la prețul de bază.
        - **Legătură cu cerințe:** Respectă regula specificată în cerințe pentru articolele vestimentare.

- string getColor() const și string getBrand() const:
    - Getter pentru culoare și marcă. Sunt utilizate pentru a accesa valorile atributelor private.
- string getType() const:
    - Returnează tipul produsului, în acest caz "Clothing". Este util pentru identificarea categoriilor de produse.
- void displayProductDetails() const override:
    - Suprascrie metoda virtuală din Product.
    - Afișează detaliile produsului, inclusiv prețurile, stocul, culoarea și marca.

---

3. Unde îndeplinește cerințele proiectului:

- **Cerințe privind produse:**
    - Fiecare articol vestimentar are un cod unic, un nume, un stoc și un preț de bază, gestionate de clasa de bază Product.
    - Prețul de livrare este calculat conform specificațiilor (20 RON adăugați la prețul de bază).
    - Detaliile fiecărui produs sunt afișate corect cu metoda displayProductDetails.
- **Gestionare stoc:**
    - Este ușor de integrat în sistemul de stocuri datorită moștenirii din Product.

---

4. Posibile întrebări din partea profesoarei:

1. **De ce ai derivat Clothing din Product?**
    - Pentru a reutiliza logica comună pentru toate produsele (name, uniqueCode, basePrice, etc.) și a extinde funcționalitatea cu elemente specifice articolelor vestimentare.
2. **Unde se calculează prețul cu livrare?**

- În metoda calculateDeliveryPrice, care adaugă costul de 20 RON conform cerințelor.

3. **Cum afișezi detaliile produsului?**
    - Cu metoda displayProductDetails, care include toate informațiile relevante, inclusiv prețul final.
4. **Cum identifici tipul produsului?**
    - Cu metoda getType, care returnează stringul "Clothing", util în categorisirea produselor.

Explicație pentru clasa Disc

Această clasă reprezintă produsele de tip disc (CD-uri sau viniluri) disponibile în magazin și derivă din clasa de bază Product. Este folosită pentru a adăuga caracteristici și comportamente specifice acestui tip de produs.

---

1. Scopul Clasei:

Clasa Disc definește și gestionează datele specifice discurilor muzicale:

- Atribute suplimentare: casa de discuri, data lansării, trupa și numele albumului.
- Calcularea prețului final cu livrare.
- Afișarea completă a detaliilor despre produs.

---

2. Membrii și metodele clasei:

- **Atribute private:**
    - string recordLabel: Casa de discuri care a lansat albumul.
    - string releaseDate: Data lansării discului.
    - string band: Trupa sau artistul.
    - string albumName: Numele albumului.
- **Constructori:**

- Disc() - Constructor implicit care inițializează un disc cu valori goale pentru toate atributele.
- Disc(string name, int stock, float basePrice, string uniqueCode, string recordLabel, string releaseDate, string band, string albumName):
  - Constructor parametrizat care inițializează toate atributele clasei, incluzând cele moștenite din Product.

- **Metode principale:**
  - float calculateDeliveryPrice() const override:
    - Suprascrie metoda virtuală din Product.
    - Calculează prețul total al produsului adăugând 5 RON (cost de livrare) la prețul de bază.
    - **Legătură cu cerințe:** Respectă regula specificată în cerințe pentru calculul prețului de livrare al discurilor.
  - **Getter Methods**:
    - string getRecordLabel() const: Returnează casa de discuri.
    - string getReleaseDate() const: Returnează data lansării.
    - string getBand() const: Returnează trupa sau artistul.
    - string getAlbumName() const: Returnează numele albumului.
  - string getType() const override:
    - Suprascrie metoda din Product și returnează "Disc" pentru identificarea categoriei produsului.
  - void displayProductDetails() const override:
    - Suprascrie metoda virtuală din Product.
    - Afișează detaliile complete ale discului, inclusiv informațiile despre trupa, album și prețurile (cu și fără livrare).

---

3. Unde îndeplinește cerințele proiectului:

- **Cerințe privind produse:**

- Discurile sunt reprezentate ca produse cu un set unic de caracteristici (record label, release date, etc.).
- Prețul de livrare este calculat conform cerințelor (5 RON adăugați la prețul de bază).
- Detaliile fiecărui produs sunt afișate corect cu metoda displayProductDetails.

- **Gestionare stoc:**
  - Este compatibilă cu gestionarea stocurilor datorită moștenirii din Product, permițând adăugare, ștergere, modificare și afișare.

---

4. Posibile întrebări din partea profesoarei:

1. **De ce ai derivat Disc din Product?**
   - Pentru a utiliza logica comună a clasei de bază (atribute și metode) și a adăuga specificații suplimentare pentru discuri.
2. **Cum calculezi prețul final pentru discuri?**
   - În metoda calculateDeliveryPrice, unde se adaugă costul standard de livrare (5 RON) la prețul de bază.
3. **Cum afișezi detaliile discului?**
   - Cu metoda displayProductDetails, care listează informațiile despre stoc, prețuri, trupa, albumul, casa de discuri și data lansării.
4. **Cum identifici tipul produsului?**
   - Cu metoda getType, care returnează "Disc" pentru a distinge acest produs de alte categorii.

Explicație pentru clasa Employee

Clasa Employee este clasa de bază pentru reprezentarea tuturor angajaților din cadrul magazinului. Ea oferă structura generală și funcționalități comune pentru toate tipurile de angajați (manageri, operatori, asistenți).

---

1. Scopul Clasei:

Clasa este concepută pentru a oferi:

- Atribute și metode comune pentru gestionarea datelor angajaților.
- Funcționalități de validare pentru date esențiale precum CNP și data angajării.
- O bază flexibilă pentru extinderea prin clase derivate care implementează roluri specifice.

---

2. Membrii și metodele clasei:

Atribute Protejate:

- int id: ID unic pentru fiecare angajat.
- string firstName și string lastName: Numele și prenumele angajatului.
- string CNP: Cod numeric personal, validat la inițializare.
- string hireDate: Data angajării (format validat).
- float baseSalary: Salariul de bază, comun tuturor angajaților (3.500 RON).

Constructori:

- **Constructor implicit**:
    - Inițializează un angajat gol, util pentru instanțieri temporare sau teste.
- **Constructor parametrizat**:
    - Inițializează toate datele angajatului. Include validări pentru CNP și hireDate.

Metode principale:

- float calculateSalary(int years) const:
    - Calculează salariul de bază în funcție de vechime.
    - Este virtuală, permițând personalizarea calculului în clasele derivate.
- void display() const:
    - Afișează detalii despre angajat, inclusiv salariul calculat.
- bool isBirthdayThisMonth() const:

    - Verifică dacă luna nașterii (extrasă din CNP) coincide cu luna curentă.
    - **Legătură cu cerințe:** Este utilizată în clasele derivate pentru a adăuga bonusuri salariale.
- static bool validateCNP(const string& CNP):
    - Verifică dacă CNP-ul are 13 cifre și este valid.
    - **Legătură cu cerințe:** Respectă cerințele pentru validarea CNP-urilor.
- static bool isValidDate(const string& date):
    - Validează formatul și consistența unei date (DD-MM-YYYY).
- int getYearsOfExperience() const:
    - Calculează vechimea angajatului bazându-se pe anul curent și data angajării.

