Cerințe proiect laborator POO

Sumar

- notă bazată pe un proiect individual cu trei etape de evaluare
- note de la 1 la 12 pentru fiecare etapă (10 de bază + maxim 2 puncte bonus)
- nota finală este media notelor la fiecare etapă și trebuie sa fie minim 5
- tema aleasă trebuie confirmată cu mine pe email / teams sau la laborator și trecută în acest tabel **până la sfârșitul săptămânii 5 (26.03.2023)**
- proiectele vor fi corectate prin prezentarea la laborator (pentru cazuri speciale, adică studenți care nu pot ajunge la laborator, îmi puteți scrie pe mail)
- baremele de notare pentru fiecare etapă se găsesc mai jos în document
- **bonus** - +0.5p la nota finală pentru cine rezolvă corect și frumos scris toate exercițiile din laboratoare

Date finale pentru fiecare etapă

Codul pentru fiecare etapă se poate scrie până la:

- sfârșitul săptămânii 7 (09.04.2023) pentru etapa 1
- sfârșitul săptămânii 9 (30.04.2023) pentru etapa 2
- sfârșitul săptămânii 12 (21.05.2023) pentru etapa 3

Prezentările de proiect se pot susține oricând până în săptămâna 13.

Prezentare generală

Nota voastră la laboratorul de POO se acordă pe baza unui **proiect individual**, dezvoltat pe parcursul primelor 13 săptămâni din semestru[1]. Proiectul va fi evaluat în **trei etape**, cu cerințe și deadline-uri distincte pentru fiecare[2].

Proiectul constă în dezvoltarea unei **aplicații C++** care **să gestioneze un model de date**, folosind paradigma de **programare orientată pe obiecte**. Va trebui să definiți atât clasele care să reprezinte entitățile din modelul vostru, cât și metodele/funcțiile corespunzătoare care să implementeze „logica de business”.

[1] În ultima săptămână veți da un colocviu (test pe calculator), până la acel moment trebuie să aveți situația la laborator încheiată.

[2] Cerințele sunt gândite în așa fel încât să continuați programul din etapa anterioară, adăugând funcționalități în plus la codul existent.

Notare

Pentru fiecare etapă, veți primi **o notă de la 1 la 12**. Nota 10 se acordă pentru implementarea corectă a tuturor **cerințelor de bază**, putând primi până la două puncte în plus dacă rezolvați și **cerințele suplimentare** (vor fi indicate corespunzător).

Nota finală este media notelor de pe fiecare etapă. Trebuie să aveți minim **nota 5** ca să promovați laboratorul și să puteți participa la colocviu/examen.

Alegerea temei

Recomandarea mea este să alegeți o temă care vă pasionează, ca să fiți motivați să lucrați la proiect și să îl duceți până la capăt.

Exemple de teme

- Magazin online (produse, cumpărători, comenzi, reduceri etc.)
- Gestiunea resurselor umane (firmă, angajați, echipe, salarii etc.)
- Gestiunea unei clinici/unui spital (medici, pacienți, consultații, medicamente etc.)
- Project management (proiect, task, echipă, membru al echipei etc.)
- Gestiunea școlarității (studenți, note, discipline, profesori etc)
- Bibliotecă (carte, autor, categorie, împrumut etc.)
- Music player (album, artist, melodie, playlist etc.)
- Grădină zoologică (animale, bilete, hrană pentru animale etc.)
- Formula 1 (echipe, mașini, piloți, raliuri etc.)
- Cinematograf (filme, actori, vizionări, bilete etc.)

Puteți alege orice alt domeniu doriți. Indiferent de tema aleasă, va **trebui să o confirmați** cu mine (în persoană când veniți la laborator, sau pe e-mail/Teams dacă nu puteți ajunge) **până la sfârșitul săptămânii 6** de facultate.

De preferat ar fi să vă alegeți teme distincte. Cel mult doi studenți pot avea o temă identică/similară. Chiar și în acest caz, vor trebui să aibă o structură diferită a claselor.

Important: este permis și chiar recomandat să vă ajutați între voi la proiecte, însă nu este tolerat să copiați cod unii de la alții sau din proiecte publice de pe internet.

Nu uitați că atât colocviul cât și examenul sunt individuale, deci va trebui să fiți capabili să le rezolvați singuri.

Recomandare: după ce v-ați ales tema și ați confirmat-o, puteți începe prin a vă face o „schiță" cu clasele de care veți avea nevoie, ce date și metode vor reține fiecare și care vor fi legăturile dintre ele. Puteți face asta pe hârtie sau folosind o aplicație cum ar fi Excalidraw.

Recomandare: familiarizați-vă cât mai curând cu și începeți să folosiți debuggerul din mediul vostru de lucru. O să vă petreceți 20% din timp scriind efectiv cod și peste 80% din timp încercând să înțelegeți de ce nu funcționează cum v-ați fi așteptat.

