AVANSAT · POO ÎN C++ · LECȚIA 9

Șabloane de proiectare (II): comportamentale — Strategy, Observer, Command, State, Template Method, Iterator

Durată: 110 min Nivel: avansat / pregătire pentru facultate și industrie Înainte: lecțiile oop-07, oop-08

CE VEI ȘTI DUPĂ LECȚIE

- Implementezi Strategy cu ierarhie de clase și cu std::function, și știi când alegi fiecare variantă.
- Implementezi Observer cu abonare și dezabonare.
- Implementezi Command cu istoric pentru undo și redo.
- Modelezi un automat cu stări prin șablonul State.
- Folosești Template Method și scrii un iterator propriu, compatibil cu range-for.

1. Cum comunică obiectele

Șabloanele **comportamentale** descriu cum își împart obiectele responsabilitățile și cum comunică, astfel încât să poți schimba un comportament **fără** să modifici restul codului. Aproape toate se sprijină pe aceeași idee: **un comportament devine un obiect** (o strategie, o comandă, o stare, un abonat), care se poate păstra, înlocui și combina la rulare.

Șablonul	Ideea într-o propoziție
Strategy	un algoritm interschimbabil, ales la rulare
Observer	când ceva se schimbă, toți cei interesați sunt anunțați
Command	o acțiune ca obiect: o poți păstra, repeta, anula
State	comportamentul depinde de starea curentă, iar fiecare stare este o clasă
Template Method	scheletul algoritmului e fix, pașii se completează în derivate
Iterator	parcurgi o colecție fără să știi cum este construită

2. Strategy: algoritmi interschimbabili

Problema. Un coș de cumpărături aplică reduceri diferite (niciuna, procent, prag fix, „Black Friday”). Cu `if` / `switch` în `total()`, fiecare reducere nouă modifică clasa `Cos`.

Soluția. Fiecare algoritm este o clasă cu aceeași interfață; contextul (`Cos`) ține o strategie și o poate schimba la rulare.

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

class StrategieReducere                    // familia de algoritmi interschimbabili
{
public:
    virtual double aplica(double total) const = 0;
    virtual string nume() const = 0;
    virtual ~StrategieReducere() = default;
};

class FaraReducere : public StrategieReducere
{
public:
    double aplica(double total) const override { return total; }
    string nume() const override { return "fara reducere"; }
};

class Procent : public StrategieReducere
{
    double p;
public:
    explicit Procent(double x) : p(x) {}
    double aplica(double total) const override { return total * (100 - p) / 100; }
    string nume() const override { return to_string((int)p) + "% reducere"; }
};

class PragFix : public StrategieReducere   // „minus s lei la peste prag lei”
{
    double prag, s;
public:
    PragFix(double pr, double x) : prag(pr), s(x) {}
    double aplica(double total) const override { return total >= prag ? total - s : total;
}
    string nume() const override { return "minus " + to_string((int)s) + " peste " +
to_string((int)prag); }
};

class Cos                                  // contextul: folosește o strategie, oricare
{
    vector<double> preturi;
    unique_ptr<StrategieReducere> strategie = make_unique<FaraReducere>();
public:
    void adauga(double p) { preturi.push_back(p); }
    void setStrategie(unique_ptr<StrategieReducere> s) { strategie = std::move(s); }   //
schimbare la rulare
    double total() const
    {
        double t = 0;
        for (double p : preturi) t += p;
        return strategie->aplica(t);
    }
    string descriere() const { return strategie->nume(); }
};
```