Setteri și Getteri:

- Setteri pentru modificarea numelui (setFirstName, setLastName) – necesari pentru gestionarea schimbărilor (ex. căsătorie).
- Getteri pentru accesarea atributelor esențiale (getFirstName, getLastName, getCNP, getId, getHireDate etc.).

Metodă virtuală pură:

- virtual string getRole() const = 0:
    - Forțează implementarea rolului specific în clasele derivate (ex. Asistent, Manager).

---

3. Unde îndeplinește cerințele proiectului:

- **Cerințe generale pentru angajați:**
    - Toți angajații au ID, nume, prenume, CNP, data angajării și salariu calculat conform vechimii.
    - Validarea CNP și a datei angajării asigură respectarea regulilor.
    - Calculul salariului include creșteri pe baza anilor de experiență.
- **Flexibilitate pentru extensii:**

- Clasele derivate (ex. Asistent, Operator) pot personaliza metodele calculateSalary și getRole.

---

4. Posibile întrebări din partea profesoarei:

1. **Cum validezi CNP-ul?**
    - Metoda statică validateCNP verifică dacă lungimea este 13 și toate caracterele sunt cifre. În caz de invaliditate, constructorul aruncă o excepție.
2. **Cum gestionezi ziua de naștere?**
    - Metoda isBirthdayThisMonth compară luna curentă cu luna extrasă din CNP (pozițiile 3-4).
3. **Cum calculezi salariul angajaților?**
    - Metoda calculateSalary adaugă 100 RON pentru fiecare an de vechime și este adaptată în clasele derivate pentru coeficienți sau bonusuri suplimentare.
4. **Cum te asiguri că fiecare angajat are un rol specific?**
    - Clasa definește metoda pură virtuală getRole, forțând clasele derivate să specifice rolul.
5. **Cum gestionezi modificarea datelor personale?**
    - Cu setterii setFirstName și setLastName, permițând schimbarea numelui după căsătorie.

Explicație pentru clasa EmployeeManager

Clasa EmployeeManager este responsabilă de gestionarea tuturor angajaților din cadrul magazinului. Aceasta oferă funcționalități esențiale pentru adăugarea, ștergerea, modificarea, validarea și raportarea informațiilor despre angajați.

---

1. Scopul Clasei:

Clasa centralizează gestionarea angajaților și asigură respectarea cerințelor organizaționale:

- Gestionarea completă a listei de angajați folosind un container din STL (vector).
- Validarea structurii de personal (manageri, operatori, asistenți).
- Generarea rapoartelor despre angajați.

---

2. Membrii și metodele clasei:

Atribut Privat:

- vector<shared_ptr<Employee>> employees:
    - Conține lista tuturor angajaților.
    - Utilizarea shared_ptr permite gestionarea memoriei într-un mod sigur, având în vedere că obiectele angajaților pot fi accesate și din alte părți ale programului.

Constructori:

1. **Constructor implicit:** Creează un manager de angajați gol.
2. **Constructor parametrizat:** Permite inițializarea cu o listă de angajați deja existentă.

Metode Publice:

1. **Adăugare angajat:** void addEmployee(shared_ptr<Employee> employee)
    - Adaugă un nou angajat în lista de angajați, verificând:
        - Dacă ID-ul angajatului este unic.
        - Dacă angajatul are cel puțin 18 ani (verificare bazată pe CNP).
    - Aruncă excepții în caz de erori (ID duplicat sau vârstă sub 18 ani).
2. **Ștergere angajat:** void removeEmployee(int id)
    - Șterge angajatul cu un anumit ID din lista de angajați folosind std::remove_if.
3. **Actualizare informații:** void updateEmployeeInfo(int id, const string& newName)

- Permite modificarea numelui angajatului identificat după ID.

4. **Accesare angajat:**
    - shared_ptr<Employee> getEmployee(int id) const: Returnează un angajat pe baza ID-ului.
    - vector<shared_ptr<Employee>> getAllEmployees() const: Returnează lista tuturor angajaților.
5. **Validare structura personalului:** bool checkOperationalStatus() const
    - Verifică dacă magazinul are structura minimă necesară:
        - Cel puțin 1 manager.
        - Cel puțin 3 operatori.
        - Cel puțin 1 asistent.
    - Returnează false și afișează un mesaj dacă cerințele nu sunt îndeplinite.
6. **Generare rapoarte:**
    - Utilizează clase auxiliare (Reports) pentru a genera fișiere de raport:
        - Angajatul cu cele mai multe comenzi procesate.
        - Top 3 angajați cu cele mai valoroase comenzi.
        - Top 3 angajați cu cele mai mari salarii.

---

3. Unde îndeplinește cerințele proiectului:

- **Gestiunea angajaților:**
    - Permite adăugarea, ștergerea, modificarea și afișarea angajaților.
    - Validarea structurilor minime de personal este implementată în metoda checkOperationalStatus.
- **Rapoarte:**
    - Generarea rapoartelor pentru comenzi și salarii este realizată prin metodele reportMostOrdersProcessed, reportTop3MostValuableOrders și reportTop3HighestSalaries.

- **Validarea datelor angajaților:**
    - Adăugarea unui angajat verifică unicitatea ID-ului și validitatea vârstei (18+ ani).

---

4. Posibile întrebări din partea profesoarei:

1. **Cum te asiguri că ID-urile angajaților sunt unice?**
    - În metoda addEmployee, folosesc getEmployee pentru a verifica dacă ID-ul există deja. Dacă ID-ul este duplicat, arunc o excepție.
2. **Cum validezi vârsta unui angajat?**
    - În metoda addEmployee, extrag anul nașterii din CNP și îl compar cu anul curent. Dacă vârsta este sub 18 ani, arunc o excepție.
3. **Cum verifici structura minimă a personalului?**
    - Folosesc dynamic_pointer_cast pentru a număra angajații din fiecare categorie (manager, operator, asistent) în metoda checkOperationalStatus.
4. **Cum gestionezi memoria pentru angajați?**
    - Utilizez shared_ptr pentru a evita scurgerile de memorie, iar vector facilitează gestionarea listei de angajați.
5. **Cum sunt generate rapoartele?**
    - Metodele de raport folosesc clasa auxiliară Reports, ceea ce asigură separarea logicii de generare a rapoartelor de gestionarea angajaților.

Explicație pentru clasa InventoryManager

Clasa InventoryManager gestionează stocul de produse din magazin. Aceasta centralizează funcționalitățile necesare pentru adăugarea, ștergerea, modificarea și afișarea produselor, precum și verificarea validității stocului.

