AVANSAT · POO ÎN C++ · LECȚIA 8

Șabloane de proiectare (I): creaționale și structurale

Durată: 110 min Nivel: avansat / pregătire pentru facultate și industrie Înainte: lecțiile oop-04, oop-05, oop-07

CE VEI ȘTI DUPĂ LECȚIE

- Explici ce este un șablon de proiectare și cele trei familii (creaționale, structurale, comportamentale).
- Implementezi Factory (cu registru), Prototype, Builder și Singleton și cunoști criticile aduse lui Singleton.
- Implementezi Adapter, Decorator și Composite și recunoști Facade și Proxy.
- Alegi un șablon pornind de la problemă, nu din reflex.

1. Ce este un șablon de proiectare?

Un **șablon de proiectare** (*design pattern*) este o soluție **cu nume** pentru o problemă de proiectare care apare iar și iar. Nu este o bibliotecă și nici cod de copiat, ci o **idee** despre cum să aranjezi clasele și obiectele. Cartea „Design Patterns" (1994, numită „Gang of Four", GoF) a descris 23 de șabloane, grupate în trei familii:

Familia	Rezolvă	Șabloane în lecțiile noastre
creaționale	cum se creează obiectele	Factory, Prototype, Builder, Singleton (aici)
structurale	cum se compun clasele și obiectele	Adapter, Decorator, Composite, Facade, Proxy (aici)
comportamentale	cum comunică și își împart responsabilitățile	Strategy, Observer, Command, State, Template Method, Iterator (oop-09)

DE CE AȘA?

De ce contează numele? Un vocabular comun scurtează discuțiile: „pune un Adapter peste biblioteca veche" spune într-o propoziție o structură întreagă. La interviuri și în revizuirile de cod, șabloanele sunt limbajul comun al proiectării. **Atenție însă:** un șablon folosit fără problema pe care o rezolvă doar complică codul (KISS, YAGNI din oop-05).

2. Factory: cine creează obiectele?

Problema. Codul clientului citește din fișier „P 3" sau „D 2 5" și trebuie să creeze obiectul potrivit. Dacă scrie peste tot `if (tip == "P") new Patrat …`, fiecare formă nouă cere modificări în toate locurile (încalcă Open/Closed).

Soluția. Creația se mută într-un singur loc, o **fabrică**: clientul cere „un obiect de tipul X” și primește un `unique_ptr<Forma>`, fără să știe clasa concretă. Varianta cu **registru** (un `map` de la nume la funcția care creează) permite adăugarea unui tip nou printr-o singură linie de înregistrare.

Prototype este înrudit: fiecare obiect știe să se **cloneze** (`clonare()` virtuală), deci poți copia un obiect fără să-i știi tipul exact.

```cpp
#include <iostream>
#include <memory>
#include <map>
#include <string>
#include <functional>
#include <vector>
using namespace std;

class Forma
{
public:
    virtual void citeste(istream &in) = 0;
    virtual double arie() const = 0;
    virtual string nume() const = 0;
    virtual unique_ptr<Forma> clonare() const = 0;     // Prototype: copie polimorfică
    virtual ~Forma() = default;
};

class Patrat : public Forma
{
    double l = 0;
public:
    void citeste(istream &in) override { in >> l; }
    double arie() const override { return l * l; }
    string nume() const override { return "patrat"; }
    unique_ptr<Forma> clonare() const override { return make_unique<Patrat>(*this); }
};

class Dreptunghi : public Forma
{
    double a = 0, b = 0;
public:
    void citeste(istream &in) override { in >> a >> b; }
    double arie() const override { return a * b; }
    string nume() const override { return "dreptunghi"; }
    unique_ptr<Forma> clonare() const override { return make_unique<Dreptunghi>(*this); }
};

class FabricaForme                          // Factory cu registru: tip nou = o linie de
înregistrare
{
    map<string, function<unique_ptr<Forma>()>> creatori;
public:
    void inregistreaza(const string &tip, function<unique_ptr<Forma>()> f) { creatori[tip]
= std::move(f); }
    unique_ptr<Forma> creeaza(const string &tip) const
    {
        auto it = creatori.find(tip);
        if (it == creatori.end()) return nullptr;
        return it->second();
    }
};

int main()
{
    FabricaForme fabrica;
    fabrica.inregistreaza("P", [] { return make_unique<Patrat>(); });
```

```
    fabrica.inregistreaza("D", [] { return make_unique<Dreptunghi>(); });

    int n;
    cin >> n;
    vector<unique_ptr<Forma>> forme;
    for (int i = 0; i < n; i++)
    {
        string tip;
        cin >> tip;
        auto f = fabrica.creeaza(tip);     // codul clientului nu scrie niciodată new
Patrat
        if (!f) { cout << "tip necunoscut: " << tip << '\n'; continue; }
        f->citeste(cin);
        forme.push_back(std::move(f));
    }
    for (const auto &f : forme) cout << f->nume() << ' ' << f->arie() << '\n';
    if (!forme.empty())
    {
        auto copie = forme[0]->clonare();  // copiem fără să știm tipul exact
        cout << "clona: " << copie->nume() << ' ' << copie->arie() << '\n';
    }
    return 0;
}
```