```
int main()
{
    int n;
    cin >> n;
    Cos c;
    for (int i = 0; i < n; i++) { double p; cin >> p; c.adauga(p); }
    cout << c.descriere() << ": " << c.total() << '\n';
    c.setStrategie(make_unique<Procent>(10));
    cout << c.descriere() << ": " << c.total() << '\n';
    c.setStrategie(make_unique<PragFix>(200, 50));
    cout << c.descriere() << ": " << c.total() << '\n';
    return 0;
}
```

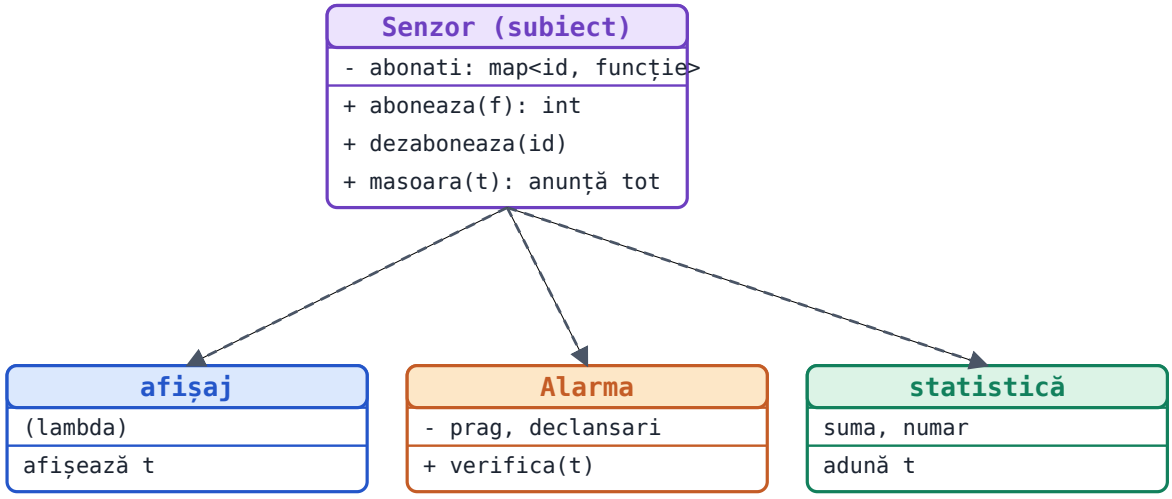
SFAT

Varianta modernă: dacă strategia este o singură funcție fără stare complicată, folosește `std::function<double(double)>` și lambda: `cos.setReducere([](double t) { return t * 0.9; });`. Clasele se justifică atunci când strategia are mai multe metode (aici `aplica` și `nume`) sau stare proprie. Un exemplu de Strategy pe care îl folosești de mult: comparatorul dat lui `sort`.

3. Observer: abonare la evenimente

Problema. Un senzor produce valori; afișajul, alarma și modulul de statistici trebuie anunțate la fiecare valoare. Senzorul nu trebuie să-i cunoască (altfel, la fiecare modul nou, modifici senzorul).

Soluția. Subiectul ține o **listă de abonați**; la fiecare schimbare îi anunță pe toți. Abonații se pot adăuga și scoate oricând.



subiectul nu cunoaște tipurile observatorilor: are doar o listă de funcții de apelat

Observer

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

class Senzor                                  // subiectul observat
{
    map<int, function<void(double)>> abonati;   // id -> funcția de anunțat
    int urmatorId = 0;
public:
    int aboneaza(function<void(double)> f) { abonati[urmatorId] = std::move(f); return
urmatorId++; }
    void dezaboneaza(int id) { abonati.erase(id); }
    void masoara(double t)                    // la fiecare valoare nouă, anunță toți abonații
    {
        for (auto &[id, f] : abonati) f(t);
    }
};

class Alarma                                  // un observator cu stare proprie
{
    double prag;
    int declansari = 0;
public:
    explicit Alarma(double p) : prag(p) {}
    void verifica(double t)
    {
        if (t > prag) { declansari++; cout << "ALARMA: " << t << " > " << prag << '\n'; }
    }
    int total() const { return declansari; }
};

int main()
{
    Senzor s;
    Alarma alarma(30);
    double suma = 0;
    int numar = 0;

    int afisaj = s.aboneaza([](double t) { cout << "afisaj: " << t << '\n'; });
    s.aboneaza([&alarma](double t) { alarma.verifica(t); });
    s.aboneaza([&](double t) { suma += t; numar++; });       // statistică

    int n;
    cin >> n;
    for (int i = 0; i < n; i++)
    {
        double t;
        cin >> t;
        if (i == 2) s.dezaboneaza(afisaj);                  // afișajul nu mai vrea
notificări
        s.masoara(t);
    }
    cout << "medie: " << suma / numar << ", alarme: " << alarma.total() << '\n';
```

```
    return 0;
}
```

