AVANSAT · POO ÎN C++ · LECȚIA 5

Proiectarea claselor: compunere vs moștenire, principiile SOLID și injectarea dependențelor

Durată: 90 min Nivel: avansat / pregătire pentru facultate și industrie Înainte: lecțiile oop-04

CE VEI ȘTI DUPĂ LECȚIE

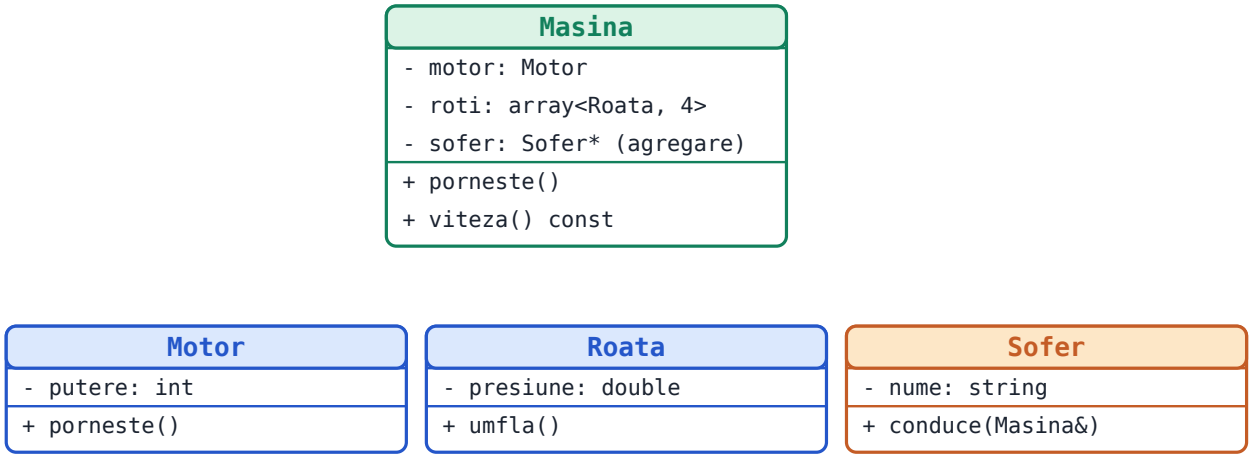
- Deosebești relațiile dintre clase: asociere, agregare, compunere, moștenire, dependență.
- Alegi compunerea în locul moștenirii acolo unde relația nu este „este un”.
- Explici și aplici cele cinci principii SOLID, cu exemple greșit/corect.
- Folosești injectarea dependențelor prin interfețe, ca să poți testa codul cu obiecte false.
- Recunoști principiile DRY, KISS, YAGNI și legea lui Demeter.

1. Codul bun se schimbă ușor

Un program de concurs se scrie o dată și nu se mai atinge. Un program real trăiește ani de zile, este modificat de mulți oameni, iar cerințele se schimbă. **Proiectarea** claselor înseamnă să împarți responsabilitățile astfel încât o schimbare să atingă **puține** locuri și să nu strice altele. Lecția aceasta adună regulile pe care programatorii experimentați le aplică aproape instinctiv.

2. Relațiile dintre clase

Relația	Înseamnă	În C++	Exemplu
dependență	„folosește un”, temporar	parametru sau variabilă locală	`Raport::tipareste(Imprimanta&)`
asociere	„cunoaște un”	membru pointer/referință	`Elev` cunoaște `Clasa` lui
agregare	„are un”, dar partea trăiește independent	pointer, referință, `shared_ptr`	`Masina` are un `Sofer`
compunere	„este format din”, partea moare cu întregul	membru obiect sau `unique_ptr`	`Masina` are un `Motor`
moștenire	„este un”	`class B : public A`	`Cerc` este o `Forma`



Motor și Roata sunt părți ale mașinii (mor odată cu ea); Sofer există independent (doar un pointer)

Compunere și agregare

3. Compunere în loc de moștenire

Moștenirea creează cea mai **strânsă** legătură posibilă între două clase: derivata depinde de toate detaliile bazei, iar o schimbare în bază poate strica toate derivatele. Regula practică: **preferă compunerea**; folosește moștenirea doar pentru „este un” și pentru polimorfism (o interfață comună).

Exemplu greșit: o stivă „moștenește” un vector ca să refolosească codul.

```
class Stiva : public vector<int>          // GREȘIT: o stivă NU este un vector
{
public:
    void push(int x) { push_back(x); }
    int top() const { return back(); }
};
// Stiva s; s.insert(s.begin(), 5);  -> se compilează: oricine poate strica regula LIFO
```

Corect: stiva **are** un vector și expune doar operațiile de stivă.

```
class Stiva
{
    vector<int> v;                        // compunere: detaliu ascuns
public:
    void push(int x) { v.push_back(x); }
    void pop() { v.pop_back(); }
    int top() const { return v.back(); }
    bool empty() const { return v.empty(); }
};
```

DE CE AȘA?