---

1. Scopul Clasei:

Clasa asigură:

- Organizarea și manipularea produselor stocate într-un container STL (vector).
- Validarea stocului pentru a respecta cerințele de funcționare ale magazinului.
- Interacțiunea cu produsele prin metode simple și eficiente.

---

2. Membrii și metodele clasei:

Atribut Privat:

- vector<shared_ptr<Product>> products:
    - Container pentru stocarea produselor.
    - Utilizarea shared_ptr permite o gestionare sigură a memoriei, mai ales în cazurile în care produsele pot fi partajate între diferite componente ale sistemului.

Constructori:

1. **Constructor implicit:** Inițializează un manager de inventar gol.
2. **Constructor parametrizat:** Permite inițializarea cu o listă de produse existente.

Metode Publice:

1. **Adăugare produs:** void addProduct(const shared_ptr<Product>& product)
    - Adaugă un produs în lista de produse.
2. **Ștergere produs:** void removeProduct(const string& uniqueCode)
    - Șterge produsul cu cod unic specificat folosind std::remove_if.
3. **Actualizare stoc:** void updateStock(const string& uniqueCode, int newStock)
    - Actualizează stocul pentru un produs identificat prin cod unic.
4. **Accesare produs:**
    - shared_ptr<Product> getProduct(const string& uniqueCode) const: Returnează produsul cu codul unic specificat sau nullptr dacă nu există.

- vector<shared_ptr<Product>> getAllProducts() const: Returnează lista tuturor produselor.

5. **Validare stoc:** bool checkStockValidity() const
   - Verifică dacă stocul conține cel puțin 2 produse din fiecare categorie:
     - Articole vestimentare (Clothing).
     - Discuri (Disc).
     - Discuri vintage (VintageDisc).
   - Afișează un mesaj detaliat despre situația actuală a stocului în caz de nevaliditate.
6. **Verificare existență produs:** bool productExists(const string& uniqueCode) const
   - Verifică dacă există un produs cu un anumit cod unic.
7. **Afișare produse:** void displayAllProducts() const
   - Afișează detaliile fiecărui produs folosind metoda virtuală displayProductDetails din clasa Product.

---

3. Unde îndeplinește cerințele proiectului:

- **Gestiunea produselor:**
  - Permite adăugarea, ștergerea, modificarea și afișarea produselor din stoc.
  - Asigură integrarea ușoară a tuturor tipurilor de produse (Clothing, Disc, VintageDisc) datorită utilizării polimorfismului (shared_ptr<Product>).
- **Validare stoc:**
  - Verificarea stocului minim necesar este realizată prin metoda checkStockValidity, conform cerințelor:
    - Cel puțin 2 produse din fiecare tip trebuie să fie în stoc pentru ca magazinul să funcționeze.
- **Interacțiune clară și detaliată:**
  - Mesajele afișate în cazul validării stocului sau afișării produselor sunt clare și ușor de înțeles.

---

4. Posibile întrebări din partea profesoarei:

1. **Cum gestionezi memoria pentru produse?**
    - Folosesc shared_ptr pentru a evita scurgerile de memorie și pentru a partaja obiectele între diferite componente.
2. **Cum verifici validitatea stocului?**
    - Prin metoda checkStockValidity, care folosește un unordered_map pentru a număra produsele din fiecare categorie (getType).
3. **Cum te asiguri că produsele pot fi șterse sau actualizate corect?**
    - Utilizând algoritmi STL precum std::remove_if și std::find_if pentru identificarea produselor în funcție de codul unic.
4. **Cum gestionezi afișarea produselor?**
    - Folosesc metoda virtuală displayProductDetails din Product, permițând fiecărui tip de produs să afișeze informațiile specifice într-un mod polimorf.
5. **De ce folosești polimorfism pentru produse?**
    - Pentru a asigura extensibilitatea. Pot adăuga ușor noi tipuri de produse fără a modifica logica din InventoryManager.

Explicație pentru clasa Manager

Clasa Manager este o specializare a clasei de bază Employee, reprezentând rolul de manager în cadrul magazinului. Această clasă implementează comportamentul specific acestui tip de angajat, cum ar fi calculul salariului și identificarea rolului.

---

1. Scopul Clasei:

Clasa definește caracteristicile și funcționalitățile unice ale unui manager:

- Calcularea salariului, care include un coeficient specific de 1.25.
- Identificarea rolului în cadrul organizației.

---

2. Membrii și metodele clasei:

Constructori:

1. **Constructor implicit:**
    - Creează un obiect Manager cu atributele implicite ale clasei Employee.
2. **Constructor parametrizat:**
    - Inițializează un Manager cu valorile furnizate pentru ID, nume, prenume, CNP și data angajării. Folosește constructorul clasei de bază pentru a valida și seta aceste atribute.

Metode principale:

1. **Calcularea salariului:** float calculateSalary(int years) const override
    - Suprascrie metoda virtuală din Employee.
    - Calculează salariul managerului astfel:
        - Salariul de bază (3500 RON) este majorat cu 100 RON pentru fiecare an de vechime.
        - La valoarea rezultată se aplică un coeficient de 1.25, conform cerințelor.
    - **Exemplu:**
        - Pentru un manager cu 5 ani de vechime: Salariu=(3500+100×5)×1.25=4375.0 RON.\text{Salariu} = (3500 + 100 \times 5) \times 1.25 = 4375.0 \, \text{RON}.Salariu=(3500+100×5)×1.25=4375.0RON.
2. **Identificarea rolului:** string getRole() const
    - Returnează stringul "Manager", care poate fi utilizat pentru afișarea rolului sau pentru raportare.

---

3. Unde îndeplinește cerințele proiectului:

- **Calculul salariului:**
    - Respectă cerința de aplicare a unui coeficient de 1.25 pentru manageri.
- **Identificarea rolului:**
    - Metoda getRole permite distingerea managerilor de ceilalți angajați.
- **Flexibilitate și extensibilitate:**
    - Moștenirea din Employee oferă acces la funcționalități comune (ex. isBirthdayThisMonth), reducând redundanța codului.

---

4. Posibile întrebări din partea profesoarei:

1. **Cum se calculează salariul pentru un manager?**
    - Salariul este calculat prin formula:
      Salariu=(Salariu de bazaˇ+100×Vechime ıˆn ani)×1.25\text{Salariu} = (\text{Salariu de bază} + 100 \times \text{Vechime în ani}) \times 1.25Salariu=(Salariu de bazaˇ+100×Vechime ıˆn ani)×1.25
        - Este implementată în metoda calculateSalary.
2. **Cum identifici tipul angajatului?**
    - Metoda getRole returnează "Manager", facilitând identificarea în raportări sau afișări.
3. **De ce ai folosit o clasă derivată?**
    - Pentru a separa logicile specifice fiecărui tip de angajat (Manager, Operator, Asistent), păstrând o arhitectură modulară și ușor de extins.
4. **Cum gestionezi modificările salariale sau alte cerințe noi?**
    - Adăugarea sau modificarea regulilor specifice pentru Manager este simplă, deoarece toate logica specifică acestui rol este concentrată în această clasă.