DE ȘTIUT

GoF descrie două variante înrudite. **Factory Method:** o metodă virtuală `creeaza()` într-o clasă de bază, iar fiecare derivată decide ce produs concret creează. **Abstract Factory:** o interfață care creează **familii** de obiecte compatibile (de exemplu, butoane, ferestre și meniuri pentru Windows sau pentru Linux). Fabrica cu registru de mai sus este varianta cea mai des întâlnită în practică.

3. Builder: construirea pas cu pas

Problema. Un obiect are multe câmpuri, unele opționale: un constructor cu 8 parametri (`Pizza(32, "subtire", "", true, false, …)`) este greu de citit și ușor de greșit.

Soluția. Un obiect separat, **builder**-ul, primește valorile prin metode cu nume, înlănțuite (`return *this`, lecția oop-01), și la final `construieste()` **verifică** totul și întoarce obiectul gata făcut.

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

class Pizza                                  // obiectul complex: se construiește doar prin
Builder
{
    friend class PizzaBuilder;
    string blat, sos;
    vector<string> toppinguri;
    int diametru = 0;
    Pizza() = default;
public:
    string descriere() const
    {
        string s = to_string(diametru) + "cm, blat " + blat + ", sos " + sos;
        for (const auto &t : toppinguri) s += ", " + t;
        return s;
    }
};

class PizzaBuilder                           // Builder: pași cu nume, în orice ordine,
verificare la final
{
    Pizza p;
public:
    PizzaBuilder& diametru(int d) { p.diametru = d; return *this; }
    PizzaBuilder& blat(const string &b) { p.blat = b; return *this; }
    PizzaBuilder& sos(const string &s) { p.sos = s; return *this; }
    PizzaBuilder& adauga(const string &t) { p.toppinguri.push_back(t); return *this; }
    Pizza construieste() const
    {
        if (p.diametru < 20 || p.diametru > 50) throw invalid_argument("diametru invalid");
        if (p.blat.empty()) throw invalid_argument("lipseste blatul");
        Pizza rez = p;
        if (rez.sos.empty()) rez.sos = "rosii";   // valoare implicită
        return rez;
    }
};