Cerințe etapa 1

Deadline: cerințele trebuie implementate până la sfârșitul **săptămânii 7** de facultate (**09.04.2023**). Pentru a fi punctați, va trebui să le și prezentați la laboratorul din **săptămâna 6**.

Criterii generale **(1p)**

- Trebuie să aveți în `main` un cod corespunzător care să testeze/utilizeze funcționalitățile implementate.
- Clasele/metodele care nu sunt utilizate sau la care nu se face referire nicăieri **nu vor fi punctate**.
- Pentru etapa 1, **nu** aveți voie să folosiți clasele container din STL (`std::vector`, `std::string` șamd). Puteți refolosi în schimb clasele implementate de voi la laborator (este recomandat să le refolosiți).
- Încercați să păstrați codul curat:
  - Folosiți nume de variabile/funcții/tipuri de date cu sens (e.g. `nr_angajati` în loc de `n`).
  - Folosiți formatarea automată oferită de editorul vostru de text și încercați să păstrați un stil uniform.
  - Folosiți mai degrabă mai multe clase/metode mai scurte, decât o singură clasă/metodă foarte lungă.

Versionarea codului **(1p)**

- Codul sursă **trebuie** să fie încărcat pe GitHub.

- Trebuie să îmi dați și mie **acces de citire** la repository-ul în care aveți proiectul, dacă nu este public.

  Instrucțiunile pentru cum puteți adăuga colaboratori la un repo se găsesc aici. Mă puteți adăuga prin username (`@JustBeYou`) sau prin e-mail (`mihailferaru2000@gmail.com`).

- **Recomandare:** adăugați de la început un fișier `.gitignore` la repo-ul vostru ca să nu includeți accidental fișierele compilate / binare în Git. Puteți găsi aici un exemplu de fișier `.gitignore` pentru C++.

- **Recomandare:** păstrați commit-urile concise și independente. Găsiți aici un set de bune practici pentru commit-urile de Git.

Documentație **(1p)**

- În repo-ul de pe GitHub trebuie să aveți și un fișier `README`, de preferat formatat cu Markdown, în care să includeți cel puțin:
  - **Numele** proiectului
  - **Tema** aleasă
  - O listă cu **clasele** pe care le-ați implementat și o scurtă descriere a ce reprezintă fiecare
  - O listă cu **funcționalitățile** pe care le are aplicația voastră la momentul actual (ex.: „capabilă să citească și să rețină o listă de angajați”, „poate calcula prețul mediu al produselor aflate în stoc” etc.)

- **Recomandare:** actualizați acest fișier pe parcurs ce dezvoltați proiectul. Vă va fi mult mai ușor să-l prezentați altor persoane.

Clase **(2p)**

- Trebuie să respectați **principiul encapsulării** (nu aveți voie să aveți date membre publice). Metodele pot fi publice sau private.

- Trebuie să definiți **minim 5 clase**, relevante pentru tema aleasă.

  Sunt luate în considerare și clasele utilitare, cum ar fi o clasă pentru șiruri de caractere sau una pentru vectori alocați dinamic.

- **Toate clasele** vor avea constructori cu parametri sau fără care să inițializeze membrii clasei.

- **Minim 3 dintre clase** trebuie să fie corelate prin **compunere** (ex. să aveți o dată membru de tip `Adresă` în clasa `Contact`, să aveți un vector de `Angajat` în clasa `Companie` etc.)

Metode **(2p)**

- **Minim 3 metode** care operează cu datele membru ale claselor (e.g calculează prețul unui produs aplicând o reducere, returnează salariul mediu al angajaților din firmă, actualizează TVA-ul unei facturi și recalculează totalul etc.). Nu se vor lua în considerare getter-ii și setter-ii simpli și fără logică "de business" [3].
- **Minim o metodă** va fi supraîncărcată (e.g. reducerea aplicată poate fi un întreg de la 0 la 100 sau un număr real între 0 și 1).

Citire și afișare **(1p)**

- Pentru **minim o clasă** trebuie să implementați o metodă de **afișare** și una de **citire** supraîncărcând operatorii `operator<<` și `operator>>`.

Alocare dinamică **(1p)**

- **Minim o clasă** trebuie să facă alocare dinamică de memorie (fie pentru un singur obiect, fie pentru un vector de obiecte).
- **Minim o clasă** care face alocare dinamică va avea implementat un constructor de copiere și operatorul `operator=`.
- Toată memoria alocată dinamic trebuie **eliberată** corespunzător. Veți fi depunctați pentru *memory leaks*.
- **Minim o clasă** trebuie să aibă un **destructor** (netrivial, adică cu eliberare de memorie).