ATENȚIE — GREȘELI FRECVENTE

Capcanele Observer-ului: (1) un abonat **mort** rămas în listă (lambda cu `[&]` spre un obiect distrus): dezabonează-l în destructor sau folosește `weak_ptr`; (2) **modificarea listei în timpul notificării** (un abonat care se dezabonează din propriul callback invalidează iteratorul): copiază lista înainte sau amână ștergerile; (3) ordinea notificărilor nu ar trebui să conteze. Șablonul stă la baza interfețelor grafice (evenimente de clic), a sistemelor de mesaje (*publish/subscribe*) și a „signals & slots” din Qt.

4. Command: acțiuni ca obiecte, cu undo

Problema. Un editor de text trebuie să poată anula (*undo*) și reface (*redo*) orice operație.

Soluția. Fiecare operație este un obiect cu `executa()` și `anuleaza()`, care ține minte ce trebuie pentru anulare. Editorul execută comenzile și le pune într-o **stivă** de istoric (lecția XI.17).

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

class Document                              // receptorul: obiectul asupra căruia lucrează comenzile
{
public:
    string text;
};

class Comanda                               // o acțiune ca obiect: se poate executa și anula
{
public:
    virtual void executa() = 0;
    virtual void anuleaza() = 0;
    virtual ~Comanda() = default;
};

class Adauga : public Comanda
{
    Document &doc;
    string bucata;
public:
    Adauga(Document &d, string b) : doc(d), bucata(std::move(b)) {}
    void executa() override { doc.text += bucata; }
    void anuleaza() override { doc.text.erase(doc.text.size() - bucata.size()); }
};

class Sterge : public Comanda               // șterge ultimele k caractere, dar ține minte ce a șters
{
    Document &doc;
    size_t k;
    string sters;
public:
    Sterge(Document &d, size_t n) : doc(d), k(n) {}
    void executa() override
    {
        k = min(k, doc.text.size());
        sters = doc.text.substr(doc.text.size() - k);
        doc.text.erase(doc.text.size() - k);
    }
    void anuleaza() override { doc.text += sters; }
};

class Editor                                // invocatorul: execută comenzi și ține istoricul
{
    Document doc;
    stack<unique_ptr<Comanda>> istoric, refaceri;
public:
    void ruleaza(unique_ptr<Comanda> c)
    {
```

```
        c->executa();
        istoric.push(std::move(c));
        refaceri = {};                    // o acțiune nouă golește lista de „refă”
    }
    void undo()
    {
        if (istoric.empty()) return;
        istoric.top()->anuleaza();
        refaceri.push(std::move(istoric.top()));
        istoric.pop();
    }
    void redo()
    {
        if (refaceri.empty()) return;
        refaceri.top()->executa();
        istoric.push(std::move(refaceri.top()));
        refaceri.pop();
    }
    Document& document() { return doc; }
};