int main()
{
    Pizza a =
PizzaBuilder().blat("subtire").diametru(32).adauga("mozzarella").adauga("ciuperci").construie
    cout << a.descriere() << '\n';
    try
    {
        Pizza b = PizzaBuilder().diametru(60).blat("pufos").construieste();
        cout << b.descriere() << '\n';
    }
    catch (const exception &e) { cout << "eroare: " << e.what() << '\n'; }
    return 0;
}
```

SFAT

Builder-ul se folosește des pentru cereri de rețea, configurări, interogări SQL, mesaje de test. Avantajul principal: obiectul final poate fi **imutabil** și mereu valid (verificările sunt într-un singur loc).

4. Singleton: o singură instanță

Problema. Unele obiecte trebuie să existe o singură dată (jurnalul aplicației, configurarea globală) și trebuie să fie accesibile de oriunde.

Soluția. Constructorul este privat, copierea interzisă, iar o metodă statică întoarce mereu aceeași instanță. În C++11 și mai nou, varianta **Meyers** (o variabilă `static` locală) este sigură și cu mai multe fire de execuție.

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

class Jurnal                                // Singleton (varianta Meyers)
{
    int numar = 0;
    Jurnal() { cout << "jurnal deschis\n"; }         // constructor privat: nimeni nu
creează altul
public:
    Jurnal(const Jurnal&) = delete;
    Jurnal& operator=(const Jurnal&) = delete;
    static Jurnal& instanta()
    {
        static Jurnal j;                    // creat la primul apel; sigur și cu mai multe
fire (C++11)
        return j;
    }
    void scrie(const string &mesaj) { cout << ++numar << ": " << mesaj << '\n'; }
};

void modulA() { Jurnal::instanta().scrie("pornit modulul A"); }
void modulB() { Jurnal::instanta().scrie("pornit modulul B"); }

int main()
{
    cout << "start\n";
    modulA();
    modulB();
    Jurnal::instanta().scrie("gata");
    cout << (&Jurnal::instanta() == &Jurnal::instanta() ? "acelasi obiect" : "obiecte
diferite") << '\n';
    return 0;
}
```

ATENȚIE — GREȘELI FRECVENTE

Singleton este cel mai **criticat** șablon: este o variabilă globală deghizată. Ascunde dependențele (din semnătura lui `modulA()` nu vezi că folosește jurnalul), face testele dificile (nu poți înlocui jurnalul cu unul fals) și creează legături între module. Alternativa preferată este **injectarea dependențelor** (oop-05): creezi un singur jurnal în `main` și îl dai prin constructor celor care au nevoie. Folosește Singleton doar pentru resurse cu adevărat unice și fără stare importantă pentru logică.

5. Adapter: interfețe incompatibile

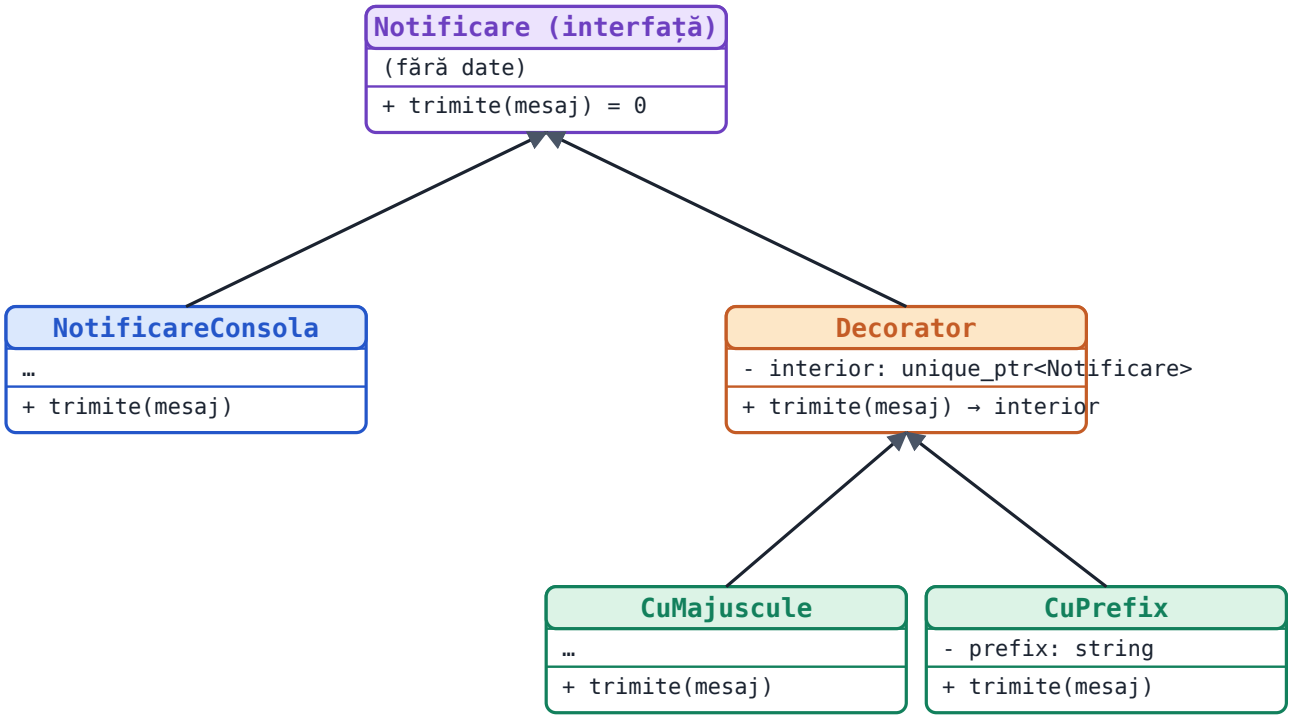
Problema. Aplicația ta folosește interfața `Notificare::trimite(string)`. Ai primit o bibliotecă veche de SMS cu funcția `sendText(const char*, const char*)`, pe care nu o poți modifica.