Bonus

Fiecare cerință bonus valorează **1 punct**.

- Implementați un **meniu interactiv** în `main`, care să permită:
  - **Citirea** de la tastatură și **crearea** a **cel puțin 3 tipuri de obiecte** dintre cele definite de voi;

[3] Ce este business logic-ul găsiți aici: https://en.wikipedia.org/wiki/Business_logic.

  - **Afișarea** acestor obiecte, după ce au fost citite de aplicație;
  - **Apelarea unei metode** pe ele (alta în afară de cea de afișare).
- Implementați **constructorul de mutare** și supraîncărcați `operator=` **de mutare** pentru cel puțin o clasă care gestionează memorie alocată dinamic.

Oficiu (1p)

Cerințe etapa 2

Deadline: cerințele trebuie implementate până la sfârșitul **săptămânii 9** de facultate (**30.04.2023**).

Începând cu etapa a 2-a, **aveți voie** să folosiți clasele container din biblioteca standard (`std::string`, `std::vector` etc.)

Moștenire (2p)

Sugestii de implementare: puteti incerca sa adaugati mostenirea in proiectul vostru fie pentru a reduce duplicarea de logica (ex. daca aveti cel putin doua clase cu multa logica comuna, puteti sa extrageti elementele comune intr-o clasa parinte si sa mosteniti) sau pentru a extinde functionalitatea unei clase existente (ex. daca aveti o clasa Produs, creati o clasa ProdusCuDiscount care sa o mosteneasca si sa implementeze logica suplimentara de discount) sau puteti folosi mostenirea pentru a implementa interfete (vezi mai jos).

- Definiți **minim două ierarhii diferite de moștenire** (două ierarhii de moștenire sunt considerate diferite dacă nu au aceeași clasă de bază și aceeași clasă care moștenește clasa/clasele de bază).
- Utilizați **minim doi modificatori de acces diferiți** pe clasa care se moștenește (`public`/`protected`/`private`); alegerea vă aparține, dar trebuie să fie justificată în funcție de nevoile proiectului.
- **Minim o clasă** care să folosească moștenire multiplă (să extindă mai multe clase/interfețe).
- Apelați **cel puțin o dată** un constructor (cu parametri) dintr-o clasă de bază, folosind o listă de inițializare în constructorul clasei copil.
- **Minim două date membru** și **minim o metodă** care să aibă modificatorul de acces `protected` (în mod util, să fie accesate/apelate dintr-o clasă care le moștenește).

Interfețe și metode virtuale (2p)

- Definiți și extindeți (moșteniți) **minim o interfață** (clasă fără date membru, doar metode pur virtuale și un destructor virtual) care să aibă **minim două metode** (alternativ: minim două interfețe, fiecare cu cel puțin o metodă).
- Definiți și extindeți (moșteniți) **minim o clasă de bază abstractă** (clasă care poate avea date membru, dar are cel puțin o metodă pur virtuală).
- Trebuie să aveți / să identificați în proiect **cel puțin o situație** în care să fie nevoie de și să se apeleze destructorul virtual (i.e. să ștergeți un obiect alocat dinamic de tipul unei clase moștenitoare, la care să vă referiți prin intermediul unui pointer la clasa de bază).
- Definiți **cel puțin 4 metode virtuale** care să fie suprascrise în clasele moștenitoare. Pot fi pur virtuale sau cu o implementare implicită. Se iau în considerare și metodele definite la celelalte subpuncte, exceptând destructorii virtuali.

Polimorfism la execuție (2p)

- Identificați **minim 4 locuri** în care să aibă loc polimorfism la execuție (*dynamic dispatch*) în proiectul vostru (e.g. apelul unor metode virtuale prin intermediul unor pointeri/referințe către clasa de bază).
- Identificați **minim 4 instanțe de upcasting** în codul vostru (e.g. atribuirea unor obiecte de tipul unor clase moștenite la pointeri/referințe către clasa de bază).
- Realizarea downcasting-ului **în cel puțin o situație** în codul vostru (unde are sens), folosind `dynamic_cast` sau RTTI.

Excepții (2p)

- Definiți **minim un tip de excepție custom**, care să extindă `std::exception`.
- Aruncați excepții în **minim 4 funcții/metode diferite** (folosiți tipuri de excepții definite de voi sau cele din biblioteca standard).
- Implementați **minim un bloc** `try...catch` care să prindă o excepție aruncată de voi (cu menționea explicită a tipului acesteia) și să o trateze într-un fel (în funcție de specificul erorii).
- Implementați **minim un bloc** `try...catch` care să prindă o excepție, să o proceseze și să re-arunce un alt tip de excepție din blocul `catch`.