Explicație pentru clasa OperatorComenzi

Clasa OperatorComenzi derivă din clasa de bază Employee și reprezintă operatorii de comenzi din cadrul magazinului. Aceasta implementează funcționalități specifice pentru gestionarea și procesarea comenzilor, precum și pentru calcularea salariului.

---

1. Scopul Clasei:

Clasa este concepută să:

- Gestioneze comenzile atribuite operatorilor.
- Calculeze salariul, incluzând bonusuri bazate pe valoarea comenzilor procesate.
- Permită adăugarea, procesarea și raportarea comenzilor atribuite unui operator.

---

2. Membrii și metodele clasei:

Atribute Private:

1. float salesBonus:
    - Totalul bonusurilor acumulate de operator (0.5% din valoarea comenzilor procesate).
2. vector<shared_ptr<Order>> assignedOrders:
    - Stochează comenzile atribuite operatorului pentru procesare.
3. int processedOrderCount:
    - Numărul total de comenzi procesate de operator.

Constructori:

1. **Constructor implicit:**
    - Creează un obiect OperatorComenzi gol.
2. **Constructor parametrizat:**
    - Inițializează un operator cu ID, nume, CNP și data angajării, apelând constructorul clasei de bază.

Metode principale:

1. **Procesare comandă:** void processOrder(float orderValue)
    - Adaugă la salesBonus 0.5% din valoarea unei comenzi procesate.
2. **Calcularea salariului:** float calculateSalary(int years) const override
    - Calculează salariul total al operatorului:
        - Salariul de bază este ajustat cu 100 RON pentru fiecare an de vechime.
        - Bonusul acumulat din comenzi este adăugat.
        - Dacă este ziua de naștere a operatorului, se adaugă un bonus de 100 RON.
3. **Rolul operatorului:** string getRole() const
    - Returnează stringul "Operator Comenzi".
4. **Gestionarea comenzilor:**
    - void assignOrder(const shared_ptr<Order>& order):
        - Adaugă o comandă în lista comenzilor atribuite operatorului.
    - shared_ptr<Order> popLastOrder():
        - Elimină și returnează ultima comandă din lista de comenzi atribuite.
    - void clearOrders():
        - Golește lista de comenzi atribuite.
    - vector<shared_ptr<Order>> getAssignedOrders() const:
        - Returnează lista tuturor comenzilor atribuite.
    - float getTotalOrderValue() const:
        - Calculează valoarea totală a comenzilor atribuite.
    - int getCurrentOrderCount() const:
        - Returnează numărul de comenzi în procesare.
5. **Numărul comenzilor procesate:** int getProcessedOrderCount() const
    - Returnează numărul total de comenzi procesate de operator.

---

3. Unde îndeplinește cerințele proiectului:

- **Gestionarea comenzilor:**
    - Operatorii pot primi comenzi (assignOrder), pot procesa comenzi (processOrder) și pot calcula valoarea totală a acestora (getTotalOrderValue).
- **Calculul salariului:**
    - Salariul include bonusul de 0.5% din valoarea comenzilor procesate, conform cerințelor.
    - Bonusul pentru ziua de naștere este adăugat dacă este cazul.
- **Monitorizarea încărcării:**
    - Numărul de comenzi atribuite și procesate este gestionat prin getCurrentOrderCount și getProcessedOrderCount.

---

4. Posibile întrebări din partea profesoarei:

1. **Cum calculezi salariul unui operator?**
    - Salariul este calculat ca suma salariului de bază, bonusurilor de 0.5% din valoarea comenzilor procesate și unui bonus suplimentar de 100 RON dacă este ziua de naștere a operatorului.
2. **Cum gestionezi comenzile atribuite unui operator?**
    - Comenzile sunt stocate într-un vector assignedOrders, care permite adăugarea, eliminarea și raportarea comenzilor.
3. **Cum calculezi valoarea totală a comenzilor procesate?**
    - Metoda getTotalOrderValue parcurge vectorul assignedOrders și adună valorile tuturor comenzilor.
4. **De ce folosești shared_ptr pentru comenzi?**
    - Pentru a permite partajarea comenzilor între mai multe componente fără a crea copii redundante sau a risca scurgeri de memorie.
5. **Cum te asiguri că operatorii nu au mai mult de 3 comenzi simultan?**

- Restricția poate fi implementată la nivelul metodei assignOrder, verificând valoarea returnată de getCurrentOrderCount.

Explicație pentru clasa Order

Clasa Order reprezintă comenzile efectuate de către clienți în magazin. Ea gestionează articolele din comandă, validarea comenzii, procesarea acesteia, calcularea prețului total și afișarea detaliilor.

---

1. Scopul Clasei:

Clasa este utilizată pentru:

- Gestionarea articolelor incluse într-o comandă (OrderItem).
- Calcularea prețului total, incluzând costurile de livrare.
- Validarea comenzii conform regulilor definite.
- Asocierea comenzii cu un proces și afișarea detaliilor relevante.

---

2. Membrii și metodele clasei:

Atribute Private:

1. int id:
    - ID-ul unic al comenzii.
    - Este generat automat utilizând atributul static nextId.
2. float totalPrice:
    - Prețul total al comenzii, calculat în funcție de produsele incluse și costurile de livrare.

3. vector<OrderItem> items:
    - Lista de articole incluse în comandă.
4. string orderDate:
    - Data plasării comenzii.
5. bool isProcessed:
    - Indică dacă comanda a fost procesată sau nu.
6. int processingDuration:
    - Durata procesării comenzii.
7. static int nextId:
    - Atribut static folosit pentru generarea ID-urilor unice.

Constructori:

1. **Constructor implicit:**
    - Inițializează comanda cu valori implicite, setând automat ID-ul și data curentă.
2. **Constructor parametrizat:**
    - Permite inițializarea comenzii cu articole, dată și durata procesării.

Metode principale:

1. **Adăugare articol:** void addItem(const OrderItem& item)
    - Adaugă un articol în comandă.
2. **Calculare preț total:** void calculateTotalPrice(const InventoryManager& inventory)
    - Calculează prețul total al comenzii, incluzând costurile de livrare pentru fiecare articol.
    - Accesează detalii despre produse prin intermediul InventoryManager.
3. **Procesare comandă:** bool processOrder(InventoryManager& inventory, vector<shared_ptr<Employee>>& employees)

- Procesează comanda dacă este validă și returnează true sau false.

4. **Validare comandă:** bool validateOrder(const InventoryManager& inventory) const
    - Verifică dacă:
        - Toate produsele există în stoc.
        - Numărul de discuri nu depășește 5.
        - Numărul de articole vestimentare nu depășește 3.
        - Valoarea totală (fără livrare) este de cel puțin 100 RON.
5. **Afișare detalii:** void displayOrderDetails() const
    - Afișează detaliile comenzii: data, prețul total, durata procesării și produsele incluse.
6. **Getteri și setteri:**
    - Permite accesarea și modificarea atributelor precum ID-ul, prețul total, durata procesării și starea procesării (isProcessed).

