Cerințe proiect laborator POO

Cerințe etapa 3

Pentru a satisface aceste cerințe, puteți refolosi/extinde codul de la etapa 1 sau 2, actualizându-vă proiectul pentru a include toate conceptele noi. Alternativ, puteți să vă schimbați tema (dar va trebui să mă anunțați).

Proiectul vostru trebuie să respecte în continuare **Criteriile generale** de la prima parte (cod curat, care respectă encapsularea, fără variabile globale, fără clase/-metode care nu sunt folosite/apelate nicăieri), precum și indicațiile referitoare la **Versionarea codului** (continuați să încărcați ce lucrați pe Git) și la **Documentație** (actualizați corespunzător `README`-ul).

Design patterns

- Implementați **3 *object-oriented design patterns* diferite** (se acordă până la un punct pentru fiecare design pattern implementat și folosit corect). (3p)

  Observație: pattern-ul trebuie să se regăsească în codul scris de voi. De exemplu, dacă ați folosit o clasă `iterator` deja existentă din biblioteca standard, nu înseamnă că ați implementat design pattern-ul *iterator*.

  Referințe utile: *design patterns* cu exemple de cod și când se utilizează, catalog de *design patterns* comune.

Programare generică

- 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.). (1p)
- 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). (1p)

Biblioteca standard

- Utilizați **minim două tipuri de date *container* diferite** din STL în clasele definite de voi. (1p)

  Exemple: `vector`, `array`, `list`, `set`, `map` etc.

- Utilizați **minim două funcții utilitare diferite** din biblioteca standard (funcții libere, nu metode ale claselor din STL). (1p)

  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>`.

- Utilizați în mod corespunzător **două tipuri diferite de *smart pointers*** din biblioteca standard (se acordă un punct dacă ați folosit corect un tip de smart pointers, două puncte dacă ați folosit două tipuri diferite). (2p)

  Prin *smart pointer* ne referim la una dintre clasele `std::reference_wrapper`[1], `std::unique_ptr`, `std::shared_ptr` sau `std::weak_ptr`.

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

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

Oficiu (1p)

Bonus

Utilizați în proiectul vostru o **bibliotecă externă** (alta decât biblioteca standard). Poate să ofere orice fel de funcționalitate care se potrivește cu nevoile temei voastre: interfață grafică, animații, audio, importarea/exportul datelor în diferite formate, conectarea la o bază de date, conexiuni pe rețea, interacțiune cu hardware-ul, algoritmi specializați etc.

Puteți primi până la **2 puncte bonus** pentru această cerință, în funcție de complexitatea bibliotecii alese și cât de bine se integrează cu proiectul vostru.

Pentru a primi punctajul complet, trebuie să includeți biblioteca în proiectul vostru într-un mod portabil, configurând în mod corespunzător *build system*-ul folosit (CodeBlocks Project, Visual Studio Project, CMake, Makefile etc.), nu copiind fișierele în proiectul/repository-ul vostru.

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