Soluția. O clasă **adaptor** implementează interfața ta și, în interior, apelează biblioteca veche, „traducând" apelurile. Ca adaptorul de priză: nu schimbi nici aparatul, nici priza.

6. Decorator: comportament adăugat în straturi

Problema. Vrei notificări cu majuscule, cu prefix, cu ambele, pe consolă sau prin SMS. Cu moștenire ar trebui câte o clasă pentru fiecare combinație (`SmsCuPrefixSiMajuscule`...): explozie de clase.

Soluția. Un **decorator** implementează aceeași interfață și **conține** un obiect de același tip. Face ceva în plus, apoi pasează apelul mai departe. Decoratorii se pot înfășura oricât, în orice ordine, la rulare.



un decorator ESTE o Notificare și ARE o Notificare: adaugă ceva, apoi pasează mai departe

Decorator

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

class Notificare                          // interfața pe care o folosește aplicația
{
public:
    virtual void trimite(const string &mesaj) = 0;
    virtual ~Notificare() = default;
};

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

// --- Adapter: o bibliotecă veche, cu altă interfață, pe care nu o putem modifica ---
class SmsVechi
{
public:
    void sendText(const char *numar, const char *text) { cout << "SMS catre " << numar <<
": " << text << '\n'; }
};

class AdaptorSms : public Notificare      // „traduce” trimite(...) în sendText(...)
{
    SmsVechi sms;
    string numar;
public:
    explicit AdaptorSms(string n) : numar(std::move(n)) {}
    void trimite(const string &mesaj) override { sms.sendText(numar.c_str(),
mesaj.c_str()); }
};

// --- Decorator: adaugă comportament fără să modifice clasele existente ---
class Decorator : public Notificare
{
protected:
    unique_ptr<Notificare> interior;
public:
    explicit Decorator(unique_ptr<Notificare> i) : interior(std::move(i)) {}
    void trimite(const string &mesaj) override { interior->trimite(mesaj); }
};

class CuMajuscule : public Decorator
{
public:
    using Decorator::Decorator;           // moștenim constructorul
    void trimite(const string &mesaj) override
    {
        string m = mesaj;
        for (char &c : m)
            if (c >= 'a' && c <= 'z') c = c - 'a' + 'A';
        interior->trimite(m);
    }
```

```
};

class CuPrefix : public Decorator
{
    string prefix;
public:
    CuPrefix(unique_ptr<Notificare> i, string p) : Decorator(std::move(i)),
prefix(std::move(p)) {}
    void trimite(const string &mesaj) override { interior->trimite("[" + prefix + "] " +
mesaj); }
};

int main()
{
    string mesaj;
    getline(cin, mesaj);

    unique_ptr<Notificare> simpla = make_unique<NotificareConsola>();
    simpla->trimite(mesaj);

    unique_ptr<Notificare> sms = make_unique<AdaptorSms>("0722000000");
    sms->trimite(mesaj);