---

3. Unde îndeplinește cerințele proiectului:

- **Validare comandă:**
    - Metoda validateOrder asigură respectarea regulilor pentru:
        - Numărul maxim de discuri și articole vestimentare.
        - Valoarea minimă a comenzii.
        - Existența produselor în stoc.
- **Calculare preț total:**
    - Metoda calculateTotalPrice adaugă costurile de livrare pentru fiecare produs, conform cerințelor fiecărui tip (Clothing, Disc, VintageDisc).
- **Gestionare comenzi:**
    - Comenzile sunt procesate și pot fi asociate cu angajați (OperatorComenzi).

---

4. Posibile întrebări din partea profesoarei:

1. **Cum validezi o comandă?**
    - Verific prin metoda validateOrder următoarele:
        - Produsele există în stoc.
        - Numărul maxim de articole per tip este respectat.
        - Valoarea minimă a comenzii este de cel puțin 100 RON.
2. **Cum calculezi prețul total al comenzii?**
    - Parcurg fiecare articol, calculez prețul final per produs (inclusiv livrare) și înmulțesc cu cantitatea. Totalul este acumulat în totalPrice.
3. **Cum asociezi comenzile cu angajați?**
    - În afara clasei, comanda poate fi atribuită unui OperatorComenzi utilizând metodele specifice din acea clasă.
4. **Cum gestionezi ID-urile comenzilor?**
    - ID-urile sunt generate automat prin incrementarea atributului static nextId.
5. **Cum gestionezi comenzile procesate?**
    - Atributul isProcessed indică starea unei comenzi. Metoda processOrder modifică această stare dacă validarea este reușită.

Explicație pentru clasa OrderItem

Clasa OrderItem reprezintă un element individual dintr-o comandă. Fiecare obiect al clasei conține informații despre un produs specific și cantitatea acestuia.

---

1. Scopul Clasei:

Clasa este utilizată pentru:

- Reprezentarea unui produs dintr-o comandă.
- Gestionarea codului unic al produsului și a cantității acestuia.

---

2. Membrii și metodele clasei:

Atribute Private:

1. string productCode:
    - Codul unic al produsului inclus în comandă. Acesta este folosit pentru identificarea produsului în inventar.
2. int quantity:
    - Cantitatea de produse incluse în comandă pentru acel cod.

Constructori:

1. **Constructor implicit:**
    - Inițializează un obiect cu un cod gol și o cantitate de 0.
2. **Constructor parametrizat:**
    - Inițializează obiectul cu un cod de produs specific și o cantitate furnizată de utilizator.

Metode Publice:

1. **Getter pentru productCode:** string getProductCode() const
    - Returnează codul produsului din acest element de comandă.
2. **Setter pentru productCode:** void setProductCode(const string& code)
    - Setează un cod unic pentru produs.
3. **Getter pentru quantity:** int getQuantity() const
    - Returnează cantitatea specificată pentru produs.
4. **Setter pentru quantity:** void setQuantity(int qty)
    - Setează cantitatea produsului.

---

3. Unde îndeplinește cerințele proiectului:

- **Gestionarea comenzilor:**

- Fiecare articol dintr-o comandă este reprezentat de un obiect OrderItem.
- Codul unic al produsului (productCode) este utilizat pentru a accesa informațiile despre produs din inventar.
- Cantitatea (quantity) permite calcularea prețului total al comenzii.

- **Simplificare și modularitate:**
  - Separarea informațiilor despre fiecare produs într-o clasă dedicată (OrderItem) permite gestionarea mai ușoară a comenzilor, inclusiv adăugarea, eliminarea sau modificarea articolelor.

---

4. Posibile întrebări din partea profesoarei:

1. **De ce ai creat o clasă separată pentru articolele din comandă?**
   - Pentru a reprezenta în mod clar fiecare produs dintr-o comandă, separând informațiile despre cod și cantitate. Aceasta simplifică gestionarea comenzilor în clasa Order.
2. **Cum accesezi detalii despre un produs dintr-o comandă?**
   - Prin metoda getProductCode, care returnează codul unic al produsului. Acest cod este folosit pentru a căuta produsul în inventar.
3. **Cum modifici cantitatea unui produs într-o comandă?**
   - Prin metoda setQuantity, care permite actualizarea cantității specificate.
4. **Cum tratezi articolele fără cantitate sau cod?**
   - Constructorul implicit setează productCode la un string gol și quantity la 0. Aceste cazuri pot fi filtrate în alte părți ale programului (de exemplu, la validarea comenzii).

Explicație pentru clasa Product

Clasa Product este o clasă abstractă ce definește caracteristicile și comportamentele comune ale tuturor produselor din magazin. Clasele derivate (ex. Clothing, Disc, VintageDisc) implementează detaliile specifice fiecărui tip de produs.

---

1. Scopul Clasei:

Clasa Product oferă:

- O bază pentru reprezentarea produselor, gestionând atribute comune precum nume, stoc, preț și cod unic.
- Declarații pentru metode virtuale pure care trebuie implementate de clasele derivate:
    - Calculul prețului cu livrare.
    - Afișarea detaliilor despre produs.
    - Determinarea tipului de produs.

---

2. Membrii și metodele clasei:

Atribute Protejate:

1. string name:
    - Numele produsului.
2. int stock:
    - Cantitatea disponibilă în stoc.
3. float basePrice:
    - Prețul de bază al produsului (fără costuri de livrare).
4. string uniqueCode:
    - Codul unic care identifică produsul în inventar.

Constructori:

1. **Constructor implicit:**
    - Inițializează produsul cu atribute implicite (nume gol, stoc 0, preț 0.0, cod unic gol).
2. **Constructor parametrizat:**

- Inițializează produsul cu valori specificate pentru nume, stoc, preț și cod unic.

Metode Publice:

1. **Metode virtuale pure:**
   - virtual float calculateDeliveryPrice() const = 0:
     - Clasele derivate implementează calculul prețului final al produsului, incluzând costurile de livrare.
   - virtual void displayProductDetails() const = 0:
     - Clasele derivate implementează afișarea detaliilor produsului.
   - virtual string getType() const = 0:
     - Returnează tipul specific al produsului (ex. "Clothing", "Disc").
2. **Getteri pentru atribute:**
   - string getName() const: Returnează numele produsului.
   - int getStock() const: Returnează cantitatea din stoc.
   - float getBasePrice() const: Returnează prețul de bază al produsului.
   - string getUniqueCode() const: Returnează codul unic al produsului.
3. **Setteri pentru atribute:**
   - void setStock(int newStock): Setează cantitatea în stoc.
   - void setName(string newName): Setează numele produsului.
   - void setBasePrice(float newBasePrice): Setează prețul de bază.
   - void setUniqueCode(string newUniqueCode): Setează codul unic.