Variabile și metode statice (1p)

- Definiți o variabilă membru statică în **cel puțin o clasă**.
- Implementați **cel puțin două metode statice** în clasele voastre (din care **cel puțin una** trebuie să acceseze/folosească variabila statică definită la subpunctul anterior).

Bonus (2p)

- Utilizati smart pointers pentru alocarile dinamice din proiectul vostru in loc de raw pointers cu new si delete.
- Utilizarea **a minim doua lambda expresii** pentru a parametriza funcționalitatea unei funcții sau a unei clase. Puteți transmite ca parametru sau reține o lambda expresie folosind tipul de date `std::function`.

  Se acordă punctaj parțial dacă doar folosiți lambda expresii pentru apelarea unor funcții care deja există în biblioteca standard (de exemplu, `std::sort`), fără să definiți voi o funcție/clasă care să primească un obiect de acest tip.

Oficiu (1p)

Cerințe etapa 3

Deadline: cerințele trebuie implementate până la sfârșitul **săptămânii 12** de facultate (21.05.2023).

Programare generică (3p)

- Utilizați **minim o clasă șablon** (`template`) **definită de voi**. Trebuie să fie parametrizată de **cel puțin un tip de date generic** (cel puțin un `typename`), care să fie folosit în mod util în interiorul clasei (e.g. pentru a defini un atribut, o metodă etc.).
- Definiți și apelați **minim o funcție șablon** (poate fi funcție liberă sau metodă a unei clase care nu este neapărat generică). Trebuie să fie parametrizată de **cel puțin un tip de date generic** (cel puțin un `typename`), care să fie folosit în definirea funcției (e.g. parametru, tip de date returnat).
- Utilizați clasa și funcția șablon definită mai sus în cel puțin un loc în proiectul vostru.

  Indicație: Puteți folosi o clasă existentă din proiectul vostru pe care să o generalizați cu șabloane, nu e neapărat nevoie să adaugați o clasă nouă.

Design patterns (2p)

Citiți despre câteva design patterns listate în acest catalog. Alegeți un pattern pe care:

- îl regăsiți deja implementat în proiectul vostru (nu mai este nevoie să implementați nimic în acest caz) sau,
- îl considerați potrivit pentru a rezolva o problema din proiectul vostru și implementați-l.

Observație: Va trebui sa puteți explica utilitatea și implementarea pattern-ului ales pentru punctare.

Biblioteca standard (2p)

- Utilizarea a **minim două tipuri de date *container* diferite** din STL.

  Exemple: `vector`, `string`, `array`, `list`, `set`, `map` etc.
- Utilizarea a **minim o funcție utilitară** din biblioteca standard (funcții libere, nu metode).

Exemple: `sort`, `find`, `search`, `all_of/any_of/none_of`, `accumulate`, `fill`, `generate`, `copy`, `reverse`, orice alte funcții din fișierul header din biblioteca standard `<algorithm>`.

Indicație: Înlocuiți în proiectul vostru un array classic cu container-ul `vector`. Folosiți `set` pentru a găsi elementele unice într-un vector sau `map` pentru a număra aparițiile unui element.

Smart pointers (2p)

Înlocuiți alocarea dinamică manuală (cea cu `new`, `delete` și raw pointers) cu smart pointers în cel puțin un loc în aplicația voastră.

Prin *smart pointer* ne referim la una dintre clasele `std::reference_wrapper`[4], `std::unique_ptr`, `std::shared_ptr` sau `std::weak_ptr`.

Indicație: Puteți să folosiți aceste clase în locul referințelor sau pointerilor obișnuiți din codul vostru.

Referințe utile: avantajele *smart pointers*, utilizarea *smart pointers* în C++.

Bonus (2p)

Fiecare cerință bonus valorează **1 punct**.

- **Utilizarea modificatorului `const`** în mod corect, peste tot unde este posibil și are sens în program (pe date membre, pe metode, la transmiterea parametrilor prin referință etc.).

  Referințe utile: `const` correctness.

- Folosirea **unei biblioteci utilitare** din biblioteca standard, în afară de cele cu care am lucrat la laborator. Puteți alege (în funcție de specificul proiectului) dintre:
  - `chrono`: lucru cu date și timp.
  - `regex`: lucru cu expresii regulate (pentru căutarea/înlocuirea de text).
  - `random`: generare de numere aleatoare.
  - `thread`: execuție paralelă a codului.

[4] `reference_wrapper` este mai degrabă un *smart reference*, dar se utilizează în mod similar cu un *smart pointer*.

Pentru a fi punctați, trebuie să **instanțiați** și să **utilizați** cel puțin câteva tipuri de date din biblioteca aleasă. De exemplu, pentru `random`: să inițializați un generator de numere random, să alegeți o distribuție, să generați niște numere aleatoare pe care să le utilizați în aplicație.

Oficiu (1p)