De ce este mai bine? Interfața clasei spune exact ce se poate face (doar LIFO), iar implementarea se poate schimba (`deque` , listă) fără ca utilizatorii să afle. Așa este construit și `std::stack` : un **adaptor** care conține un container, nu unul care îl moștenește.

4. Principiile SOLID

Literă	Principiul	Pe scurt
S	Single Responsibility	o clasă are un singur motiv să se schimbe
O	Open/Closed	deschisă pentru extindere, închisă pentru modificare
L	Liskov Substitution	o derivată poate înlocui baza fără surprize
I	Interface Segregation	mai multe interfețe mici, nu una mare
D	Dependency Inversion	depinzi de abstracții (interfețe), nu de clase concrete

S: o singură responsabilitate

Greșit: o clasă care calculează mediile, le formatează ca HTML **și** le salvează pe disc. Dacă se schimbă formatul fișierului, modifici aceeași clasă ca atunci când se schimbă formula mediei: două motive de schimbare, riscul de a strica unul când îl repari pe celălalt.

Corect: `CalculatorMedii` (doar calcule), `FormatorHtml` (doar aspect), `DepozitFisiere` (doar salvare). Fiecare se testează și se schimbă separat.

> **SFAT**
>
> Testul rapid: încearcă să descrii clasa într-o propoziție **fără „și”**. „Clasa calculează mediile” este bine; „calculează mediile și le salvează și trimite e-mail” înseamnă trei clase.

O: deschis pentru extindere, închis pentru modificare

Greșit: o funcție care face `switch` pe tipul formei. La fiecare formă nouă trebuie modificată (și retestată) funcția, și toate celelalte `switch`-uri de același fel din program.

```
double arie(const FormaSimpla &f)
{
    switch (f.tip)                          // GREȘIT: la fiecare formă nouă modifici aici
    {
        case CERC: return 3.14159 * f.r * f.r;
        case DREPTUNGHI: return f.a * f.b;
    }
    return 0;
}
```

Corect: ierarhia din lecția oop-04. O formă nouă este o clasă **nouă**; codul existent rămâne neatins.

L: substituția Liskov

Orice cod care funcționează cu baza trebuie să funcționeze la fel de corect cu orice derivată. Exemplul clasic: un `Patrat` derivat din `Dreptunghi`. Pentru dreptunghi, codul `d.setLatime(5);`

`d.setInaltime(4);` se așteaptă ca aria să fie 20. Un pătrat care, ca să rămână pătrat, schimbă ambele laturi la fiecare apel dă 16: **surprinde** codul scris pentru dreptunghiuri. Pătratul matematic „este un" dreptunghi, dar un `Patrat` *modificabil* nu se comportă ca un `Dreptunghi` modificabil. Soluții: amândouă derivă din `Forma`, sau formele devin imutabile.

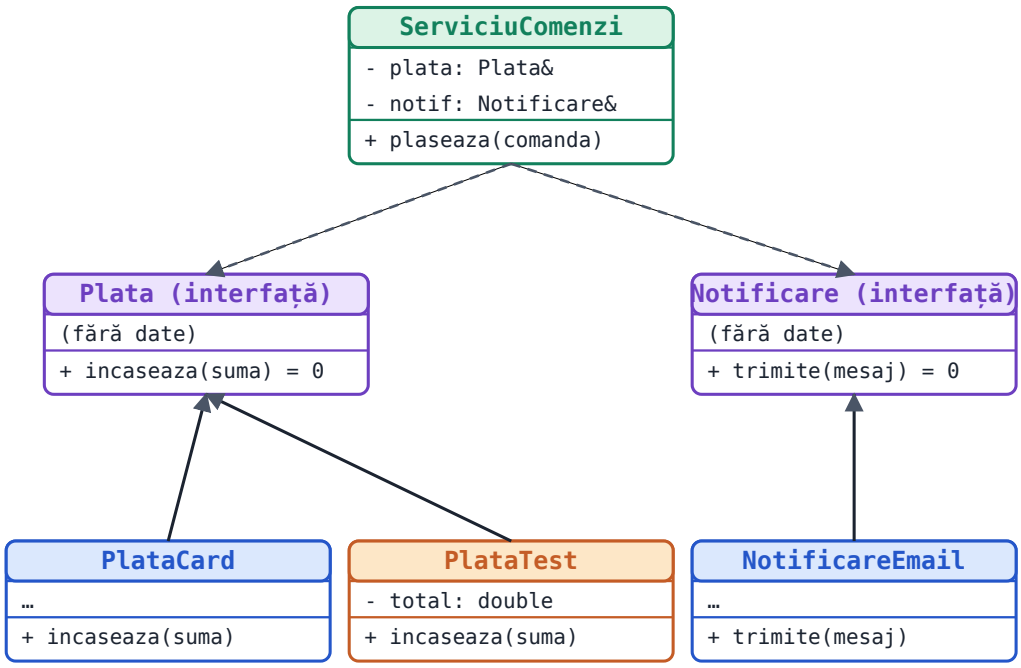
> **DE ȘTIUT**
>
> Semne că LSP este încălcat: derivata aruncă excepții acolo unde baza nu arunca („`Pinguin::zboara()` aruncă *nu pot zbura*"), are precondiții mai stricte sau nu respectă invariantul bazei. De obicei înseamnă că ierarhia este greșită (nu toate păsările zboară, deci `zboara()` nu aparține lui `Pasare`).