---

3. Unde îndeplinește cerințele proiectului:

- **Definirea caracteristicilor comune ale produselor:**
  - Atribute precum name, stock, basePrice și uniqueCode sunt relevante pentru toate produsele din magazin.

- **Extensibilitate:**
    - Metodele virtuale pure permit clasele derivate să implementeze comportamente specifice (ex. calculul costurilor de livrare pentru discuri sau articole vestimentare).
- **Gestionarea produselor:**
    - Getterii și setterii facilitează interacțiunea cu produsele din clasele de gestionare (ex. InventoryManager).

---

4. Posibile întrebări din partea profesoarei:

1. **De ce ai creat o clasă abstractă pentru produse?**
    - Pentru a centraliza atributele și comportamentele comune tuturor produselor, reducând redundanța codului. Clasele derivate implementează doar logica specifică fiecărui tip de produs.
2. **Cum asiguri extensibilitatea pentru noi tipuri de produse?**
    - Prin metodele virtuale pure (calculateDeliveryPrice, displayProductDetails, getType), care trebuie implementate de noile clase derivate.
3. **De ce sunt atributele protejate și nu private?**
    - Atributele protected permit acces direct în clasele derivate, simplificând implementarea fără a utiliza getterii/setterii din clasa de bază.
4. **Cum calculezi prețul final al unui produs?**
    - Clasele derivate implementează metoda calculateDeliveryPrice, aplicând costurile specifice fiecărui tip de produs.
5. **Cum identifici tipul unui produs?**
    - Metoda virtuală getType este implementată de clasele derivate pentru a returna tipul specific (ex. "Clothing", "Disc").

Explicație pentru clasa Reports

Clasa Reports este utilizată pentru a genera rapoarte despre angajații magazinului. Aceasta este o clasă statică (toate metodele sunt statice) și permite exportarea datelor relevante în fișiere CSV pentru analiză sau raportare.

---

1. Scopul Clasei:

Clasa este folosită pentru:

- Crearea de rapoarte detaliate despre performanța angajaților.
- Exportarea datelor în fișiere CSV pentru utilizare ulterioară.
- Generarea automată a topurilor bazate pe criterii precum numărul de comenzi procesate, valoarea comenzilor și salariile angajaților.

---

2. Membrii și metodele clasei:

Metode Publice Statice:

1. **generateReportMostOrdersProcessed:**
    - Generează un raport cu operatorii de comenzi care au procesat cele mai multe comenzi.
    - **Intrări:**
        - EmployeeManager& manager: Referință la managerul de angajați pentru accesarea tuturor angajaților.
        - string filename: Numele fișierului CSV unde se salvează raportul.
    - **Funcționare:**
        - Parcurge lista de angajați, filtrează operatorii de comenzi și determină numărul maxim de comenzi procesate.
        - Scrie în fișier numele, prenumele și numărul de comenzi pentru operatorii care au procesat cel mai mare număr de comenzi.
    - **Ieșire:** Fișier CSV cu operatorii și comenzi procesate.
2. **generateReportTop3MostValuableOrders:**

- Generează un raport cu top 3 operatori care au gestionat comenzile cele mai valoroase.
- **Intrări:**
  - EmployeeManager& manager: Referință la managerul de angajați.
  - string filename: Numele fișierului CSV pentru raport.
- **Funcționare:**
  - Parcurge lista de angajați și filtrează operatorii de comenzi.
  - Calculează valoarea totală a comenzilor gestionate de fiecare operator.
  - Sortează operatorii descrescător după valoarea totală a comenzilor.
  - Scrie primii 3 operatori în fișier.
- **Ieșire:** Fișier CSV cu top 3 operatori și valorile comenzilor procesate.

3. **generateReportTop3HighestSalaries:**
   - Generează un raport cu top 3 angajați care au cele mai mari salarii.
   - **Intrări:**
     - EmployeeManager& manager: Referință la managerul de angajați.
     - string filename: Numele fișierului CSV pentru raport.
   - **Funcționare:**
     - Sortează angajații descrescător după salariul calculat.
     - Scrie primii 3 angajați (nume, prenume și salariu) în fișier.
   - **Ieșire:** Fișier CSV cu top 3 angajați și salariile lor.

---

3. Unde îndeplinește cerințele proiectului:

- **Rapoarte cerute:**
  - Clasa generează rapoarte despre:
    - Angajatul cu cele mai multe comenzi procesate.
    - Top 3 angajați care au gestionat cele mai valoroase comenzi.

        - Top 3 angajați cu cele mai mari salarii.
- **Export fișiere:**
    - Toate rapoartele sunt exportate în fișiere CSV, conform cerințelor proiectului.
- **Modularitate și separare a responsabilităților:**
    - Clasa Reports este dedicată exclusiv raportării, separând această funcționalitate de alte componente ale aplicației.

---

4. Posibile întrebări din partea profesoarei:

1. **De ce ai folosit metode statice?**
    - Metodele de raport nu depind de starea internă a unui obiect Reports. De aceea, utilizarea metodelor statice este mai eficientă și clară.
2. **Cum gestionezi cazul în care nu există operatori în sistem?**
    - În metoda generateReportMostOrdersProcessed, dacă lista operatorilor este goală, se scrie un mesaj corespunzător în fișier.
3. **Cum este sortată lista angajaților în funcție de salariu?**
    - În metoda generateReportTop3HighestSalaries, angajații sunt sortați descrescător după salariu folosind un comparator lambda.
4. **Cum te asiguri că rapoartele sunt salvate corect?**
    - Verific deschiderea fișierului înainte de a scrie datele și afișez un mesaj de eroare dacă fișierul nu poate fi accesat.
5. **Cum gestionezi legătura cu alte componente?**
    - Metodele din Reports se bazează pe EmployeeManager pentru accesarea angajaților și a datelor acestora.

Explicație pentru clasa VintageDisc

Clasa VintageDisc derivă din clasa Disc și reprezintă un tip specific de discuri, caracterizate prin condiția mint (ca noi) și coeficientul de raritate. Această clasă

implementează comportamente specifice pentru calcularea prețului de livrare și afișarea detaliilor.

---

1. Scopul Clasei:

Clasa VintageDisc este utilizată pentru:

- Reprezentarea discurilor vintage, care au caracteristici suplimentare față de discurile obișnuite.
- Calcularea prețului de livrare, incluzând un adaos bazat pe raritate.
- Afișarea completă a detaliilor despre produs, incluzând starea mint și coeficientul de raritate.

---

2. Membrii și metodele clasei:

Atribute Private:

1. bool isMintCondition:
    - Indică dacă discul este în condiție mint (ca nou).
    - Valoare booleană (true sau false).
2. int rarityCoefficient:
    - Coeficientul de raritate al discului, cuprins între 1 și 5.
    - Este utilizat pentru calcularea prețului de livrare.