int main()
{
    Editor e;
    string cmd;
    while (cin >> cmd)
    {
        if (cmd == "add") { string s; cin >> s; e.ruleaza(make_unique<Adauga>(e.document(),
s)); }
        else if (cmd == "del") { size_t k; cin >> k; e.ruleaza(make_unique<Sterge>
(e.document(), k)); }
        else if (cmd == "undo") e.undo();
        else if (cmd == "redo") e.redo();
        else if (cmd == "print") cout << '"' << e.document().text << "\"\n";
    }
    return 0;
}
```

Comanda	Text după	Istoric (vârf la dreapta)	Refaceri
`add ana`	ana	Adauga(ana)	–
`add are`	anaare	+Adauga(are)	–
`del 4`	an	+Sterge(4, „aare”)	–
`undo`	anaare	Adauga(ana), Adauga(are)	Sterge
`undo`	ana	Adauga(ana)	Sterge, Adauga(are)
`redo`	anaare	Adauga(ana), Adauga(are)	Sterge
`add mere`	anaaremere	+Adauga(mere)	golită
`redo`	neschimbat	–	nimic de refăcut

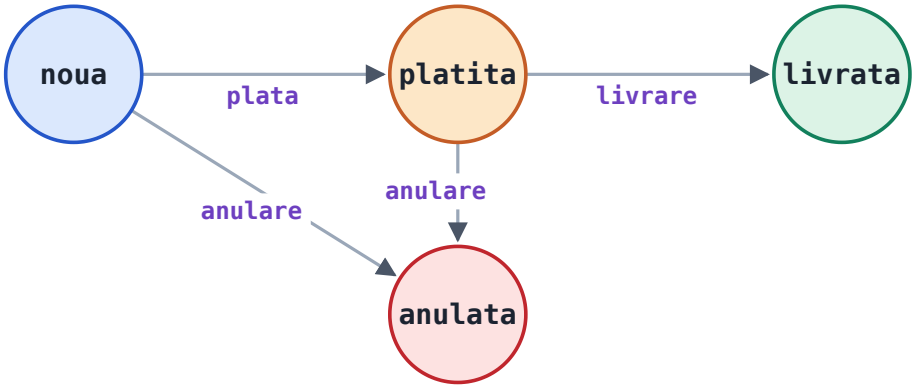
DE CE AȘA?

De ce o acțiune nouă golește refacerile? După un `undo` și o acțiune nouă, „viitorul” vechi nu mai are sens (ar trebui aplicat peste alt text). Așa se comportă toate editoarele. **Unde mai apare Command?** Cozi de sarcini și thread pool-uri (o sarcină = un obiect), macro-uri (o listă de comenzi rejucată), tranzacții, butoane și meniuri care declanșează aceeași acțiune.

5. State: automat cu stări

Problema. O comandă online trece prin stările *nouă* → *plătită* → *livrată* (sau *anulată*). Fiecare acțiune are alt efect în funcție de stare. Cu `if (stare == …)` în fiecare metodă, logica se împrăștie și se strică ușor.

Soluția. Fiecare stare este o clasă care decide ce înseamnă fiecare acțiune și **în ce stare se trece**. Contextul doar deleagă.



orice acțiune care nu are săgeată din starea curentă este refuzată de acea stare

Stările unei comenzi

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

class Stare                                  // fiecare stare decide ce înseamnă fiecare
acțiune
{
public:
    virtual string nume() const = 0;
    virtual unique_ptr<Stare> plateste() const { cout << "  nu se poate plati in starea "
<< nume() << '\n'; return nullptr; }
    virtual unique_ptr<Stare> livreaza() const { cout << "  nu se poate livra in starea "
<< nume() << '\n'; return nullptr; }
    virtual unique_ptr<Stare> anuleaza() const { cout << "  nu se poate anula in starea "
<< nume() << '\n'; return nullptr; }
    virtual ~Stare() = default;
};

struct Anulata : Stare { string nume() const override { return "anulata"; } };
struct Livrata : Stare { string nume() const override { return "livrata"; } };

struct Platita : Stare
{
    string nume() const override { return "platita"; }
    unique_ptr<Stare> livreaza() const override { return make_unique<Livrata>(); }
    unique_ptr<Stare> anuleaza() const override { cout << "  se returneaza banii\n"; return
make_unique<Anulata>(); }
};

struct Noua : Stare
{
    string nume() const override { return "noua"; }
    unique_ptr<Stare> plateste() const override { return make_unique<Platita>(); }
    unique_ptr<Stare> anuleaza() const override { return make_unique<Anulata>(); }
};

class Comanda                                // contextul: deleagă acțiunile stării curente
{
    unique_ptr<Stare> stare = make_unique<Noua>();
    void treci(unique_ptr<Stare> noua) { if (noua) stare = std::move(noua); }
public:
    void plateste() { treci(stare->plateste()); }
    void livreaza() { treci(stare->livreaza()); }
    void anuleaza() { treci(stare->anuleaza()); }
    string stareCurenta() const { return stare->nume(); }
};

int main()
{
    Comanda c;
    string actiune;
    while (cin >> actiune)
    {
        cout << actiune << ":\n";
        if (actiune == "plata") c.plateste();
        else if (actiune == "livrare") c.livreaza();
```

```
        else if (actiune == "anulare") c.anuleaza();
        cout << "  stare: " << c.stareCurenta() << '\n';
    }
    return 0;
}
```