    // decoratorii se înfășoară: cel din exterior lucrează primul (majuscule), apoi
prefixul, apoi consola
    unique_ptr<Notificare> urgent = make_unique<CuMajuscule>(
        make_unique<CuPrefix>(make_unique<NotificareConsola>(), "urgent"));
    urgent->trimite(mesaj);

    unique_ptr<Notificare> smsUrgent = make_unique<CuPrefix>(make_unique<AdaptorSms>
("0733111111"), "alerta");
    smsUrgent->trimite(mesaj);
    return 0;
}
```

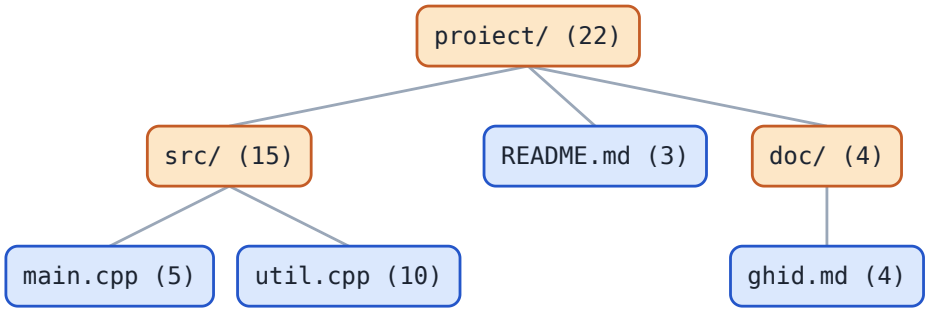
DE CE AȘA?

De ce `[urgent]` apare cu litere mici? Lanțul este `CuMajuscule(CuPrefix(Consola))`. Decoratorul din **exterior** primește apelul primul: transformă mesajul în majuscule, apoi îl dă lui `CuPrefix`, care adaugă `[urgent]` (deja după transformare), apoi consolei. Ordinea înfășurării contează. **Unde se folosește în realitate?** Fluxurile de date (comprimare peste criptare peste fișier), middleware-urile serverelor web, adăugarea de jurnalizare sau cache peste un serviciu.

7. Composite: părți și întreguri tratate la fel

Problema. Un folder conține fișiere **și** alte foldere. Vrei să calculezi mărimea oricărui nod fără `if (este folder) … else …` peste tot.

Soluția. Frunzele (`Fisier`) și grupurile (`Folder`) implementează aceeași interfață (`Nod`). Un grup își delegă operațiile copiilor și combină rezultatele. Clientul tratează un singur fișier și un arbore întreg exact la fel.



mărime(folder) = suma mărimilor copiilor; clientul apelează mărime() fără să știe ce tip are nodul

Composite

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

class Nod                                   // Composite: interfața comună pentru frunze și grupuri
{
protected:
    string nume;
public:
    explicit Nod(string n) : nume(std::move(n)) {}
    virtual long long marime() const = 0;
    virtual void afiseaza(int nivel) const = 0;
    virtual ~Nod() = default;
};

class Fisier : public Nod
{
    long long octeti;
public:
    Fisier(string n, long long o) : Nod(std::move(n)), octeti(o) {}
    long long marime() const override { return octeti; }
    void afiseaza(int nivel) const override
    {
        cout << string(2 * nivel, ' ') << nume << " (" << octeti << ")\n";
    }
};

class Folder : public Nod
{
    vector<unique_ptr<Nod>> copii;          // copiii pot fi fișiere SAU alte foldere
public:
    explicit Folder(string n) : Nod(std::move(n)) {}
    Folder& adauga(unique_ptr<Nod> c) { copii.push_back(std::move(c)); return *this; }
    long long marime() const override
    {
        long long s = 0;
        for (const auto &c : copii) s += c->marime();      // recursiv, fără să știm tipul copilului
        return s;
    }
    void afiseaza(int nivel) const override
    {
        cout << string(2 * nivel, ' ') << nume << "/ (" << marime() << ")\n";
        for (const auto &c : copii) c->afiseaza(nivel + 1);
    }
};

int main()
{
    auto src = make_unique<Folder>("src");
    src->adauga(make_unique<Fisier>("main.cpp", 5)).adauga(make_unique<Fisier>("util.cpp", 10));
    auto doc = make_unique<Folder>("doc");
    doc->adauga(make_unique<Fisier>("ghid.md", 4));
```

```
    Folder proiect("proiect");
    proiect.adauga(std::move(src)).adauga(make_unique<Fisier>("README.md",
3)).adauga(std::move(doc));
    proiect.afiseaza(0);
    cout << "total: " << proiect.marime() << '\n';
    return 0;
}
```