Constructori:

1. **Constructor implicit:**
    - Inițializează discul cu condiția false și coeficientul de raritate 0.
2. **Constructor parametrizat:**
    - Inițializează discul cu toate valorile necesare, incluzând cele moștenite din Disc, precum și isMintCondition și rarityCoefficient.

Metode Publice:

1. **Calculare preț livrare:** float calculateDeliveryPrice() const override
    - Suprascrie metoda din Disc.
    - Calculează prețul final astfel:
        - Preț de bază + 5 RON (livrare standard pentru discuri) + 15 RON * coeficientul de raritate.

Exemplu: Pentru un disc cu preț de bază 100 RON și coeficient de raritate 3:

Preț final=100+5+(15×3)=150 RON.\text{Preț final} = 100 + 5 + (15 \times 3) = 150 \, \text{RON}.Preț final=100+5+(15×3)=150RON.

2. **Stare mint:** bool getIsMintCondition() const
    - Returnează valoarea atributului isMintCondition.
3. **Coeficient raritate:** int getRarityCoefficient() const
    - Returnează valoarea coeficientului de raritate.
4. **Tip produs:** string getType() const override
    - Returnează stringul "VintageDisc", indicând tipul produsului.
5. **Afișare detalii produs:** void displayProductDetails() const override
    - Suprascrie metoda din Disc.
    - Afișează toate detaliile relevante despre produs, inclusiv starea mint și coeficientul de raritate.

---

3. Unde îndeplinește cerințele proiectului:

- **Caracteristici suplimentare pentru discuri vintage:**
    - Adaugă condiția mint și coeficientul de raritate, extinzând funcționalitățile clasei Disc.
- **Calculare preț livrare:**
    - Prețul de livrare include costuri suplimentare bazate pe raritate, conform cerințelor.

- **Afișare detalii complete:**
    - Metoda displayProductDetails listează toate detaliile produsului, inclusiv caracteristicile unice ale discurilor vintage.

---

4. Posibile întrebări din partea profesoarei:

1. **Cum calculezi prețul final pentru un disc vintage?**
    - Prețul final este calculat prin adăugarea unui cost de 5 RON pentru livrare și 15 RON înmulțit cu coeficientul de raritate:

Preț final=Preț de baza˘+5+(15×Coeficient raritate)\text{Preț final} = \text{Preț de bază} + 5 + (15 \times \text{Coeficient raritate})Preț final=Preț de baza˘+5+(15×Coeficient raritate)

2. **Ce reprezintă condiția mint?**
    - Condiția mint (isMintCondition) indică dacă discul este în stare ca nou.
3. **Cum afișezi detaliile unui disc vintage?**
    - Metoda displayProductDetails afișează:
        - Numele, stocul, prețurile (bază și cu livrare).
        - Casa de discuri, data lansării, trupa și albumul.
        - Starea mint și coeficientul de raritate.
4. **Cum este implementată extensibilitatea pentru produse?**
    - Clasa VintageDisc moștenește clasa Disc și adaugă funcționalități suplimentare. Această ierarhie permite extinderea fără a afecta alte tipuri de produse.

Explicație pentru clasa OrderManager

Clasa OrderManager gestionează procesarea comenzilor în cadrul magazinului, atribuindu-le operatorilor de comenzi și monitorizând progresul acestora. Ea implementează logica pentru distribuția sarcinilor, coada de așteptare a comenzilor, și simularea procesării comenzilor în timp real.

---

1. Scopul Clasei:

Clasa OrderManager este utilizată pentru:

- Gestionarea comenzilor plasate de clienți.
- Atribuirea comenzilor operatorilor disponibili.
- Monitorizarea progresului comenzilor procesate.
- Simularea unui sistem în timp real pentru procesarea comenzilor.

---

2. Membrii și metodele clasei:

Atribute Private:

1. **vector<shared_ptr<OperatorComenzi>> operators**:
    - Lista operatorilor disponibili pentru procesarea comenzilor.
2. **queue<shared_ptr<Order>> orderQueue**:
    - Coada de comenzi care așteaptă să fie procesate.
3. **InventoryManager& inventory**:
    - Referință la managerul de inventar pentru validarea comenzilor.
4. **atomic<bool> isSimulationRunning**:
    - Indică dacă simularea procesării comenzilor este activă.
5. **unordered_map<int, chrono::steady_clock::time_point> orderStartTimes**:
    - Asociază comenzile cu momentul în care au fost atribuite unui operator.

Constructori:

1. **Constructor implicit:** OrderManager(InventoryManager& inv)
    - Inițializează managerul de comenzi cu inventarul specificat.
2. **Constructor parametrizat:** OrderManager(const vector<shared_ptr<OperatorComenzi>>& initialOperators, InventoryManager& inv)
    - Inițializează managerul cu o listă de operatori și inventarul specificat.

Metode Publice:

1. **Adăugare operator:** void addOperator(shared_ptr<OperatorComenzi> operatori)
    - Adaugă un operator în lista de operatori disponibili.
2. **Atribuire comandă:** void assignOrder(const shared_ptr<Order>& order)
    - Atribuie o comandă unui operator disponibil.
    - Dacă toți operatorii sunt ocupați, comanda este plasată în coada de așteptare.
3. **Start simulare:** void startSimulation()
    - Pornește simularea în timp real pentru procesarea comenzilor.
4. **Oprire simulare:** void stopSimulation()
    - Oprește simularea.
5. **Procesare comenzi în așteptare:** void processWaitingOrders()
    - Verifică și atribuie comenzile din coada de așteptare operatorilor disponibili.
6. **Distribuție sarcini:** void distributeWorkLoad()
    - Reechilibrează încărcarea de lucru între operatori.
7. **Verificare comenzi finalizate:** void checkCompletedOrders()
    - Verifică dacă există comenzi finalizate și actualizează statusul operatorilor.
8. **Afișare status operatori:** void displayOperatorStatus() const
    - Afișează starea curentă a operatorilor, comenzile atribuite și progresul acestora.
9. **Încărcare comenzi din fișier:** void loadOrdersFromFile(const string& filename)
    - Încarcă comenzi dintr-un fișier și le procesează.
10. **Obținere operatori:** vector<shared_ptr<OperatorComenzi>> getOperators() const
    - Returnează lista operatorilor disponibili.

---

3. Unde îndeplinește cerințele proiectului:

- **Gestionarea comenzilor:**
    - Comenzile sunt validate, atribuite operatorilor și monitorizate până la finalizare.
    - Comenzile care nu pot fi procesate imediat sunt plasate într-o coadă de așteptare.
- **Simulare în timp real:**
    - Procesarea comenzilor este simulată în timp real folosind un thread separat.
- **Distribuție eficientă a sarcinilor:**
    - Sarcinile sunt redistribuite automat pentru a echilibra încărcarea operatorilor.
- **Afișare status:**
    - Statusul operatorilor și progresul comenzilor sunt afișate clar și detaliat.