DE ȘTIUT

Baza `Stare` refuză implicit orice acțiune; fiecare stare suprascrie doar tranzițiile permise. O stare nouă (de exemplu, „în curs de livrare") înseamnă o clasă nouă și o modificare a tranzițiilor din starea anterioară, nu `if`-uri noi în toate metodele. Pentru automate mici, un `enum` cu un tabel de tranziții sau `std::variant` sunt alternative mai scurte.

6. Template Method și Iterator

Template Method. Baza fixează **ordinea** pașilor într-o metodă **nevirtuală**, iar pașii sunt metode virtuale (eventual cu variantă implicită). Derivatele schimbă pașii, dar nu pot schimba ordinea. Este opusul lui Strategy: acolo înlocuiești tot algoritmul prin compunere, aici înlocuiești pași prin moștenire.

Iterator. Range-for (`for (int x : colectie)`) funcționează cu orice obiect care are `begin()` și `end()`, întorcând iteratori cu `*`, `++` și `!=`. Mai jos, `Interval` nu reține nicio valoare: le **calculează** pe măsură ce sunt cerute.

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

struct Rand { string nume; int nota; };

class Raport                                // Template Method: scheletul algoritmului e fix
{
public:
    void genereaza(const vector<Rand> &date) const   // nevirtuală: ordinea pașilor nu se schimbă
    {
        antet();
        for (const auto &r : date) linie(r);
        subsol(date.size());
    }
    virtual ~Raport() = default;
protected:
    virtual void antet() const = 0;         // pașii pe care îi completează derivatele
    virtual void linie(const Rand &r) const = 0;
    virtual void subsol(size_t n) const { cout << "(" << n << " randuri)\n"; }   // pas cu variantă implicită
};

class RaportText : public Raport
{
protected:
    void antet() const override { cout << "== Situatie note ==\n"; }
    void linie(const Rand &r) const override { cout << r.nume << ": " << r.nota << '\n'; }
};

class RaportCsv : public Raport
{
protected:
    void antet() const override { cout << "nume,nota\n"; }
    void linie(const Rand &r) const override { cout << r.nume << ',' << r.nota << '\n'; }
    void subsol(size_t) const override {}  // CSV-ul nu are subsol
};

// --- Iterator: o colecție „virtuală” parcursă cu range-for, fără să rețină elementele ---
class Interval
{
    int a, b, pas;
public:
    Interval(int st, int dr, int p = 1) : a(st), b(dr), pas(p) {}

    class iterator
    {
        int x, pas;
    public:
        iterator(int v, int p) : x(v), pas(p) {}
        int operator*() const { return x; }
        iterator& operator++() { x += pas; return *this; }
        bool operator!=(const iterator &o) const { return x < o.x; }   // ne oprim când trecem de sfârșit
    };
```