DE ȘTIUT

Composite apare oriunde există ierarhii: elementele unei interfețe grafice (ferestre care conțin panouri care conțin butoane), arborii de expresii (`(a + b) * c`), meniurile cu submeniuri, documentele HTML (DOM).

8. Încă două structurale, pe scurt

Șablonul	Ideea	Exemplu
Facade	o clasă simplă care ascunde un subsistem complicat	`Calculator::porneste()` care inițializează discul, rețeaua și ecranul în ordinea corectă
Proxy	un obiect cu aceeași interfață care controlează accesul la cel real	încărcare leneșă a unei imagini mari, verificarea drepturilor, cache, obiect aflat pe alt server; și smart pointers sunt un fel de proxy

9. Cum alegi?

Dacă problema este...	Încearcă
„obiectul concret depinde de date citite la rulare”	Factory (cu registru)
„trebuie să copiez obiecte polimorfice”	Prototype (`clonare()`)
„constructor cu prea mulți parametri, unii opționali”	Builder
„o singură instanță, accesibilă global”	mai întâi injectare; dacă nu, Singleton
„interfața pe care o am nu se potrivește cu cea cerută”	Adapter
„vreau combinații de funcționalități adăugate”	Decorator
„arbore de părți și întreguri tratate uniform”	Composite
„subsistem complicat, vreau o intrare simplă”	Facade
„controlez accesul la un obiect costisitor sau îndepărtat”	Proxy

10. Greșeli frecvente

Greșeala	Problema	Corect
șabloane „ca să fie”	complexitate fără beneficiu	doar când există problema
Singleton pentru orice serviciu	dependențe ascunse, teste grele	injectarea dependențelor
fabrică cu `switch` mare pe tip	trebuie modificată la fiecare tip	registru `map<string, function<…>>`
decorator care nu pasează apelul mai departe	obiectul interior nu mai lucrează	apelează `interior->metoda()`
Composite cu metode de frunză care aruncă „nu se poate”	încalcă Liskov	interfață comună minimă
Builder fără validare la `construieste()`	obiecte invalide	verifică totul la final
`clonare()` care întoarce `Forma` prin valoare	slicing	`unique_ptr<Forma>`

11. Exersează pe hârtie

1. Pentru fiecare situație, numește șablonul: (a) „o bibliotecă de grafice cere `Punct2D`, dar noi avem `Coordonate`”; (b) „o cafea cu lapte, frișcă și scorțișoară, în orice combinație”; (c) „meniuri cu submeniuri”; (d) „un obiect `Raport` cu 12 câmpuri opționale”.

2. Ce trebuie modificat în programul cu fabrica pentru a adăuga `Cerc`? Ce **nu** trebuie modificat?

3. De ce constructorul lui `Jurnal` este privat? Ce s-ar întâmpla cu `Jurnal j2 = Jurnal::instanta();` fără `= delete`?

4. Explică de ce Singleton îngreunează testarea, cu un exemplu.

5. Ce afișează lanțul `CuPrefix(CuMajuscule(Consola), "info")` pentru mesajul „ok”?

6. Care este diferența dintre Adapter și Decorator, dacă ambele „înfășoară” un obiect?

7. În Composite, de ce `Folder::marime()` nu trebuie să știe dacă un copil este fișier sau folder?