---

4. Posibile întrebări din partea profesoarei:

1. **Cum gestionezi coada de așteptare a comenzilor?**
    - Comenzile care nu pot fi atribuite operatorilor sunt plasate într-o coadă FIFO. Când un operator devine disponibil, comenzile sunt atribuite din coadă.
2. **Cum redistribui sarcinile între operatori?**
    - În metoda distributeWorkLoad, operatorii sunt sortați după numărul de comenzi atribuite, iar comenzile sunt redistribuite astfel încât să minimizeze dezechilibrele.
3. **Cum verifici progresul comenzilor?**
    - În metoda checkCompletedOrders, timpul scurs de la atribuire este comparat cu durata de procesare. Comenzile finalizate sunt marcate, iar cele restante rămân active.
4. **Cum încarci comenzile dintr-un fișier?**
    - Metoda loadOrdersFromFile citește datele din fișier, validează comenzile și le adaugă pentru procesare.
5. **Cum asiguri că simularea rulează eficient?**

- Simularea este realizată pe un thread separat, folosind verificări periodice (la fiecare secundă).

Analiză detaliată a main.cpp

1. Rolul fișierului main.cpp:

- Inițializează principalele componente ale aplicației: EmployeeManager, InventoryManager, și OrderManager.
- Populează sistemul cu angajați și produse predefinite.
- Verifică condițiile operaționale ale magazinului (stoc minim, structura de personal).
- Oferă un meniu interactiv utilizatorului pentru gestionarea angajaților, produselor, comenzilor și generarea rapoartelor.

---

2. Structura fișierului:

Inițializarea aplicației:

```
EmployeeManager employeeManager;
InventoryManager inventoryManager;


// Inițializare operatori și asociere cu `OrderManager`.
vector<shared_ptr<OperatorComenzi>> operators = {
  make_shared<OperatorComenzi>(2, "Maria", "Ionescu", "2980804123456", "15-06-2018"),
  make_shared<OperatorComenzi>(3, "Alex", "Pop", "1990501123456", "20-02-2023"),
  make_shared<OperatorComenzi>(4, "Elena", "Marin", "2980103123456", "05-03-2021")
};
OrderManager orderManager(operators, inventoryManager);


// Adăugare angajați în sistem.
```

```
employeeManager.addEmployee(make_shared<Manager>(1, "Ion", "Popescu",
"1990401123456", "01-01-2015"));

for (auto& operatorComenzi : operators) {

  employeeManager.addEmployee(operatorComenzi);

}

employeeManager.addEmployee(make_shared<Asistent>(5, "Elena", "Mocanu",
"2960508123456", "01-06-2019"));

employeeManager.addEmployee(make_shared<Asistent>(6, "Dan", "Niculescu",
"1950122123456", "15-11-2021"));
```

1. Creează managerul de angajați și managerul de inventar.
2. Inițializează operatorii de comenzi și îi adaugă în OrderManager.
3. Adaugă angajați predefiniți în EmployeeManager.

Popularea inventarului:

```
inventoryManager.addProduct(make_shared<Clothing>("Jacheta Rock Star", 10, 250.0,
"C001", "Negru", "RockLeather"));

// Adăugare alte produse (articole vestimentare, discuri și discuri vintage)...
```

- Produsele sunt adăugate în InventoryManager, inclusiv articole vestimentare, discuri standard și discuri vintage.

Verificarea stării operaționale:

```
if (!employeeManager.checkOperationalStatus() || !inventoryManager.checkStockValidity())
{

  cout << "Magazinul nu poate functiona. Se inchide programul.\n";

  return 1;

}
```

- Aplicația verifică dacă:
    - Există suficienți angajați pentru funcționare.

- Stocul magazinului respectă cerințele minime.

Dacă una dintre aceste condiții nu este îndeplinită, aplicația se închide.

Meniul interactiv:

```
bool running = true;
while (running) {
   cout << "\n=== Meniu Principal ===\n";
   cout << "1. Gestiune Angajati\n";
   cout << "2. Gestiune Stoc\n";
   cout << "3. Procesare Comenzi\n";
   cout << "4. Generare Rapoarte\n";
   cout << "5. Iesire\n";
   cout << "Selectati o optiune: ";
   int option;
   cin >> option;

   switch (option) {
   case 1:
      manageEmployees(employeeManager);
      break;
   case 2:
      manageInventory(inventoryManager);
      break;
   case 3:
      processOrders(orderManager);
      break;
```

```
    case 4:
        generateReports(employeeManager);
        break;
    case 5:
        cout << "Iesire din program.\n";
        orderManager.stopSimulation();
        running = false;
        break;
    default:
        cout << "Optiune invalida. Incercati din nou.\n";
        break;
    }
}
```

- Utilizatorul poate selecta una dintre opțiunile din meniu:
    1. **Gestiunea angajaților:** Accesează funcția manageEmployees.
    2. **Gestiunea stocului:** Accesează funcția manageInventory.
    3. **Procesarea comenzilor:** Accesează funcția processOrders.
    4. **Generarea rapoartelor:** Accesează funcția generateReports.
    5. **Ieșire:** Oprește simularea și închide aplicația.

---

3. Funcționalități detaliate:

1. **Gestiunea angajaților (manageEmployees):**
    - Permite adăugarea, ștergerea și modificarea angajaților.
    - Validează datele introduse (CNP, data angajării).
    - Utilizează polimorfism pentru a crea diferite tipuri de angajați.
2. **Gestiunea stocului (manageInventory):**

- Permite adăugarea, ștergerea și modificarea produselor.
- Acceptă mai multe tipuri de produse: Clothing, Disc, VintageDisc.
- Validează unicitatea codurilor produselor.

3. **Procesarea comenzilor (processOrders):**
   - Încarcă comenzile dintr-un fișier CSV.
   - Atribuie comenzile operatorilor disponibili sau le plasează în coada de așteptare.
4. **Generarea rapoartelor (generateReports):**
   - Creează fișiere CSV cu informații despre angajați și comenzi:
     - Operatorii cu cele mai multe comenzi procesate.
     - Top 3 operatori cu cele mai valoroase comenzi.
     - Top 3 angajați cu cele mai mari salarii.

---

4. Posibile întrebări din partea profesoarei:

1. **Cum asiguri validarea datelor la adăugarea unui angajat?**
   - Prin metode precum Employee::validateCNP și Employee::isValidDate.
2. **Cum verifici funcționarea magazinului înainte de a începe simularea?**
   - Prin EmployeeManager::checkOperationalStatus și InventoryManager::checkStockValidity.
3. **Cum tratezi comenzile care nu pot fi procesate imediat?**
   - Acestea sunt plasate în coada de așteptare prin OrderManager::assignOrder.
4. **Cum generezi rapoartele?**
   - Utilizând funcțiile din clasa Reports pentru a crea fișiere CSV.
5. **Cum se redistribuie comenzile între operatori?**
   - Prin OrderManager::distributeWorkLoad, care echilibrează încărcarea între operatori.