I: interfețe mici

Greșit: o interfață `Multifunctionala` cu `tipareste`, `scaneaza`, `trimiteFax`. O imprimantă simplă trebuie să implementeze și `scaneaza` (gol sau cu excepție): încalcă și LSP. Corect: interfețe separate, `Imprimabil`, `Scanabil`, `Fax`; un aparat multifuncțional le implementează pe toate, unul simplu doar pe cea de care are nevoie.

D: inversarea dependențelor

Modulele de nivel înalt (logica aplicației) nu trebuie să depindă de detalii de nivel jos (card bancar, e-mail, bază de date), ci **amândouă de o abstracție**. În practică: logica primește o **referință la o interfață** (prin constructor), iar cine construiește obiectul decide ce implementare concretă primește. Aceasta se numește **injectarea dependențelor** (*dependency injection*).



serviciul depinde doar de interfețe; implementările concrete (reale sau de test) se injectează din afară

Inversarea dependențelor

5. Exemplu complet: serviciu de comenzi testabil

`ServiciuComenzi` nu știe dacă plata se face cu cardul sau într-un test. În `main` îi dăm o plată „de test", cu o limită, și o notificare care doar afișează: putem verifica logica fără bancă și fără e-mail.

```cpp
#include <iostream>
#include <string>
using namespace std;

struct Comanda
{
    string client;
    double suma;
};

class Plata                                 // abstracție: ce înseamnă „a încasa”
{
public:
    virtual bool incaseaza(const string &client, double suma) = 0;
    virtual ~Plata() = default;
};

class Notificare                            // abstracție: ce înseamnă „a anunța”
{
public:
    virtual void trimite(const string &catre, const string &mesaj) = 0;
    virtual ~Notificare() = default;
};

class ServiciuComenzi                       // logica: nu știe nimic despre carduri sau e-
mail
{
    Plata &plata;
    Notificare &notif;
    double incasat = 0;
    int respinse = 0;
public:
    ServiciuComenzi(Plata &p, Notificare &n) : plata(p), notif(n) {}   // dependențele vin
din afară

    void plaseaza(const Comanda &c)
    {
        if (c.suma <= 0)
        {
            respinse++;
            notif.trimite(c.client, "comanda invalida");
            return;
        }
        if (plata.incaseaza(c.client, c.suma))
        {
            incasat += c.suma;
            notif.trimite(c.client, "plata acceptata");
        }
        else
        {
            respinse++;
            notif.trimite(c.client, "plata respinsa");
        }
    }
    double totalIncasat() const { return incasat; }
    int totalRespinse() const { return respinse; }
};
```

```
class PlataCuLimita : public Plata        // implementare „de test”: acceptă până la o limită
{
    double limita;
public:
    explicit PlataCuLimita(double l) : limita(l) {}
    bool incaseaza(const string &, double suma) override { return suma <= limita; }
};

class NotificareConsola : public Notificare
{
public:
    void trimite(const string &catre, const string &mesaj) override
    {
        cout << '[' << catre << "] " << mesaj << '\n';
    }
};

int main()
{
    int n;
    double limita;
    cin >> n >> limita;
    PlataCuLimita plata(limita);
    NotificareConsola notif;
    ServiciuComenzi serviciu(plata, notif); // injectăm implementările alese
    for (int i = 0; i < n; i++)
    {
        Comanda c;
        cin >> c.client >> c.suma;
        serviciu.plaseaza(c);
    }
    cout << "incasat " << serviciu.totalIncasat() << ", respinse " << serviciu.totalRespinse() << '\n';
    return 0;
}
```

DE CE AȘA?