8. Ce câștigi cu Builder față de un constructor cu valori implicite?

9. De ce `clonare()` trebuie să fie virtuală? Ce s-ar întâmpla cu `make_unique<Forma>(*f)`?

10. Dă un exemplu de Facade din viața de zi cu zi sau dintr-o bibliotecă pe care o cunoști.

12. Mini-grile

1. Singleton face parte din familia: a) structurală b) creațională c) comportamentală d) niciuna

2. Decorator adaugă comportament: a) prin moștenire pentru fiecare combinație b) înfășurând un obiect cu aceeași interfață c) modificând clasa originală d) prin variabile globale

3. Adapter se folosește când: a) vrei o singură instanță b) o interfață existentă nu se potrivește cu cea așteptată c) construiești un arbore d) copiezi obiecte

4. Composite permite: a) tratarea uniformă a părților și a întregurilor b) crearea unui singur obiect c) conversia între interfețe d) evitarea moștenirii

13. Mai departe (pentru cine vrea mai mult)

- **Abstract Factory** și **Factory Method** în forma lor completă.
- **Flyweight:** partajarea stării comune între multe obiecte mici (de exemplu, caracterele unui editor).
- **Bridge:** separă abstracția de implementare, ca să varieze independent.
- **Pimpl:** varianta C++ a ascunderii implementării (compilare mai rapidă, interfață binară stabilă).
- **Șabloane comportamentale:** lecția următoare.

Rezumat

- Un șablon de proiectare = o soluție cu nume pentru o problemă recurentă; trei familii.
- **Factory** centralizează crearea (registru = tip nou fără modificări); **Prototype** clonează polimorfic.
- **Builder** construiește pas cu pas și validează la final; **Singleton** = o instanță, dar preferă injectarea.
- **Adapter** traduce o interfață în alta; **Decorator** adaugă comportament în straturi; **Composite** tratează uniform arborii.
- **Facade** simplifică un subsistem; **Proxy** controlează accesul.
- Folosește un șablon doar când există problema pe care o rezolvă.

Citește și exersează în plus

- refactoring.guru — Design Patterns — toate cele 23 de șabloane clasice, cu ilustrații și exemple în C++ (engleză) https://refactoring.guru/design-patterns
- refactoring.guru — Design Patterns in C++ — exemple de cod C++ pentru fiecare șablon (engleză) https://refactoring.guru/design-patterns/cpp
- Wikipedia — Design Patterns (cartea GoF) — cartea din 1994 care a numit cele 23 de șabloane (engleză) https://en.wikipedia.org/wiki/Design_Patterns

Răspunsuri

Exersează pe hârtie

1. (a) Adapter; (b) Decorator; (c) Composite; (d) Builder.

2. Se scrie clasa `Cerc` și se adaugă o linie `fabrica.inregistreaza("C", [] { return make_unique<Cerc>(); });`. Nu se modifică bucla de citire, afișarea sau clasa fabricii.

3. Ca nimeni să nu poată crea alt `Jurnal`. Fără `= delete` la copiere, `j2` ar fi o **copie**, adică a doua instanță, cu propriul contor: s-ar încălca ideea de unicitate.

4. Dacă `ServiciuComenzi` scrie direct în `Jurnal::instanta()`, într-un test nu poți înlocui jurnalul cu unul care doar reține mesajele ca să le verifici; testele se influențează între ele prin contorul comun.

5. `[INFO] OK`: întâi prefixul (exterior), apoi majusculele transformă tot șirul, inclusiv prefixul.

6. Adapter **schimbă** interfața (din cea veche în cea cerută). Decorator **păstrează** interfața și adaugă comportament.

7. Pentru că apelează `marime()` prin interfața comună `Nod`; fiecare copil își calculează singur mărimea (polimorfism + recursivitate).

8. Pașii au nume (cod ușor de citit), ordinea nu contează, valorile opționale se omit fără a ține minte poziții, iar validarea completă se face într-un singur loc, la final.

9. Ca să fie apelată versiunea tipului real. `make_unique<Forma>(*f)` nici nu se compilează (`Forma` este abstractă); dacă nu ar fi abstractă, ar face slicing.

10. De exemplu, un buton „pornește” al unei mașini de spălat (ascunde pompa, motorul, încălzirea) sau funcția `std::sort`, care ascunde un algoritm hibrid complicat.

Mini-grile

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