```
    iterator begin() const { return iterator(a, pas); }
    iterator end() const { return iterator(b + 1, pas); }
};

int main()
{
    vector<Rand> date = {{"Ana", 10}, {"Dan", 8}};
    RaportText().genereaza(date);
    RaportCsv().genereaza(date);

    int st, dr, pas;
    cin >> st >> dr >> pas;
    for (int x : Interval(st, dr, pas)) cout << x << ' ';   // range-for folosește begin(), end(), !=, ++, *
    cout << '\n';
    return 0;
}
```

DE CE AȘA?

De ce `genereaza` nu este virtuală? Ca nicio derivată să nu poată încurca ordinea (antet, linii, subsol): este garanția oferită de bază. Este și ideea **NVI** (*Non-Virtual Interface*) din oop-04. **De ce `x < o.x` în `operator!=`?** Cu pasul 3, valorile 1, 4, 7, 10, 13 nu ating exact `end` = 11; comparația „mai mic" oprește corect bucla. Un iterator standard complet are și tipuri asociate (`value_type`, `iterator_category`), necesare algoritmilor STL.

7. Cum alegi?

Dacă problema este...	Încearcă
„vreau să schimb algoritmul la rulare"	Strategy (sau `std::function`)
„mai multe module trebuie anunțate când se schimbă ceva"	Observer
„vreau undo/redo, cozi de sarcini, macro-uri"	Command
„comportamentul depinde de o stare cu multe tranziții"	State
„toate variantele urmează aceiași pași, în aceeași ordine"	Template Method
„vreau să parcurg o structură fără să-i expun interiorul"	Iterator

8. Greșeli frecvente

Greșeala	Problema	Corect
ierarhie de clase pentru o strategie de o linie	cod lung inutil	`std::function` + lambda

Greșeala	Problema	Corect
abonați rămași după distrugere	apel spre obiect mort	dezabonare în destructor, `weak_ptr`
dezabonare în timpul notificării	iterator invalid	copie a listei sau ștergere amânată
comandă care nu ține minte ce a șters	`undo` imposibil	salvează starea necesară în comandă
refacerile nu se golesc după o acțiune nouă	`redo` aplică ceva fără sens	golește stiva de refaceri
`switch` pe stare în toate metodele	tranziții greu de urmărit	State sau tabel de tranziții
Template Method cu metoda-schelet virtuală	derivatele pot strica ordinea	schelet nevirtual
iterator cu `!=` exact când pasul sare peste capăt	buclă infinită	comparație `<` sau `end` calculat exact

9. Exersează pe hârtie

1. Ce afișează programul cu reduceri pentru prețurile `150 60`?

2. Scrie strategia `BlackFriday` (30% reducere, dar nu mai mult de 100 de lei).

3. În programul cu senzorul, ce s-ar afișa dacă afișajul nu s-ar dezabona deloc?

4. De ce lambda pentru statistică poate captura `[&]` fără risc aici? Când ar deveni periculos?

5. Urmărește editorul pe: `add ab`, `add cd`, `del 3`, `undo`, `undo`, `print`.

6. Scrie comanda `Majuscule` (transformă tot textul în majuscule). Ce trebuie să rețină ca să poată fi anulată?

7. În automatul comenzii, ce se întâmplă la `plata, livrare, anulare`? Desenează traseul.

8. Adaugă starea „în livrare" între „plătită" și „livrată". Ce clase modifici?

9. Ce afișează `for (int x : Interval(2, 9, 4)) cout << x << ' ';`?

10. Care este diferența dintre Strategy și Template Method?

10. Mini-grile

1. Undo/redo se implementează de obicei cu: a) Command b) Singleton c) Adapter d) Composite

2. Observer este potrivit când: a) un obiect trebuie anunțat de mai multe ori b) mai mulți abonați trebuie anunțați când un subiect se schimbă c) creezi obiecte complexe d) ai o singură instanță