Ce am câștigat? (1) **Testare:** logica de comenzi se verifică fără bancă reală; în proiectele mari, obiectele „false” (*mock*, *fake*) sunt baza testelor automate. (2) **Schimbare ușoară:** o plată prin transfer bancar este o clasă nouă; `ServiciuComenzi` nu se modifică (O din SOLID). (3) **Responsabilități clare:** fiecare clasă face un singur lucru (S). **De ce referințe și nu obiecte membre?** Serviciul nu deține plata și notificarea (agregare); viața lor o controlează cine le-a creat, aici `main`.

6. Alte principii de proiectare

Principiul	Înseamnă	Încălcare tipică
DRY (*Don’t Repeat Yourself*)	fiecare cunoștință apare o singură dată	aceeași formulă copiată în trei locuri

Principiul	Înseamnă	Încălcare tipică
KISS (*Keep It Simple*)	soluția cea mai simplă care merge	o ierarhie de 6 clase pentru o funcție de 10 rânduri
YAGNI (*You Aren't Gonna Need It*)	nu construi pentru nevoi imaginare	parametri și opțiuni pe care nu le cere nimeni
Legea lui Demeter	vorbește doar cu „vecinii"	`comanda.client().adresa().oras().cod()`
Tell, don't ask	spune obiectului ce să facă, nu-i cere datele ca să decizi tu	`if (cont.sold() >= x) cont.setSold(cont.sold() - x);` în loc de `cont.retrage(x)`
Încapsulare	datele private, interfața minimă	getter și setter pentru fiecare câmp

ATENȚIE — GREȘELI FRECVENTE

Principiile sunt **ghiduri**, nu legi. Aplicate mecanic produc cod complicat: zeci de interfețe cu o singură implementare, abstracții pentru lucruri care nu se vor schimba niciodată. Întrebarea corectă este mereu: „ce se va schimba probabil și cum fac ca schimbarea aceea să fie ușoară?".

7. Greșeli frecvente

Greșeala	Problema	Corect
moștenire doar pentru a refolosi cod	legătură strânsă, interfață expusă greșit	compunere
„clasa-zeu" (*God class*) care face tot	orice schimbare atinge aceeași clasă	împarte pe responsabilități (S)
`switch` pe tip în multe locuri	fiecare tip nou cere modificări peste tot	polimorfism (O)
derivată care aruncă „nu se poate"	încalcă substituția	reproiectează ierarhia (L)
interfață uriașă	implementări goale sau false	interfețe mici (I)
`new ClasaConcreta` adânc în logică	imposibil de testat sau înlocuit	injectează prin constructor (D)
getter + setter pentru tot	încapsulare doar de formă	metode care exprimă operații
abstracții „pentru orice eventualitate"	complexitate inutilă	YAGNI

8. În practică și la interviu

Întrebări frecvente: „explică SOLID cu exemple", „compunere sau moștenire?", „ce este dependency injection și de ce ajută la teste?", „ce încalcă `Patrat : Dreptunghi`?". În proiectele mari (de exemplu, software pentru telecomunicații sau automotive) aceste principii determină cât de ușor se adaugă o funcționalitate nouă fără regresii, iar testele unitare cu obiecte false sunt obligatorii.

9. Exersează pe hârtie

1. Clasifică relațiile: `Universitate` – `Facultate`, `Facultate` – `Profesor`, `Student` – `Persoana`, `Raport` – `Imprimanta` (primită ca parametru).

2. De ce `class Coada : public list<int>` este o idee proastă? Rescrie-o cu compunere.

3. O clasă `Factura` calculează totalul, generează PDF-ul și trimite e-mail. Ce principiu încalcă? Cum o împarți?

4. Un program are în cinci locuri `if (tip == "card") … else if (tip == "transfer") …`. Ce principiu încalcă? Cum adaugi „criptomonede" corect?

5. `class Strut : public Pasare` cu `void zboara() override { throw logic_error("nu zbor"); }`. Ce principiu este încălcat? Propune altă ierarhie.

6. O interfață `Angajat` are `lucreaza()`, `primesteSalariu()`, `conduceEchipa()`. Ce principiu încalcă pentru un stagiar?

7. Cum ai testa `ServiciuComenzi` dacă plata ar fi creată în interiorul lui (`PlataCard plata;` membru)? Ce schimbi?

8. Rescrie respectând „tell, don't ask": `if (stoc.cantitate(p) >= k) stoc.setCantitate(p, stoc.cantitate(p) - k);`.

9. Ce problemă are `a.getB().getC().getD().f()`?

10. Dă un exemplu în care aplicarea strictă a unui principiu face codul mai rău.

10. Mini-grile

1. „O clasă trebuie să aibă un singur motiv să se schimbe" este principiul: a) Open/Closed b) Single Responsibility c) Liskov d) Dependency Inversion

2. Compunerea înseamnă că: a) partea trăiește independent de întreg b) partea este creată și distrusă odată cu întregul c) clasa moștenește din parte d) clasa este abstractă

3. Injectarea dependențelor ajută mai ales la: a) viteză b) memorie c) testare și înlocuirea implementărilor d) compilare mai rapidă

4. Principiul Liskov este încălcat când: a) derivata adaugă metode noi b) derivata nu poate înlocui baza fără să surprindă codul existent c) baza este abstractă d) folosim `override`

11. Mai departe (pentru cine vrea mai mult)

- **Șabloane de proiectare** (lecțiile oop-08 și oop-09): soluții numite pentru probleme care se repetă.
- **Testare unitară** în C++: GoogleTest, Catch2; *mock*-uri cu GoogleMock.

- **Clean Architecture / arhitectura hexagonală:** DIP aplicat la nivelul întregii aplicații.
- **Coeziune și cuplare:** două măsuri ale calității unui modul (coeziune mare, cuplare mică).
- **Refactorizare:** schimbarea structurii fără schimbarea comportamentului, protejată de teste.

Rezumat

- Relații: dependență, asociere, agregare, compunere, moștenire; compunerea = partea moare cu întregul.
- Preferă compunerea; moștenirea doar pentru „este un” și interfețe polimorfice.
- **SOLID:** o responsabilitate; extinzi fără să modifici; derivata înlocuiește baza; interfețe mici; depinde de abstracții.
- Injectarea dependențelor: logica primește interfețe prin constructor ⇒ testare cu obiecte false.
- DRY, KISS, YAGNI, Demeter, „tell, don’t ask”: ghiduri, nu legi.

Citește și exersează în plus

- C++ Core Guidelines — Interfaces — reguli pentru interfețe bune (engleză) https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-interfaces
- Wikipedia — SOLID — originea principiilor și trimiteri (engleză) https://en.wikipedia.org/wiki/SOLID
- refactoring.guru — Design patterns — ilustrații clare pentru principii și șabloane de proiectare (engleză) https://refactoring.guru/design-patterns

Răspunsuri

Exersează pe hârtie

1. `Universitate` – `Facultate`: compunere (facultatea nu există fără universitate). `Facultate` – `Profesor`: agregare/asociere (profesorul există independent). `Student` – `Persoana`: moștenire. `Raport` – `Imprimanta` ca parametru: dependență.

2. Expune toată interfața listei (inserare la mijloc, ștergere oriunde), deci regula FIFO poate fi încălcată; în plus, `list` nu are destructor virtual. Corect: `class Coada { list<int> l; public: void push(int x) { l.push_back(x); } void pop() { l.pop_front(); } int front() const { return l.front(); } bool empty() const { return l.empty(); } };`.

3. Single Responsibility. `Factura` (date și total), `GeneratorPdf`, `ServiciuEmail`; un coordonator le folosește pe rând.

4. Open/Closed. Interfață `MetodaPlata` cu `incaseaza()`; fiecare metodă este o clasă; criptomonedele devin o clasă nouă, iar codul existent nu se modifică.

5. Liskov. Ierarhie: `Pasare` (fără `zboara`), `PasareZburatoare : Pasare` cu `zboara()`; `Strut : Pasare`, `Vultur : PasareZburatoare`. Sau interfață separată `Zburator`.

6. Interface Segregation: stagiarul ar trebui să implementeze `conduceEchipa()` deși nu are sens. Interfețe separate: `Lucrator`, `Platit`, `Lider`.

7. Nu se poate testa fără card real. Se primește plata ca `Plata&` (sau `unique_ptr<Plata>`) prin constructor; testul îi dă o implementare falsă.

8. `stoc.scoate(p, k);` iar `Stoc::scoate` verifică singur cantitatea și raportează eroarea (de exemplu, întoarce `false` sau aruncă excepție).

9. Încalcă legea lui Demeter: codul depinde de structura internă a trei clase; orice schimbare în lanț îl strică. Mai bine `a.f()`, care delegă intern.

10. De exemplu, o interfață și o fabrică pentru o funcție care calculează o medie: mai mult cod, mai greu de citit, fără niciun câștig, pentru că formula nu are de ce să varieze (YAGNI și KISS câștigă).

Mini-grile

1. b). **2.** b). **3.** c). **4.** b).