3. În Template Method, metoda-schelet este de obicei: a) virtuală pură b) nevirtuală c) statică d) privată și virtuală

4. Pentru ca range-for să meargă pe clasa ta, ai nevoie de: a) `operator[]` b) `size()` c) `begin()` și `end()` care întorc iteratori d) moștenire din `vector`

11. Mai departe (pentru cine vrea mai mult)

- **Visitor:** operații noi pe o ierarhie fără a o modifica; varianta modernă `std::visit` pe `std::variant`.
- **Chain of Responsibility:** cererea trece printr-un lanț de handler-e (middleware).
- **Mediator:** obiectele comunică printr-un intermediar, nu direct.
- **Memento:** salvarea și restaurarea stării (alternativă pentru undo).
- **Programare reactivă** și sisteme de evenimente bazate pe Observer.

Rezumat

- **Strategy:** algoritm interschimbabil prin interfață sau `std::function`.
- **Observer:** subiectul anunță lista de abonați; atenție la abonați morți și la modificarea listei în timpul notificării.
- **Command:** acțiuni ca obiecte cu `executa` / `anuleaza`; două stive pentru undo/redo.
- **State:** fiecare stare este o clasă care decide tranzițiile; contextul deleagă.
- **Template Method:** schelet nevirtual, pași virtuali; **Iterator:** `begin` / `end` + `*`, `++`, `!=` pentru range-for.

Citește și exersează în plus

- refactoring.guru — Behavioral Design Patterns — toate șabloanele comportamentale, ilustrate (engleză) https://refactoring.guru/design-patterns/behavioral-patterns
- cppreference — std::function — baza variantelor moderne pentru Strategy, Observer, Command (engleză) https://en.cppreference.com/w/cpp/utility/functional/function
- cppreference — Range-based for loop — ce cere range-for de la un iterator (engleză) https://en.cppreference.com/w/cpp/language/range-for

Răspunsuri

Exersează pe hârtie

1. `fara reducere: 210`, `10% reducere: 189`, `minus 50 peste 200: 160`.

2. `class BlackFriday : public StrategieReducere { public: double aplica(double t) const override { return t - min(t * 0.3, 100.0); } string nume() const override { return "black friday"; } };`

3. Încă două linii `afisaj: 28` și `afisaj: 35`, fiecare înaintea mesajelor celorlalți abonați pentru aceeași valoare (alarma la 35).

4. Variabilele `suma` și `numar` trăiesc în `main` cel puțin cât senzorul. Ar fi periculos dacă senzorul ar trăi mai mult (de exemplu, ar fi creat în altă funcție și întors, iar variabilele capturate ar muri).

5. `ab` → `abcd` → `a` (șterse „bcd") → undo: `abcd` → undo: `ab`. Se afișează `"ab"`.

6. Trebuie să rețină textul de dinainte (sau pozițiile literelor care erau mici), ca `anuleaza()` să-l poată restaura: `executa() { vechi = doc.text; … }`, `anuleaza() { doc.text = vechi; }`.

7. noua → (plata) platita → (livrare) livrata → anulare: „nu se poate anula în starea livrata", rămâne livrata.

8. Clasa nouă `InLivrare` (cu `livreaza()` → `Livrata`, eventual `anuleaza()`); în `Platita`, `livreaza()` întoarce acum `InLivrare`. Celelalte stări și `Comanda` nu se schimbă.

9. `2 6` (următorul ar fi 10 > 9).

10. Strategy înlocuiește întregul algoritm prin compunere (obiectul ține o strategie, schimbabilă la rulare). Template Method fixează scheletul în bază și lasă derivatele să schimbe pași, prin moștenire (ales la compilare, odată cu tipul).

Mini-grile

1. a) Command. **2.** b). **3.** b) nevirtuală. **4.** c).